**Python Data Science Toolbox (Part 2)**

**Hugo Bowne-Anderson**

[**View Source Code**](https://github.com/mclix85/datacamp)

**Course Description**

In the second Python Data Science Toolbox course, you’ll continue to build your Python data science skills. First, you’ll learn about iterators, objects you have already encountered in the context of for loops. You’ll then learn about list comprehensions, which are extremely handy tools for all data scientists working in Python. You’ll end the course by working through a case study in which you’ll apply all the techniques you learned in both parts of this course.

**Using iterators in PythonLand**

You’ll learn all about iterators and iterables, which you have already worked with when writing for loops. You’ll learn some handy functions that will allow you to effectively work with iterators. And you’ll finish the chapter with a use case that is pertinent to the world of data science and dealing with large amounts of data—in this case, data from Twitter that you will load in chunks using iterators.

**Introduction to iterators**

**Iterators vs. Iterables**

Let’s do a quick recall of what you’ve learned about **iterables** and **iterators**. Recall from the video that an *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 this exercise, you will identify which object is an *iterable* and which is an *iterator*.

The environment has been pre-loaded with the variables flash1 and flash2. Try printing out their values with print() and next() to figure out which is an *iterable* and which is an *iterator*.

# edited/added

flash1 = ['jay garrick', 'barry allen', 'wally west', 'bart allen']

flash2 = iter(flash1)

* Both flash1 and flash2 are iterators.
* Both flash1 and flash2 are iterables.
* flash1 is an iterable and flash2 is an iterator.

Correct!

**Iterating over iterables (1)**

Great, you’re familiar with what iterables and iterators are! In this exercise, you will reinforce your knowledge about these by iterating over and printing from iterables and iterators.

You are provided with a list of strings flash. You will practice iterating over the list by using a for loop. You will also create an iterator for the list and access the values from the iterator.

 Create a for loop to loop over flash and print the values in the list. Use person as the loop variable.

 Create an *iterator* for the list flash and assign the result to superhero.

 Print each of the items from superhero using next() 4 times.

# 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)

## jay garrick

## barry allen

## wally west

## bart allen

# Create an iterator for flash: superhero

superhero = iter(flash)

# Print each item from the iterator

print(next(superhero))

## jay garrick

print(next(superhero))

## barry allen

print(next(superhero))

## wally west

print(next(superhero))

## bart allen

Great work!

**Iterating over iterables (2)**

One of the things you learned about in this chapter is that 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)

Recall that range() doesn’t actually create the list; instead, it creates a range object with an iterator that produces the values until it reaches the limit (in the example, until the value 4). If range() created the actual list, calling it with a value of 10100

may not work, especially since a number as big as that may go over a regular computer’s memory. The value 10100

is actually what’s called a **Googol** which is a 1 followed by a hundred 0s. That’s a huge number!

Your task for this exercise is to show that calling range() with 10100

won’t actually pre-create the list.

 Create an **iterator** object small\_value over range(3) using the function iter().

 Using a for loop, iterate over range(3), printing the value for every iteration. Use num as the loop variable.

 Create an **iterator** object googol over range(10 \*\* 100).

# Create an iterator for range(3): small\_value

small\_value = iter(range(3))

# Print the values in small\_value

print(next(small\_value))

## 0

print(next(small\_value))

## 1

print(next(small\_value))

## 2

# Loop over range(3) and print the values

for num in range(3):

print(num)

## 0

## 1

## 2

# Create an iterator for range(10 \*\* 100): googol

googol = iter(range(10 \*\* 100))

# Print the first 5 values from googol

print(next(googol))

## 0

print(next(googol))

## 1

print(next(googol))

## 2

print(next(googol))

## 3

print(next(googol))

## 4

Great work!

**Iterators as function arguments**

You’ve been using the iter() function to get an iterator object, as well as the next() function to retrieve the values one by one from the iterator object.

There are also functions that take iterators and iterables as arguments. For example, the list() and sum() functions return a list and the sum of elements, respectively.

In this exercise, you will use these functions by passing an iterable from range() and then printing the results of the function calls.

 Create a range object that would produce the values from 10 to 20 using range(). Assign the result to values.

 Use the list() function to create a list of values from the range object values. Assign the result to values\_list.

 Use the sum() function to get the sum of the values from 10 to 20 from the range object values. Assign the result to values\_sum.

# Create a range object: values

values = range(10, 21)

# Print the range object

print(values)

## range(10, 21)

# Create a list of integers: values\_list

values\_list = list(values)

# Print values\_list

print(values\_list)

## [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

# Get the sum of values: values\_sum

values\_sum = sum(values)

# Print values\_sum

print(values\_sum)

## 165

Great work!

**Playing with iterators**

**Using enumerate**

You’re really getting the hang of using iterators, great job!

You’ve just gained several new ideas on iterators from the last video and one of them is the enumerate() function. Recall that enumerate() returns an enumerate object that produces a sequence of tuples, and each of the tuples is an *index-value* pair.

In this exercise, you are given a list of strings mutants and you will practice using enumerate() on it by printing out a list of tuples and unpacking the tuples using a for loop.

 Create a list of tuples from mutants and assign the result to mutant\_list. Make sure you generate the tuples using enumerate() and turn the result from it into a list using list().

 Complete the first for loop by unpacking the tuples generated by calling enumerate() on mutants. Use index1 for the index and value1 for the value when unpacking the tuple.

 Complete the second for loop similarly as with the first, but this time change the starting index to start from 1 by passing it in as an argument to the start parameter of enumerate(). Use index2 for the index and value2 for the value when unpacking the tuple.

