python_data_science_toolbox_pt2

April 5, 2019

1 Intro to Iterators

Iterables - "An object that has an associated `iter()` method". Lists, strings and dictionaries can be looped over with simple for loop. Applying an `iter()` to an iterable creates an iterator. This is the `for` loop under the hood

Iterator - Produces the consecutive value with `next()`

ex:

```
word = 'Da'
it = iter(word)
next(it) -> 'D'
next(it) -> 'a'
```

or can you asterisk "*" also called "splat" operator to print out all values

```
word = 'Data'
it = iter(word)
print(*it) -> D a t a
```

Once all values of an iterator are printed out, returns error if you try to use `next()` again.

Review: `Iterable` is an object that can return an `iterator`, while an `iterator` is an object that keeps state and produces the next value when you call `next()` on it.

```
In [1]: # Practice with iterators and iterables
        flash1 = ['jay garrick', 'barry allen', 'wally west', 'bart allen']
        flash2 = iter(flash1)
        print(next(flash2))
        print(next(flash2))
```

```
jay garrick
barry allen
```

Above, `flash1` is an iterable and `flash2` is an iterator.

More below

```
In [2]: # Create a list of strings: flash
        flash = ['jay garrick', 'barry allen', 'wally west', 'bart allen']

        # Print each list item in flash using a for loop
        for person in flash:
            print(person)

        # Create an iterator for flash: superspeed
        superspeed = iter(flash)

        # Print each item from the iterator
        print(next(superspeed))
        print(next(superspeed))
        print(next(superspeed))
        print(next(superspeed))
```

```
jay garrick
barry allen
wally west
bart allen
jay garrick
barry allen
wally west
bart allen
```

Not all iterables are actual lists. A couple of examples that we looked at are strings and the use of the `range()` function. In this exercise, we will focus on the `range()` function.

You can use `range()` in a for loop as if it's a list to be iterated over:

```
for i in range(5):
    print(i)
```

```
In [3]: # Create an iterator for range(3): small_value
        small_value = iter(range(3))

        # Print the values in small_value
        print(next(small_value))
        print(next(small_value))
        print(next(small_value))

        # Loop over range(3) and print the values
        for i in range(3):
            print(i)


        # Create an iterator for range(10 ** 100): googol
        googol = iter(range(10 ** 100))
```

```
        # Print the first 5 values from googol
        print(next(googol))
        print(next(googol))
        print(next(googol))
        print(next(googol))
        print(next(googol))
```

```
0
1
2
0
1
2
0
1
2
3
4
```

Function like `list()` and `sum()` take iterators as arguments

```
In [4]: # Create a range object: values
        values = range(10,21)

        # Print the range object
        print(values)

        # Create a list of integers: values_list
        values_list = list(values)

        # Print values_list
        print(values_list)

        # Get the sum of values: values_sum
        values_sum = sum(values)

        # Print values_sum
        print(values_sum)
```

```
range(10, 21)
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
165
```

1.1 Useful functions for iterables

- `enumerate()` - returns an enumerate object that produces a sequence of tuples, and each of the tuples is an index-value pair. You can also change where in the iterable it starts from

- `zip()` - Takes any number of iterables and returns a zip object that is an iterator of tuples. If you wanted to print the values of a zip object, you can convert it into a list and then print it. Printing just a zip object will not return the values unless you unpack it first.

```
In [5]: # Create a list of strings: mutants
        mutants = ['charles xavier',
                   'bobby drake',
                   'kurt wagner',
                   'max eisenhardt',
                   'kitty pryde']

        # Create a list of tuples: mutant_list
        mutant_list = list(enumerate(mutants))

        # Print the list of tuples
        print(mutant_list)

        # Unpack and print the tuple pairs
        for index1, value1 in enumerate(mutants):
            print(index1, value1)

        # Change the start index
        for index2, value2 in enumerate(mutants, start = 1):
            print(index2, value2)
```

```
[(0, 'charles xavier'), (1, 'bobby drake'), (2, 'kurt wagner'), (3, 'max eisenhardt'), (4, 'kit
0 charles xavier
1 bobby drake
2 kurt wagner
3 max eisenhardt
4 kitty pryde
1 charles xavier
2 bobby drake
3 kurt wagner
4 max eisenhardt
5 kitty pryde
```

```
In [6]: mutants = ['charles xavier',
         'bobby drake',
         'kurt wagner',
         'max eisenhardt',
         'kitty pryde']
        aliases = ['prof x', 'iceman', 'nightcrawler', 'magneto', 'shadowcat']
        powers = ['telepathy',
         'thermokinesis',
         'teleportation',
         'magnetokinesis',
```

```
 'intangibility']

# Create a list of tuples: mutant_data
mutant_data = list(zip(mutants, aliases, powers))

# Print the list of tuples
print(mutant_data)

# Create a zip object using the three lists: mutant_zip
mutant_zip = zip(mutants, aliases, powers)

# Print the zip object
print(mutant_zip)

# Unpack the zip object and print the tuple values
for value1, value2, value3 in mutant_zip:
    print(value1, value2, value3)
```

```
[('charles xavier', 'prof x', 'telepathy'), ('bobby drake', 'iceman', 'thermokinesis'), ('kurt
<zip object at 0x112885988>
charles xavier prof x telepathy
bobby drake iceman thermokinesis
kurt wagner nightcrawler teleportation
max eisenhardt magneto magnetokinesis
kitty pryde shadowcat intangibility
```

We can reverse what has been zipped together by using `zip()` with a little help from asterisk `*`! Asterisk `*` unpacks an iterable such as a list or a tuple into positional arguments in a function call.

```
In [7]: mutants = ('charles xavier',
         'bobby drake',
         'kurt wagner',
         'max eisenhardt',
         'kitty pryde')
        powers = ('telepathy',
         'thermokinesis',
         'teleportation',
         'magnetokinesis',
         'intangibility')

        # Create a zip object from mutants and powers: z1
        z1 = zip(mutants, powers)

        # Print the tuples in z1 by unpacking with *
        print(*z1)
```

```
        # Re-create a zip object from mutants and powers: z1
        z1 = zip(mutants, powers)

        # 'Unzip' the tuples in z1 by unpacking with * and zip(): result1, result2
        result1, result2 = zip(*z1)

        # Check if unpacked tuples are equivalent to original tuples
        print(result1 == mutants)
        print(result2 == powers)
```

```
('charles xavier', 'telepathy') ('bobby drake', 'thermokinesis') ('kurt wagner', 'teleportatio
True
True
```

1.2 Using iterators for big data

Sometimes, the data we have to process reaches a size that is too much for a computer's memory to handle. This is a common problem faced by data scientists. A solution to this is to process an entire data source chunk by chunk, instead of a single go all at once.

With large data, you can chunk the data and treat it as an iterator. Performing a variety of functions per chunk/element

Use the argument `chunk_size` within `pd.read_csv()`

```
In [9]: import pandas as pd

        # Initialize an empty dictionary: counts_dict
        counts_dict = dict()

        # Iterate over the file chunk by chunk
        for chunk in pd.read_csv("tweets.csv", chunksize = 10):

            # Iterate over the column in DataFrame
            for entry in chunk['lang']:
                if entry in counts_dict.keys():
                    counts_dict[entry] += 1
                else:
                    counts_dict[entry] = 1

        # Print the populated dictionary
        print(counts_dict)
```

```
{'en': 97, 'et': 1, 'und': 2}
```

Now we'll do the same thing as above but put it in the form of a function

```
In [10]: # Define count_entries()
         def count_entries(csv_file, c_size, colname):
```

```
            """Return a dictionary with counts of
            occurrences as value for each key."""

            # Initialize an empty dictionary: counts_dict
            counts_dict = {}

            # Iterate over the file chunk by chunk
            for chunk in pd.read_csv(csv_file, chunksize= c_size):

                # Iterate over the column in DataFrame
                for entry in chunk[colname]:
                    if entry in counts_dict.keys():
                        counts_dict[entry] += 1
                    else:
                        counts_dict[entry] = 1

            # Return counts_dict
            return counts_dict

        # Call count_entries(): result_counts
        result_counts = count_entries('tweets.csv', 10, 'lang')

        # Print result_counts
        print(result_counts)
```

```
{'en': 97, 'et': 1, 'und': 2}
```

2 List Comprehensions

A simple and efficient method for creating new iterables (such as lists) into a single line of code. Much more efficient than `for` loops

It can also replace nested `for` loops

Tradeoff of list comprehensions is readability

A basic example is producing a list of the first character in each string of a list. How can we get the second list from the first list?