# 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)

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

# Unpack and print the tuple pairs

for index1, value1 in enumerate(mutants):

print(index1, value1)

## 0 charles xavier

## 1 bobby drake

## 2 kurt wagner

## 3 max eisenhardt

## 4 kitty pryde

# Change the start index

for index2, value2 in enumerate(mutants, start=1):

print(index2, value2)

## 1 charles xavier

## 2 bobby drake

## 3 kurt wagner

## 4 max eisenhardt

## 5 kitty pryde

Great work!

**Using zip**

Another interesting function that you’ve learned is zip(), which 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 this exercise, you will explore this for yourself.

Three lists of strings are pre-loaded: mutants, aliases, and powers. First, you will use list() and zip() on these lists to generate a list of tuples. Then, you will create a zip object using zip(). Finally, you will unpack this zip object in a for loop to print the values in each tuple. Observe the different output generated by printing the list of tuples, then the zip object, and finally, the tuple values in the for loop.

 Using zip() with list(), create a *list* of *tuples* from the three lists mutants, aliases, and powers (in that order) and assign the result to mutant\_data.

 Using zip(), create a *zip object* called mutant\_zip from the three lists mutants, aliases, and powers.

 Complete the for loop by unpacking the zip object you created and printing the tuple values. Use value1, value2, value3 for the values from each of mutants, aliases, and powers, in that order.

# edited/added

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)

## [('charles xavier', 'prof x', 'telepathy'), ('bobby drake', 'iceman', 'thermokinesis'), ('kurt wagner', 'nightcrawler', 'teleportation'), ('max eisenhardt', 'magneto', 'magnetokinesis'), ('kitty pryde', 'shadowcat', 'intangibility')]

# Create a zip object using the three lists: mutant\_zip

mutant\_zip = zip(mutants, aliases, powers)

# Print the zip object

print(mutant\_zip)

## <zip object at 0x7f87f08519c0>

# 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 wagner nightcrawler teleportation

## max eisenhardt magneto magnetokinesis

## kitty pryde shadowcat intangibility

Great work!

**Using \* and zip to ‘unzip’**

You know how to use zip() as well as how to print out values from a zip object. Excellent!

Let’s play around with zip() a little more. There is no *unzip* function for doing the reverse of what zip() does. We can, however, reverse what has been zipped together by using zip() with a little help from *!* unpacks an *iterable* such as a list or a tuple into *positional arguments* in a function call.

In this exercise, you will use \* in a call to zip() to unpack the tuples produced by zip().

Two tuples of strings, mutants and powers have been pre-loaded.

 Create a zip object by using zip() on mutants and powers, in that order. Assign the result to z1.

 Print the tuples in z1 by unpacking them into positional arguments using the \* operator in a print() call.

 Because the previous print() call would have exhausted the elements in z1, recreate the zip object you defined earlier and assign the result again to z1.

 ‘Unzip’ the tuples in z1 by unpacking them into positional arguments using the \* operator in a zip() call. Assign the results to result1 and result2, in that order.

 The last print() statements prints the output of comparing result1 to mutants and result2 to powers. Click Submit Answer to see if the unpacked result1 and result2 are equivalent to mutants and powers, respectively.

# Create a zip object from mutants and powers: z1

z1 = zip(mutants, powers)

# Print the tuples in z1 by unpacking with \*

print(\*z1)

## ('charles xavier', 'telepathy') ('bobby drake', 'thermokinesis') ('kurt wagner', 'teleportation') ('max eisenhardt', 'magnetokinesis') ('kitty pryde', 'intangibility')

# 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)

## False

print(result2 == powers)

## False

Great work!

**Using iterators to load large files into memory**

**Processing large amounts of Twitter 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.

In this exercise, you will do just that. You will process a large csv file of Twitter data in the same way that you processed ‘tweets.csv’ in [*Bringing it all together*](https://campus.datacamp.com/courses/python-data-science-toolbox-part-1/writing-your-own-functions?ex=12) exercises of the prequel course, but this time, working on it in chunks of 10 entries at a time.

If you are interested in learning how to access Twitter data so you can work with it on your own system, refer to [Part 2](https://www.datacamp.com/courses/importing-data-in-python-part-2) of the DataCamp course on Importing Data in Python.

The pandas package has been imported as pd and the file ‘tweets.csv’ is in your current directory for your use.

*Be aware that this is real data from Twitter and as such there is always a risk that it may contain profanity or other offensive content (in this exercise, and any following exercises that also use real Twitter data).*

 Initialize an empty dictionary counts\_dict for storing the results of processing the Twitter data.

 Iterate over the ‘tweets.csv’ file by using a for loop. Use the loop variable chunk and iterate over the call to pd.read\_csv() with a chunksize of 10.

 In the inner loop, iterate over the column ‘lang’ in chunk by using a for loop. Use the loop variable entry.

# edited/added

import pandas as pd

# Initialize an empty dictionary: counts\_dict

counts\_dict = {}

# Iterate over the file chunk by chunk

for chunk in pd.read\_csv('datasets/Python-Data-Science-Toolbox-Part-2/tweets.csv', chunksize=10): # edited/added

# 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}

Great work!

**Extracting information for large amounts of Twitter data**

Great job chunking out that file in the previous exercise. You now know how to deal with situations where you need to process a very large file and that’s a very useful skill to have!

It’s good to know how to process a file in smaller, more manageable chunks, but it can become very tedious having to write and rewrite the same code for the same task each time. In this exercise, you will be making your code more *reusable* by putting your work in the last exercise in a *function definition*.

The pandas package has been imported as pd and the file ‘tweets.csv’ is in your current directory for your use.

 Define the function count\_entries(), which has 3 parameters. The first parameter is csv\_file for the filename, the second is c\_size for the chunk size, and the last is colname for the column name.

 Iterate over the file in csv\_file file by using a for loop. Use the loop variable chunk and iterate over the call to pd.read\_csv(), passing c\_size to chunksize.