`['house', 'cuddy', 'chase', 'thirteen', 'wilson']`

to

`['h', 'c', 'c', 't', 'w']`

```
In [11]: # Basic example
         doctor = ['house', 'cuddy', 'chase', 'thirteen', 'wilson']

         # Produce the second list
         [doc[0] for doc in doctor]
```

```
Out[11]: ['h', 'c', 'c', 't', 'w']
```

All below can have list comprehensions built over them

```
doctor = ['house', 'cuddy', 'chase', 'thirteen', 'wilson']

range(50)

underwood = 'After all, we are nothing more or less than what we choose to reveal.'

jean = '24601'

flash = ['jay garrick', 'barry allen', 'wally west', 'bart allen']
```

But `valjean = 24601` can't have a list comprehension built over it

```
In [12]: # A simple list comprehension, a list with numbers 0-9
         # and squaring all the results
         squares = [i**2 for i in range(10)]

         print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

General build is `[<operation> for <iterator> in <iterable>]`

2.0.1 Nested list comprehensions

Now to build a list comprehension WITHIN a list comprehension. This is to replace a nested loop

A common use of nested lists is representing multi-dimensional data such as matrices

```
matrix = [[0, 1, 2, 3, 4],
          [0, 1, 2, 3, 4],
          [0, 1, 2, 3, 4],
          [0, 1, 2, 3, 4],
          [0, 1, 2, 3, 4]]
```

Your task is to recreate this matrix by using nested listed comprehensions. Recall that you can create one of the rows of the matrix with a single list comprehension. To create the list of lists, you simply have to supply the list comprehension as the output expression of the overall list comprehension:

`[[output expression] for iterator variable in iterable]`

Here, for a nested list comprehension, 'output expression' is itself a list comprehension

```
In [13]: # Create a 5 x 5 matrix using a list of lists: matrix
         matrix = [[col for col in range(5)] for row in range(5)]

         # Print the matrix
         for row in matrix:
             print(row)
```

```
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
```

2.0.2 Advanced Comprehensions using Conditionals

An interesting mechanism in list comprehensions is that you can also create lists with values that meet only a certain condition. One way of doing this is by using conditionals on iterator variables.

`[ <output expression> for <iterator variable> in <iterable> if <predicate expression> ]`

```
In [14]: # Create a list of strings: fellowship
         fellowship = ['frodo', 'samwise', 'merry', 'aragorn', 'legolas', 'boromir', 'gimli']

         # Create list comprehension: new_fellowship
         new_fellowship = [member for member in fellowship if len(member) >= 7]

         # Print the new list
         print(new_fellowship)
```

```
['samwise', 'aragorn', 'legolas', 'boromir']
```

Now to make things trickier, instead of getting rid of strings with less than 7 characters, replace them with an empty string

```
In [15]: # Create a list of strings: fellowship
         fellowship = ['frodo', 'samwise', 'merry', 'aragorn', 'legolas', 'boromir', 'gimli']

         # Create list comprehension: new_fellowship
         new_fellowship = [member if len(member) >= 7 else "" for member in fellowship]

         # Print the new list
         print(new_fellowship)
```

```
['', 'samwise', '', 'aragorn', 'legolas', 'boromir', '']
```

2.0.3 Dictionary comprehensions

There are many other objects you can build using comprehensions, such as dictionaries, pervasive objects in Data Science. You will create a dictionary using the comprehension syntax for this exercise. In this case, the comprehension is called a **dict comprehension**.

Recall that the main difference between a list comprehension and a dict comprehension is the use of curly braces `{}` instead of `[]`. Additionally, members of the dictionary are created using a colon `:`, as in `<key> : <value>`

```
In [16]: # Create a list of strings: fellowship
         fellowship = ['frodo', 'samwise', 'merry', 'aragorn', 'legolas', 'boromir', 'gimli']