 In the inner loop, iterate over the column given by colname in chunk by using a for loop. Use the loop variable entry.

 Call the count\_entries() function by passing to it the filename ‘tweets.csv’, the size of chunks 10, and the name of the column to count, ‘lang’. Assign the result of the call to the variable result\_counts.

# 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('datasets/Python-Data-Science-Toolbox-Part-2/tweets.csv', 10, 'lang') # edited/added

# Print result\_counts

print(result\_counts)

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

Great work!

**List comprehensions and generators**

In this chapter, you’ll build on your knowledge of iterators and be introduced to list comprehensions, which allow you to create complicated lists—and lists of lists—in one line of code! List comprehensions can dramatically simplify your code and make it more efficient, and will become a vital part of your Python data science toolbox. You’ll then learn about generators, which are extremely helpful when working with large sequences of data that you may not want to store in memory, but instead generate on the fly.

**List comprehensions**

**Write a basic list comprehension**

In this exercise, you will practice what you’ve learned from the video about writing list comprehensions. You will write a list comprehension and identify the output that will be produced.

The following list has been pre-loaded in the environment.

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

How would a list comprehension that produces a list of the **first character** of each string in doctor look like? Note that the list comprehension uses doc as the iterator variable. What will the output be?

* The list comprehension is [for doc in doctor: doc[0]] and produces the list [‘h’, ‘c’, ‘c’, ‘t’, ‘w’].
* The list comprehension is [doc[0] in doctor] and produces the list [‘h’, ‘c’, ‘c’, ‘t’, ‘w’].
* The list comprehension is [doc[0] in doctor] and produces the list [‘h’, ‘c’, ‘c’, ‘t’, ‘w’].

Correct!

**List comprehension over iterables**

You know that list comprehensions can be built over iterables. Given the following objects below, which of these can we build list comprehensions over?

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']

valjean = 24601

* You can build list comprehensions over all the objects except the string of number characters jean.
* You can build list comprehensions over all the objects except the string lists doctor and flash.
* You can build list comprehensions over all the objects except range(50).
* You can build list comprehensions over all the objects except the integer object valjean.

Correct!

**Writing list comprehensions**

You now have all the knowledge necessary to begin writing list comprehensions! Your job in this exercise is to write a list comprehension that produces a list of the squares of the numbers ranging from 0 to 9.

 Using the range of numbers from 0 to 9 as your iterable and i as your iterator variable, write a list comprehension that produces a list of numbers consisting of the squared values of i.

# Create list comprehension: squares

squares = [i\*\*2 for i in range(0,10)]

Great work!

**Nested list comprehensions**

Great! At this point, you have a good grasp of the basic syntax of list comprehensions. Let’s push your code-writing skills a little further. In this exercise, you will be writing a list comprehension *within* another list comprehension, or nested list comprehensions. It sounds a little tricky, but you can do it!

Let’s step aside for a while from strings. One of the ways in which lists can be used are in representing multi-dimension objects such as **matrices**. Matrices can be represented as a list of lists in Python. For example a 5 x 5 matrix with values 0 to 4 in each row can be written as:

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*]

Note that here, the **output expression** is itself a list comprehension.

 In the inner list comprehension - that is, the **output expression** of the nested list comprehension - create a list of values from 0 to 4 using range(). Use col as the iterator variable.

 In the **iterable** part of your nested list comprehension, use range() to count 5 rows - that is, create a list of values from 0 to 4. Use row as the iterator variable; note that you won’t be needing this variable to create values in the list of lists.

# 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]

Great work!

**Advanced comprehensions**

**Using conditionals in comprehensions (1)**

You’ve been using list comprehensions to build lists of values, sometimes using operations to create these values.

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. In this exercise, you will do exactly that!

Recall from the video that you can apply a conditional statement to test the iterator variable by adding an if statement in the optional *predicate expression* part after the for statement in the comprehension:

[ *output expression* for *iterator variable* in *iterable* if *predicate expression* ].

You will use this recipe to write a list comprehension for this exercise. You are given a list of strings fellowship and, using a list comprehension, you will create a list that only includes the members of fellowship that have 7 characters or more.

 Use member as the iterator variable in the list comprehension. For the conditional, use len() to evaluate the iterator variable. Note that you only want strings with 7 characters or more.

# 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']

Great work!

**Using conditionals in comprehensions (2)**

In the previous exercise, you used an if conditional statement in the *predicate expression* part of a list comprehension to evaluate an iterator variable. In this exercise, you will use an if-else statement on the *output expression* of the list.

You will work on the same list, fellowship and, using a list comprehension and an if-else conditional statement in the output expression, create a list that keeps members of fellowship with 7 or more characters and replaces others with an empty string. Use member as the iterator variable in the list comprehension.

 In the output expression, keep the string as-is **if** the number of characters is >= 7, **else** replace it with an *empty string* - that is, ’’ or ““.

# 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', '']

Great work!

**Dict comprehensions**

Comprehensions aren’t relegated merely to the world of lists. 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>.

You are given a list of strings fellowship and, using a **dict comprehension**, create a dictionary with the members of the list as the keys and the length of each string as the corresponding values.

Create a dict comprehension where the key is a string in fellowship and the value is the length of the string. Remember to use the syntax <key> : <value> in the output expression part of the comprehension to create the members of the dictionary. Use member as the iterator variable.

# 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 dictionary

print(new\_fellowship)

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

Great work!

**Introduction to generator expressions**

**List comprehensions vs. generators**

You’ve seen from the videos that list comprehensions and generator expressions look very similar in their syntax, except for the use of parentheses () in generator expressions and brackets [] in list comprehensions.

In this exercise, you will recall the difference between list comprehensions and generators. To help with that task, the following code has been pre-loaded in the environment:

# 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)

Try to play around with fellow1 and fellow2 by figuring out their types and printing out their values. Based on your observations and what you can recall from the video, select from the options below the best description for the difference between list comprehensions and generators.