         # Create dict comprehension: new_fellowship
         new_fellowship = {member: len(member) for member in fellowship}

         # Print the new list
         print(new_fellowship)
```

```
{'frodo': 5, 'samwise': 7, 'merry': 5, 'aragorn': 7, 'legolas': 7, 'boromir': 7, 'gimli': 5}
```

3 Generator Expressions

Related to comprehensions. Generators are created similarly but use parentheses `()` instead of brackets `[]`

Generator objects are not stored in memory but they can be iterated over

Helps when working with large amounts of data so not all are stored in memory but bits by bits are. You can proceed through the elements of a generator object with the `next()` function

Examples of generators include the `.items()` method for looping over dictionaries and the `range()` function

```
In [17]: # List of strings
         fellowship = ['frodo', 'samwise', 'merry', 'aragorn', 'legolas', 'boromir', 'gimli']

         # List comprehension
         fellow1 = [member for member in fellowship if len(member) >= 7]

         # Generator expression
         fellow2 = (member for member in fellowship if len(member) >= 7)
```

```
In [18]: # Create generator object: result
         result = (num for num in range(31))

         # Print the first 5 values
         print(next(result))
         print(next(result))
         print(next(result))
         print(next(result))
         print(next(result))

         # Print the rest of the values
         for value in result:
             print(value)
```

```
0
1
2
```

```
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
```

In [19]:
```
# Create a list of strings: lannister
lannister = ['cersei', 'jaime', 'tywin', 'tyrion', 'joffrey']

# Create a generator object: lengths
lengths = (len(person) for person in lannister)

# Iterate over and print the values in lengths
for value in lengths:
    print(value)
```

```
6
5
5
6
7
```

3.0.1 Generator functions

Functions that, like generator expressions, yield a series of values, instead of returning a single value. A generator function is defined as you do a regular function, but whenever it generates a value, it uses the keyword `yield` instead of `return`

```
In [20]: # Create a list of strings
         lannister = ['cersei', 'jaime', 'tywin', 'tyrion', 'joffrey']

         # Define generator function get_lengths
         def get_lengths(input_list):
             """Generator function that yields the
             length of the strings in input_list."""

             # Yield the length of a string
             for person in input_list:
                 yield len(person)

         # Print the values generated by get_lengths()
         for value in get_lengths(lannister):
             print(value)
```

```
6
5
5
6
7
```

You will be using a list comprehension to extract the time from time-stamped Twitter data.

```
In [21]: df = pd.read_csv('tweets.csv')

         # Extract the created_at column from df: tweet_time
         tweet_time = df.created_at

         # Extract the clock time: tweet_clock_time
         tweet_clock_time = [entry[11:19] for entry in tweet_time]

         # Print the extracted times
         print(tweet_clock_time)
```

Add a conditional expression to the list comprehension so that you only select the times in which `entry[17:19]` is equal to '19'

```
In [ ]: # Extract the created_at column from df: tweet_time
        tweet_time = df['created_at']

        # Extract the clock time: tweet_clock_time
```

```
tweet_clock_time = [entry[11:19] for entry in tweet_time if entry[17:19] == '19']

# Print the extracted times
print(tweet_clock_time)
```

4 Case Study

Bring all this together using World Bank Data

First below shows how to transform a zip object into a dictionary

```
In [ ]: feature_names = ['CountryName',
         'CountryCode',
         'IndicatorName',
         'IndicatorCode',
         'Year',
         'Value']

        row_vals = ['Arab World',
         'ARB',
         'Adolescent fertility rate (births per 1,000 women ages 15-19)',
         'SP.ADO.TFRT',
         '1960',
         '133.56090740552298']

        # Zip lists: zipped_lists
        zipped_lists = zip(feature_names, row_vals)

        # Create a dictionary: rs_dict
        rs_dict = dict(zipped_lists)

        # Print the dictionary
        print(rs_dict)
```

Same as above but in the form of a function

```
In [ ]: # Define lists2dict()
        def lists2dict(list1, list2):
            """Return a dictionary where list1 provides
            the keys and list2 provides the values."""

            # Zip lists: zipped_lists
            zipped_lists = zip(list1, list2)

            # Create a dictionary: rs_dict
            rs_dict = dict(zipped_lists)

            # Return the dictionary
            return rs_dict
```

```
# Call lists2dict: rs_fxn
rs_fxn = lists2dict(feature_names, row_vals)

# Print rs_fxn
print(rs_fxn)
```

Now using list comprehensions

```
In [23]: # @hidden_cell
         row_lists = [['Arab World',
           'ARB',
           'Adolescent fertility rate (births per 1,000 women ages 15-19)',
           'SP.ADO.TFRT',
           '1960',
           '133.56090740552298'],
          ['Arab World',
           'ARB',
           'Age dependency ratio (% of working-age population)',
           'SP.POP.DPND',
           '1960',
           '87.7976011532547'],
          ['Arab World',
           'ARB',
           'Age dependency ratio, old (% of working-age population)',
           'SP.POP.DPND.OL',
           '1960',
           '6.634579191565161'],
          ['Arab World',
           'ARB',
           'Age dependency ratio, young (% of working-age population)',
           'SP.POP.DPND.YG',
           '1960',
           '81.02332950839141'],
          ['Arab World',
           'ARB',
           'Arms exports (SIPRI trend indicator values)',
           'MS.MIL.XPRT.KD',
           '1960',
           '3000000.0'],
          ['Arab World',
           'ARB',
           'Arms imports (SIPRI trend indicator values)',
           'MS.MIL.MPRT.KD',
           '1960',
           '538000000.0'],
          ['Arab World',
           'ARB',
```

```
 'Birth rate, crude (per 1,000 people)',
 'SP.DYN.CBRT.IN',
 '1960',
 '47.697888095096395'],
['Arab World',
 'ARB',
 'CO2 emissions (kt)',
 'EN.ATM.CO2E.KT',
 '1960',
 '59563.9892169935'],
['Arab World',
 'ARB',
 'CO2 emissions (metric tons per capita)',
 'EN.ATM.CO2E.PC',
 '1960',
 '0.6439635478877049'],
['Arab World',
 'ARB',
 'CO2 emissions from gaseous fuel consumption (% of total)',
 'EN.ATM.CO2E.GF.ZS',
 '1960',
 '5.041291753975099'],
['Arab World',
 'ARB',
 'CO2 emissions from liquid fuel consumption (% of total)',
 'EN.ATM.CO2E.LF.ZS',
 '1960',
 '84.8514729446567'],
['Arab World',
 'ARB',
 'CO2 emissions from liquid fuel consumption (kt)',
 'EN.ATM.CO2E.LF.KT',
 '1960',
 '49541.707291032304'],
['Arab World',
 'ARB',
 'CO2 emissions from solid fuel consumption (% of total)',
 'EN.ATM.CO2E.SF.ZS',
 '1960',
 '4.72698138789597'],
['Arab World',
 'ARB',
 'Death rate, crude (per 1,000 people)',
 'SP.DYN.CDRT.IN',
 '1960',
 '19.7544519237187'],
['Arab World',
 'ARB',
```

```
 'Fertility rate, total (births per woman)',
 'SP.DYN.TFRT.IN',
 '1960',
 '6.92402738655897'],
['Arab World',
 'ARB',
 'Fixed telephone subscriptions',
 'IT.MLT.MAIN',
 '1960',
 '406833.0'],
['Arab World',
 'ARB',
 'Fixed telephone subscriptions (per 100 people)',
 'IT.MLT.MAIN.P2',
 '1960',
 '0.6167005703199'],
['Arab World',
 'ARB',
 'Hospital beds (per 1,000 people)',
 'SH.MED.BEDS.ZS',
 '1960',
 '1.9296220724398703'],
['Arab World',
 'ARB',
 'International migrant stock (% of population)',
 'SM.POP.TOTL.ZS',
 '1960',
 '2.9906371279862403'],
['Arab World',
 'ARB',
 'International migrant stock, total',
 'SM.POP.TOTL',
 '1960',
 '3324685.0']]
```

In [ ]:
```
# Print the first two lists in row_lists
print(row_lists[0])
print(row_lists[1])

# Turn list of lists into list of dicts: list_of_dicts
list_of_dicts = [lists2dict(feature_names, sublist) for sublist in row_lists]

# Print the first two dictionaries in list_of_dicts
print(list_of_dicts[0])
print(list_of_dicts[1])
```

In [ ]:
```
# Import the pandas package
import pandas as pd
```

```
# Turn list of lists into list of dicts: list_of_dicts
list_of_dicts = [lists2dict(feature_names, sublist) for sublist in row_lists]