* List comprehensions and generators are not different at all; they are just different ways of writing the same thing.
* A list comprehension produces a list as output, a generator produces a generator object.
* A list comprehension produces a list as output that can be iterated over, a generator produces a generator object that can’t be iterated over.

Correct!

**Write your own generator expressions**

You are familiar with what generators and generator expressions are, as well as its difference from list comprehensions. In this exercise, you will practice building generator expressions on your own.

Recall that generator expressions basically have the same syntax as list comprehensions, except that it uses parentheses () instead of brackets []; this should make things feel familiar! Furthermore, if you have ever iterated over a dictionary with .items(), or used the range() function, for example, you have already encountered and used generators before, without knowing it! When you use these functions, Python creates generators for you behind the scenes.

Now, you will start simple by creating a generator object that produces numeric values.

 Create a generator object that will produce values from 0 to 30. Assign the result to result and use num as the iterator variable in the generator expression.

 Print the first 5 values by using next() appropriately in print().

 Print the rest of the values by using a for loop to iterate over the generator object.

# Create generator object: result

result = (num for num in range(31))

# Print the first 5 values

print(next(result))

## 0

print(next(result))

## 1

print(next(result))

## 2

print(next(result))

## 3

print(next(result))

# Print the rest of the values

## 4

for value in result:

print(value)

## 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

Great work!

**Changing the output in generator expressions**

Great! At this point, you already know how to write a basic generator expression. In this exercise, you will push this idea a little further by adding to the output expression of a generator expression. Because generator expressions and list comprehensions are so alike in syntax, this should be a familiar task for you!

You are given a list of strings lannister and, using a generator expression, create a generator object that you will iterate over to print its values.

 Write a generator expression that will generate the **lengths** of each string in lannister. Use person as the iterator variable. Assign the result to lengths.

 Supply the correct iterable in the for loop for printing the values in the generator object.

# 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

Great work!

**Build a generator**

In previous exercises, you’ve dealt mainly with writing generator expressions, which uses comprehension syntax. Being able to use comprehension syntax for generator expressions made your work so much easier!

Now, recall from the video that not only are there generator expressions, there are *generator functions* as well. **Generator functions** are 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 this exercise, you will create a generator function with a similar mechanism as the generator expression you defined in the previous exercise:

lengths = (len(person) for person in lannister)

 Complete the function header for the function get\_lengths() that has a single parameter, input\_list.

 In the for loop in the function definition, yield the *length* of the strings in input\_list.

 Complete the iterable part of the for loop for printing the values generated by the get\_lengths() generator function. Supply the call to get\_lengths(), passing in the list lannister.

# 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

Great work!

**Wrapping up comprehensions and generators.**

**List comprehensions for time-stamped data**

You will now make use of what you’ve learned from this chapter to solve a simple data extraction problem. You will also be introduced to a data structure, the pandas **Series**, in this exercise. We won’t elaborate on it much here, but what you should know is that it is a data structure that you will be working with a lot of times when analyzing data from pandas DataFrames. You can think of DataFrame columns as single-dimension arrays called Series.

In this exercise, you will be using a list comprehension to extract the time from time-stamped Twitter data. The pandas package has been imported as pd and the file ‘tweets.csv’ has been imported as the df DataFrame for your use.

 Extract the column ‘created\_at’ from df and assign the result to tweet\_time. Fun fact: the extracted column in tweet\_time here is a Series data structure!

 Create a list comprehension that extracts the time from each row in tweet\_time. Each row is a string that represents a timestamp, and you will access the *12th to 19th characters* in the string to extract the time. Use entry as the *iterator variable* and assign the result to tweet\_clock\_time. Remember that Python uses 0-based indexing!

# edited/added

df = pd.read\_csv('datasets/Python-Data-Science-Toolbox-Part-2/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)

## ['23:40:17', '23:40:17', '23:40:17', '23:40:17', '23:40:17', '23:40:17', '23:40:18', '23:40:17', '23:40:18', '23:40:18', '23:40:18', '23:40:17', '23:40:18', '23:40:18', '23:40:17', '23:40:18', '23:40:18', '23:40:17', '23:40:18', '23:40:17', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:17', '23:40:18', '23:40:18', '23:40:17', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:18', '23:40:19', '23:40:18', '23:40:18', '23:40:18', '23:40:19', '23:40:19', '23:40:19', '23:40:18', '23:40:19', '23:40:19', '23:40:19', '23:40:18', '23:40:19', '23:40:19', '23:40:19', '23:40:18', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19']

Great work!

**Conditional list comprehensions for time-stamped data**

Great, you’ve successfully extracted the data of interest, the time, from a pandas DataFrame! Let’s tweak your work further by adding a conditional that further specifies which entries to select.

In this exercise, you will be using a list comprehension to extract the time from time-stamped Twitter data. You will add a conditional expression to the list comprehension so that you only select the times in which entry[17:19] is equal to ‘19’. The pandas package has been imported as pd and the file ‘tweets.csv’ has been imported as the df DataFrame for your use.

 Extract the column ‘created\_at’ from df and assign the result to tweet\_time.

 Create a list comprehension that extracts the time from each row in tweet\_time. Each row is a string that represents a timestamp, and you will access the *12th to 19th characters* in the string to extract the time. Use entry as the *iterator variable* and assign the result to tweet\_clock\_time. Additionally, add a conditional expression that checks whether entry[17:19] is equal to ‘19’.

# 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)

## ['23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19', '23:40:19']

Great work!

**Bringing it all together!**

This chapter will allow you to apply your newly acquired skills toward wrangling and extracting meaningful information from a real-world dataset—the World Bank’s World Development Indicators. You’ll have the chance to write your own functions and list comprehensions as you work with iterators and generators to solidify your Python data science chops.