# Turn list of dicts into a DataFrame: df
df = pd.DataFrame(list_of_dicts)

# Print the head of the DataFrame
print(df.head())
```

4.0.1 Python Generators for streaming data

You will process the first 1000 rows of a file line by line, to create a dictionary of the counts of how many times each country appears in a column in the dataset.

To begin, you need to open a connection to this file using what is known as a context manager. For example, the command with `open('datacamp.csv')` as datacamp binds the csv file 'datacamp.csv' as datacamp in the **context manager**. Here, the with statement is the context manager, and its purpose is to ensure that resources are efficiently allocated when opening a connection to a file.

```
In [1]: # Open a connection to the file
        with open('world_ind_pop_data.csv') as file:

            # Skip the column names
            file.readline()

            # Initialize an empty dictionary: counts_dict
            counts_dict = {}

            # Process only the first 1000 rows
            for j in range(1000):

                # Split the current line into a list: line
                line = file.readline().split(',')

                # Get the value for the first column: first_col
                first_col = line[0]

                # If the column value is in the dict, increment its value
                if first_col in counts_dict.keys():
                    counts_dict[first_col] += 1

                # Else, add to the dict and set value to 1
                else:
                    counts_dict[first_col] = 1

        # Print the resulting dictionary
        print(counts_dict)
```

```
{'Arab World': 5, 'Caribbean small states': 5, 'Central Europe and the Baltics': 5, 'East Asia
```

You processed a file line by line for a given number of lines. What if, however, you want to do this for the entire file?

In this case, it would be useful to use generators. Generators allow users to lazily evaluate data. This concept of lazy evaluation is useful when you have to deal with very large datasets because it lets you generate values in an efficient manner by yielding only chunks of data at a time instead of the whole thing at once.

You will define a generator function read_large_file() that produces a generator object which yields a single line from a file each time `next()` is called on it. The csv file 'world_dev_ind.csv' is in your current directory for your use.

Note that when you open a connection to a file, the resulting file object is already a generator!

```
In [ ]: # Define read_large_file()
        def read_large_file(file_object):
            """A generator function to read a large file lazily."""

            # Loop indefinitely until the end of the file
            while True:

                # Read a line from the file: data
                data = file_object.readline()

                # Break if this is the end of the file
                if not data:
                    break

                # Yield the line of data
                yield data

        # Open a connection to the file
        with open('world_ind_pop_data.csv') as file:

            # Create a generator object for the file: gen_file
            gen_file = read_large_file(file)

            # Print the first three lines of the file
            print(next(gen_file))
            print(next(gen_file))
            print(next(gen_file))
```

Now let's use your generator function to process the World Bank dataset like you did previously. You will process the file line by line, to create a dictionary of the counts of how many times each country appears in a column in the dataset. For this exercise, however, you won't process just 1000 rows of data, you'll process the entire dataset!

```
In [ ]: # Initialize an empty dictionary: counts_dict
        counts_dict = {}
```

```
# Open a connection to the file
with open('world_ind_pop_data.csv') as file:

    # Iterate over the generator from read_large_file()
    for line in read_large_file(file):

        row = line.split(',')
        first_col = row[0]

        if first_col in counts_dict.keys():
            counts_dict[first_col] += 1
        else:
            counts_dict[first_col] = 1

# Print
print(counts_dict)
```

4.0.2 Pandas' read_csv() to stream data

Another way to read data too large to store in memory in chunks is to read the file in as DataFrames of a certain length, say, 100. For example, with the pandas package (imported as pd), you can do `pd.read_csv(filename, chunksize=100)`.