**Welcome to the case study!**

**Dictionaries for data science**

For this exercise, you’ll use what you’ve learned about the zip() function and combine two lists into a dictionary.

These lists are actually extracted from a [bigger dataset file of world development indicators from the World Bank.](https://datacatalog.worldbank.org/search/dataset/0037712) For pedagogical purposes, we have pre-processed this dataset into the lists that you’ll be working with.

The first list feature\_names contains header names of the dataset and the second list row\_vals contains actual values of a row from the dataset, corresponding to each of the header names.

 Create a zip object by calling zip() and passing to it feature\_names and row\_vals. Assign the result to zipped\_lists.

 Create a dictionary from the zipped\_lists zip object by calling dict() with zipped\_lists. Assign the resulting dictionary to rs\_dict.

# edited/added

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)

## {'CountryName': 'Arab World', 'CountryCode': 'ARB', 'IndicatorName': 'Adolescent fertility rate (births per 1,000 women ages 15-19)', 'IndicatorCode': 'SP.ADO.TFRT', 'Year': '1960', 'Value': '133.56090740552298'}

Great work!

**Writing a function to help you**

Suppose you needed to repeat the same process done in the previous exercise to many, many rows of data. Rewriting your code again and again could become very tedious, repetitive, and unmaintainable.

In this exercise, you will create a function to house the code you wrote earlier to make things easier and much more concise. Why? This way, you only need to call the function and supply the appropriate lists to create your dictionaries! Again, the lists feature\_names and row\_vals are preloaded and these contain the header names of the dataset and actual values of a row from the dataset, respectively.

 Define the function lists2dict() with two parameters: first is list1 and second is list2.

 Return the resulting dictionary rs\_dict in lists2dict().

 Call the lists2dict() function with the arguments feature\_names and row\_vals. Assign the result of the function call to rs\_fxn.

# 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)

## {'CountryName': 'Arab World', 'CountryCode': 'ARB', 'IndicatorName': 'Adolescent fertility rate (births per 1,000 women ages 15-19)', 'IndicatorCode': 'SP.ADO.TFRT', 'Year': '1960', 'Value': '133.56090740552298'}

Great work!

**Using a list comprehension**

This time, you’re going to use the lists2dict() function you defined in the last exercise to turn a bunch of lists into a list of dictionaries with the help of a list comprehension.

The lists2dict() function has already been preloaded, together with a couple of lists, feature\_names and row\_lists. feature\_names contains the header names of the World Bank dataset and row\_lists is a list of lists, where each sublist is a list of actual values of a row from the dataset.

Your goal is to use a list comprehension to generate a list of dicts, where the *keys* are the header names and the *values* are the row entries.

 Inspect the contents of row\_lists by printing the first two lists in row\_lists.

 Create a list comprehension that generates a dictionary using lists2dict() for each sublist in row\_lists. The keys are from the feature\_names list and the values are the row entries in row\_lists. Use sublist as your iterator variable and assign the resulting list of dictionaries to list\_of\_dicts.

 Look at the first two dictionaries in list\_of\_dicts by printing them out.

# edited/added

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']]

# Print the first two lists in row\_lists

print(row\_lists[0])

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

print(row\_lists[1])

## ['Arab World', 'ARB', 'Age dependency ratio (% of working-age population)', 'SP.POP.DPND', '1960', '87.7976011532547']

# 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])

## {'CountryName': 'Arab World', 'CountryCode': 'ARB', 'IndicatorName': 'Adolescent fertility rate (births per 1,000 women ages 15-19)', 'IndicatorCode': 'SP.ADO.TFRT', 'Year': '1960', 'Value': '133.56090740552298'}

print(list\_of\_dicts[1])

## {'CountryName': 'Arab World', 'CountryCode': 'ARB', 'IndicatorName': 'Age dependency ratio (% of working-age population)', 'IndicatorCode': 'SP.POP.DPND', 'Year': '1960', 'Value': '87.7976011532547'}

Great work!

**Turning this all into a DataFrame**

You’ve zipped lists together, created a function to house your code, and even used the function in a list comprehension to generate a list of dictionaries. That was a lot of work and you did a great job!

You will now use all of these to convert the list of dictionaries into a pandas DataFrame. You will see how convenient it is to generate a DataFrame from dictionaries with the DataFrame() function from the pandas package.

The lists2dict() function, feature\_names list, and row\_lists list have been preloaded for this exercise.

Go for it!

 To use the DataFrame() function you need, first import the pandas package with the alias pd.

 Create a DataFrame from the list of dictionaries in list\_of\_dicts by calling pd.DataFrame(). Assign the resulting DataFrame to df.

 Inspect the contents of df printing the head of the DataFrame. Head of the DataFrame df can be accessed by calling df.head().

# 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())

## CountryName CountryCode ... Year Value

## 0 Arab World ARB ... 1960 133.56090740552298

## 1 Arab World ARB ... 1960 87.7976011532547

## 2 Arab World ARB ... 1960 6.634579191565161

## 3 Arab World ARB ... 1960 81.02332950839141

## 4 Arab World ARB ... 1960 3000000.0

##

## [5 rows x 6 columns]

Great work!

**Using Python generators for streaming data**

**Processing data in chunks (1)**

Sometimes, data sources can be so large in size that storing the entire dataset in memory becomes too resource-intensive. In this exercise, 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.

The csv file ‘world\_dev\_ind.csv’ is in your current directory for your use. 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.

If you’d like to learn more about context managers, refer to the [DataCamp course on Importing Data in Python.](https://www.datacamp.com/courses/importing-data-in-python-part-1)

 Use open() to bind the csv file ‘world\_dev\_ind.csv’ as file in the context manager.

 Complete the for loop so that it iterates **1000** times to perform the loop body and process only the first 1000 rows of data of the file.

# Open a connection to the file

with open('datasets/Python-Data-Science-Toolbox-Part-2/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(0, 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

## 'CountryName,CountryCode,Year,Total Population,Urban population (% of total)\n'

# Print the resulting dictionary

print(counts\_dict)

## {'Arab World': 5, 'Caribbean small states': 5, 'Central Europe and the Baltics': 5, 'East Asia & Pacific (all income levels)': 5, 'East Asia & Pacific (developing only)': 5, 'Euro area': 5, 'Europe & Central Asia (all income levels)': 5, 'Europe & Central Asia (developing only)': 5, 'European Union': 5, 'Fragile and conflict affected situations': 5, 'Heavily indebted poor countries (HIPC)': 5, 'High income': 5, 'High income: nonOECD': 5, 'High income: OECD': 5, 'Latin America & Caribbean (all income levels)': 5, 'Latin America & Caribbean (developing only)': 5, 'Least developed countries: UN classification': 5, 'Low & middle income': 5, 'Low income': 5, 'Lower middle income': 5, 'Middle East & North Africa (all income levels)': 5, 'Middle East & North Africa (developing only)': 5, 'Middle income': 5, 'North America': 5, 'OECD members': 5, 'Other small states': 5, 'Pacific island small states': 5, 'Small states': 5, 'South Asia': 5, 'Sub-Saharan Africa (all income levels)': 5, 'Sub-Saharan Africa (developing only)': 5, 'Upper middle income': 5, 'World': 4, 'Afghanistan': 4, 'Albania': 4, 'Algeria': 4, 'American Samoa': 4, 'Andorra': 4, 'Angola': 4, 'Antigua and Barbuda': 4, 'Argentina': 4, 'Armenia': 4, 'Aruba': 4, 'Australia': 4, 'Austria': 4, 'Azerbaijan': 4, '"Bahamas': 4, 'Bahrain': 4, 'Bangladesh': 4, 'Barbados': 4, 'Belarus': 4, 'Belgium': 4, 'Belize': 4, 'Benin': 4, 'Bermuda': 4, 'Bhutan': 4, 'Bolivia': 4, 'Bosnia and Herzegovina': 4, 'Botswana': 4, 'Brazil': 4, 'Brunei Darussalam': 4, 'Bulgaria': 4, 'Burkina Faso': 4, 'Burundi': 4, 'Cabo Verde': 4, 'Cambodia': 4, 'Cameroon': 4, 'Canada': 4, 'Cayman Islands': 4, 'Central African Republic': 4, 'Chad': 4, 'Channel Islands': 4, 'Chile': 4, 'China': 4, 'Colombia': 4, 'Comoros': 4, '"Congo': 8, 'Costa Rica': 4, "Cote d'Ivoire": 4, 'Croatia': 4, 'Cuba': 4, 'Curacao': 4, 'Cyprus': 4, 'Czech Republic': 4, 'Denmark': 4, 'Djibouti': 4, 'Dominica': 4, 'Dominican Republic': 4, 'Ecuador': 4, '"Egypt': 4, 'El Salvador': 4, 'Equatorial Guinea': 4, 'Eritrea': 4, 'Estonia': 4, 'Ethiopia': 4, 'Faeroe Islands': 4, 'Fiji': 4, 'Finland': 4, 'France': 4, 'French Polynesia': 4, 'Gabon': 4, '"Gambia': 4, 'Georgia': 4, 'Germany': 4, 'Ghana': 4, 'Greece': 4, 'Greenland': 4, 'Grenada': 4, 'Guam': 4, 'Guatemala': 4, 'Guinea': 4, 'Guinea-Bissau': 4, 'Guyana': 4, 'Haiti': 4, 'Honduras': 4, '"Hong Kong SAR': 4, 'Hungary': 4, 'Iceland': 4, 'India': 4, 'Indonesia': 4, '"Iran': 4, 'Iraq': 4, 'Ireland': 4, 'Isle of Man': 4, 'Israel': 4, 'Italy': 4, 'Jamaica': 4, 'Japan': 4, 'Jordan': 4, 'Kazakhstan': 4, 'Kenya': 4, 'Kiribati': 4, '"Korea': 8, 'Kuwait': 4, 'Kyrgyz Republic': 4, 'Lao PDR': 4, 'Latvia': 4, 'Lebanon': 4, 'Lesotho': 4, 'Liberia': 4, 'Libya': 4, 'Liechtenstein': 4, 'Lithuania': 4, 'Luxembourg': 4, '"Macao SAR': 4, '"Macedonia': 4, 'Madagascar': 4, 'Malawi': 4, 'Malaysia': 4, 'Maldives': 4, 'Mali': 4, 'Malta': 4, 'Marshall Islands': 4, 'Mauritania': 4, 'Mauritius': 4, 'Mexico': 4, '"Micronesia': 4, 'Moldova': 4, 'Monaco': 4, 'Mongolia': 4, 'Montenegro': 4, 'Morocco': 4, 'Mozambique': 4, 'Myanmar': 4, 'Namibia': 4, 'Nepal': 4, 'Netherlands': 4, 'New Caledonia': 4, 'New Zealand': 4, 'Nicaragua': 4, 'Niger': 4, 'Nigeria': 4, 'Northern Mariana Islands': 4, 'Norway': 4, 'Oman': 4, 'Pakistan': 4, 'Palau': 4, 'Panama': 4, 'Papua New Guinea': 4, 'Paraguay': 4, 'Peru': 4, 'Philippines': 4, 'Poland': 4, 'Portugal': 4, 'Puerto Rico': 4, 'Qatar': 4, 'Romania': 4, 'Russian Federation': 4, 'Rwanda': 4, 'Samoa': 4, 'San Marino': 4, 'Sao Tome and Principe': 4, 'Saudi Arabia': 4, 'Senegal': 4, 'Seychelles': 4, 'Sierra Leone': 4, 'Singapore': 4, 'Slovak Republic': 4, 'Slovenia': 4, 'Solomon Islands': 4, 'Somalia': 4, 'South Africa': 4, 'South Sudan': 4, 'Spain': 4, 'Sri Lanka': 4, 'St. Kitts and Nevis': 4, 'St. Lucia': 4, 'St. Vincent and the Grenadines': 4, 'Sudan': 4, 'Suriname': 4, 'Swaziland': 4, 'Sweden': 4, 'Switzerland': 4, 'Syrian Arab Republic': 4, 'Tajikistan': 4, 'Tanzania': 4, 'Thailand': 4, 'Timor-Leste': 4, 'Togo': 4, 'Tonga': 4, 'Trinidad and Tobago': 4, 'Tunisia': 4, 'Turkey': 4, 'Turkmenistan': 4, 'Turks and Caicos Islands': 4, 'Tuvalu': 4, 'Uganda': 4, 'Ukraine': 4, 'United Arab Emirates': 4, 'United Kingdom': 4, 'United States': 4, 'Uruguay': 4, 'Uzbekistan': 4, 'Vanuatu': 4, '"Venezuela': 4, 'Vietnam': 4, 'Virgin Islands (U.S.)': 4, '"Yemen': 4, 'Zambia': 4, 'Zimbabwe': 4}