```
In [ ]: # Import the pandas package
        import pandas as pd

        # Initialize reader object: df_reader
        df_reader = pd.read_csv('world_ind_pop_data.csv', chunksize = 10)

        # Print two chunks
        print(next(df_reader))
        print(next(df_reader))
```

```
In [ ]: # Initialize reader object: urb_pop_reader
        urb_pop_reader = pd.read_csv('world_ind_pop_data.csv', chunksize = 1000)

        # Get the first DataFrame chunk: df_urb_pop
        df_urb_pop = next(urb_pop_reader)

        # Check out the head of the DataFrame
        print(df_urb_pop.head())

        # Check out specific country: df_pop_ceb
        df_pop_ceb = df_urb_pop[df_urb_pop['CountryCode'] == 'CEB']

        # Zip DataFrame columns of interest: pops
        pops = zip(df_pop_ceb['Total Population'], df_pop_ceb['Urban population (% of total)'])
```

```
        # Turn zip object into list: pops_list
        pops_list = list(pops)

        # Print pops_list
        print(pops_list)
```

In [ ]:
```
import matplotlib.pyplot as plt

# Code from previous exercise
urb_pop_reader = pd.read_csv('world_ind_pop_data.csv', chunksize=1000)
df_urb_pop = next(urb_pop_reader)
df_pop_ceb = df_urb_pop[df_urb_pop['CountryCode'] == 'CEB']
pops = zip(df_pop_ceb['Total Population'],
           df_pop_ceb['Urban population (% of total)'])
pops_list = list(pops)

# Use list comprehension to create new DataFrame column 'Total Urban Population'
df_pop_ceb['Total Urban Population'] = [int(i[0]*i[1]/100) for i in pops_list]

# Plot urban population data
df_pop_ceb.plot(kind="scatter", x = 'Year', y='Total Urban Population')
plt.show()
```

In [ ]:
```
# Initialize reader object: urb_pop_reader
urb_pop_reader = pd.read_csv('ind_pop_data.csv', chunksize=1000)

# Initialize empty DataFrame: data
data = pd.DataFrame()

# Iterate over each DataFrame chunk
for df_urb_pop in urb_pop_reader:

    # Check out specific country: df_pop_ceb
    df_pop_ceb = df_urb_pop[df_urb_pop['CountryCode'] == 'CEB']

    # Zip DataFrame columns of interest: pops
    pops = zip(df_pop_ceb['Total Population'],
               df_pop_ceb['Urban population (% of total)'])

    # Turn zip object into list: pops_list
    pops_list = list(pops)

    # Use list comprehension to create new DataFrame column 'Total Urban Population'
    df_pop_ceb['Total Urban Population'] = [int(tup[0] * tup[1] * 0.01) for tup in pops

    # Append DataFrame chunk to data: data
    data = data.append(df_pop_ceb)
```

```
        # Plot urban population data
        data.plot(kind='scatter', x='Year', y='Total Urban Population')
        plt.show()
```

You're going to define the function plot_pop() which takes two arguments: the filename of the file to be processed, and the country code of the rows you want to process in the dataset.

Because all of the previous code you've written in the previous exercises will be housed in plot_pop(), calling the function already does the following:

Loading of the file chunk by chunk, Creating the new column of urban population values, and Plotting the urban population data. That's a lot of work, but the function now makes it convenient to repeat the same process for whatever file and country code you want to process and visualize!

```
In [ ]: # Define plot_pop()
        def plot_pop(filename, country_code):

            # Initialize reader object: urb_pop_reader
            urb_pop_reader = pd.read_csv(filename, chunksize=1000)

            # Initialize empty DataFrame: data
            data = pd.DataFrame()

            # Iterate over each DataFrame chunk
            for df_urb_pop in urb_pop_reader:
                # Check out specific country: df_pop_ceb
                df_pop_ceb = df_urb_pop[df_urb_pop['CountryCode'] == country_code]

                # Zip DataFrame columns of interest: pops
                pops = zip(df_pop_ceb['Total Population'],
                           df_pop_ceb['Urban population (% of total)'])

                # Turn zip object into list: pops_list
                pops_list = list(pops)

                # Use list comprehension to create new DataFrame column 'Total Urban Populatio
                df_pop_ceb['Total Urban Population'] = [int(tup[0] * tup[1] * 0.01) for tup in

                # Append DataFrame chunk to data: data
                data = data.append(df_pop_ceb)

            # Plot urban population data
            data.plot(kind='scatter', x='Year', y='Total Urban Population')
            plt.show()

        # Set the filename: fn
        fn = 'ind_pop_data.csv'

        # Call plot_pop for country code 'CEB'
```

```
plot_pop('ind_pop_data.csv', 'CEB')

# Call plot_pop for country code 'ARB'
plot_pop('ind_pop_data.csv', 'ARB')
```