Great work!

**Writing a generator to load data in chunks (2)**

In the previous exercise, 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*](https://www.blog.pythonlibrary.org/2014/01/27/python-201-an-intro-to-generators/) 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.

In this exercise, 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! So out in the wild, you won’t have to explicitly create generator objects in cases such as this. However, for pedagogical reasons, we are having you practice how to do this here with the read\_large\_file() function. Go for it!

 In the function read\_large\_file(), read a line from file\_object by using the method readline(). Assign the result to data.

 In the function read\_large\_file(), yield the line read from the file data.

 In the context manager, create a generator object gen\_file by calling your generator function read\_large\_file() and passing file to it.

 Print the first three lines produced by the generator object gen\_file using next().

# 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('datasets/Python-Data-Science-Toolbox-Part-2/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))

## CountryName,CountryCode,Year,Total Population,Urban population (% of total)

##

## Arab World,ARB,1960,92495902.0,31.285384211605397

##

## Caribbean small states,CSS,1960,4190810.0,31.5974898513652

Wonderful work! Note that since a file object is already a generator, you don’t have to explicitly create a generator object with your read\_large\_file() function. However, it is still good to practice how to create generators - well done!

**Writing a generator to load data in chunks (3)**

Great! You’ve just created a generator function that you can use to help you process large files.

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!

The generator function read\_large\_file() and the csv file ‘world\_dev\_ind.csv’ are preloaded and ready for your use. Go for it!

 Bind the file ‘world\_dev\_ind.csv’ to file in the context manager with open().

 Complete the for loop so that it iterates over the generator from the call to read\_large\_file() to process all the rows of the file.

# Initialize an empty dictionary: counts\_dict

counts\_dict = {}

# Open a connection to the file

with open('datasets/Python-Data-Science-Toolbox-Part-2/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)

## {'CountryName': 1, 'Arab World': 55, 'Caribbean small states': 55, 'Central Europe and the Baltics': 55, 'East Asia & Pacific (all income levels)': 55, 'East Asia & Pacific (developing only)': 55, 'Euro area': 55, 'Europe & Central Asia (all income levels)': 55, 'Europe & Central Asia (developing only)': 55, 'European Union': 55, 'Fragile and conflict affected situations': 55, 'Heavily indebted poor countries (HIPC)': 55, 'High income': 55, 'High income: nonOECD': 55, 'High income: OECD': 55, 'Latin America & Caribbean (all income levels)': 55, 'Latin America & Caribbean (developing only)': 55, 'Least developed countries: UN classification': 55, 'Low & middle income': 55, 'Low income': 55, 'Lower middle income': 55, 'Middle East & North Africa (all income levels)': 55, 'Middle East & North Africa (developing only)': 55, 'Middle income': 55, 'North America': 55, 'OECD members': 55, 'Other small states': 55, 'Pacific island small states': 55, 'Small states': 55, 'South Asia': 55, 'Sub-Saharan Africa (all income levels)': 55, 'Sub-Saharan Africa (developing only)': 55, 'Upper middle income': 55, 'World': 55, 'Afghanistan': 55, 'Albania': 55, 'Algeria': 55, 'American Samoa': 55, 'Andorra': 55, 'Angola': 55, 'Antigua and Barbuda': 55, 'Argentina': 55, 'Armenia': 55, 'Aruba': 55, 'Australia': 55, 'Austria': 55, 'Azerbaijan': 55, '"Bahamas': 55, 'Bahrain': 55, 'Bangladesh': 55, 'Barbados': 55, 'Belarus': 55, 'Belgium': 55, 'Belize': 55, 'Benin': 55, 'Bermuda': 55, 'Bhutan': 55, 'Bolivia': 55, 'Bosnia and Herzegovina': 55, 'Botswana': 55, 'Brazil': 55, 'Brunei Darussalam': 55, 'Bulgaria': 55, 'Burkina Faso': 55, 'Burundi': 55, 'Cabo Verde': 55, 'Cambodia': 55, 'Cameroon': 55, 'Canada': 55, 'Cayman Islands': 55, 'Central African Republic': 55, 'Chad': 55, 'Channel Islands': 55, 'Chile': 55, 'China': 55, 'Colombia': 55, 'Comoros': 55, '"Congo': 110, 'Costa Rica': 55, "Cote d'Ivoire": 55, 'Croatia': 55, 'Cuba': 55, 'Curacao': 55, 'Cyprus': 55, 'Czech Republic': 55, 'Denmark': 55, 'Djibouti': 55, 'Dominica': 55, 'Dominican Republic': 55, 'Ecuador': 55, '"Egypt': 55, 'El Salvador': 55, 'Equatorial Guinea': 55, 'Eritrea': 55, 'Estonia': 55, 'Ethiopia': 55, 'Faeroe Islands': 55, 'Fiji': 55, 'Finland': 55, 'France': 55, 'French Polynesia': 55, 'Gabon': 55, '"Gambia': 55, 'Georgia': 55, 'Germany': 55, 'Ghana': 55, 'Greece': 55, 'Greenland': 55, 'Grenada': 55, 'Guam': 55, 'Guatemala': 55, 'Guinea': 55, 'Guinea-Bissau': 55, 'Guyana': 55, 'Haiti': 55, 'Honduras': 55, '"Hong Kong SAR': 55, 'Hungary': 55, 'Iceland': 55, 'India': 55, 'Indonesia': 55, '"Iran': 55, 'Iraq': 55, 'Ireland': 55, 'Isle of Man': 55, 'Israel': 55, 'Italy': 55, 'Jamaica': 55, 'Japan': 55, 'Jordan': 55, 'Kazakhstan': 55, 'Kenya': 55, 'Kiribati': 55, '"Korea': 110, 'Kuwait': 52, 'Kyrgyz Republic': 55, 'Lao PDR': 55, 'Latvia': 55, 'Lebanon': 55, 'Lesotho': 55, 'Liberia': 55, 'Libya': 55, 'Liechtenstein': 55, 'Lithuania': 55, 'Luxembourg': 55, '"Macao SAR': 55, '"Macedonia': 55, 'Madagascar': 55, 'Malawi': 55, 'Malaysia': 55, 'Maldives': 55, 'Mali': 55, 'Malta': 55, 'Marshall Islands': 55, 'Mauritania': 55, 'Mauritius': 55, 'Mexico': 55, '"Micronesia': 55, 'Moldova': 55, 'Monaco': 55, 'Mongolia': 55, 'Montenegro': 55, 'Morocco': 55, 'Mozambique': 55, 'Myanmar': 55, 'Namibia': 55, 'Nepal': 55, 'Netherlands': 55, 'New Caledonia': 55, 'New Zealand': 55, 'Nicaragua': 55, 'Niger': 55, 'Nigeria': 55, 'Northern Mariana Islands': 55, 'Norway': 55, 'Oman': 55, 'Pakistan': 55, 'Palau': 55, 'Panama': 55, 'Papua New Guinea': 55, 'Paraguay': 55, 'Peru': 55, 'Philippines': 55, 'Poland': 55, 'Portugal': 55, 'Puerto Rico': 55, 'Qatar': 55, 'Romania': 55, 'Russian Federation': 55, 'Rwanda': 55, 'Samoa': 55, 'San Marino': 55, 'Sao Tome and Principe': 55, 'Saudi Arabia': 55, 'Senegal': 55, 'Seychelles': 55, 'Sierra Leone': 55, 'Singapore': 55, 'Slovak Republic': 55, 'Slovenia': 55, 'Solomon Islands': 55, 'Somalia': 55, 'South Africa': 55, 'South Sudan': 55, 'Spain': 55, 'Sri Lanka': 55, 'St. Kitts and Nevis': 55, 'St. Lucia': 55, 'St. Vincent and the Grenadines': 55, 'Sudan': 55, 'Suriname': 55, 'Swaziland': 55, 'Sweden': 55, 'Switzerland': 55, 'Syrian Arab Republic': 55, 'Tajikistan': 55, 'Tanzania': 55, 'Thailand': 55, 'Timor-Leste': 55, 'Togo': 55, 'Tonga': 55, 'Trinidad and Tobago': 55, 'Tunisia': 55, 'Turkey': 55, 'Turkmenistan': 55, 'Turks and Caicos Islands': 55, 'Tuvalu': 55, 'Uganda': 55, 'Ukraine': 55, 'United Arab Emirates': 55, 'United Kingdom': 55, 'United States': 55, 'Uruguay': 55, 'Uzbekistan': 55, 'Vanuatu': 55, '"Venezuela': 55, 'Vietnam': 55, 'Virgin Islands (U.S.)': 55, '"Yemen': 55, 'Zambia': 55, 'Zimbabwe': 55, 'Serbia': 25, 'West Bank and Gaza': 25, 'Sint Maarten (Dutch part)': 17}

Great work!

**Using pandas’ read\_csv iterator for streaming data**

**Writing an iterator to load data in chunks (1)**

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). This creates an iterable **reader object**, which means that you can use next() on it.

In this exercise, you will read a file in small DataFrame chunks with read\_csv(). You’re going to use the World Bank Indicators data ‘ind\_pop.csv’, available in your current directory, to look at the urban population indicator for numerous countries and years.

 Use pd.read\_csv() to read in ‘ind\_pop.csv’ in chunks of size 10. Assign the result to df\_reader.

 Print the first two chunks from df\_reader.

# Import the pandas package

import pandas as pd

# Initialize reader object: df\_reader

df\_reader = pd.read\_csv('datasets/Python-Data-Science-Toolbox-Part-2/world\_ind\_pop\_data.csv', chunksize=10)

# Print two chunks

print(next(df\_reader))

## CountryName ... Urban population (% of total)

## 0 Arab World ... 31.285384

## 1 Caribbean small states ... 31.597490

## 2 Central Europe and the Baltics ... 44.507921

## 3 East Asia & Pacific (all income levels) ... 22.471132

## 4 East Asia & Pacific (developing only) ... 16.917679

## 5 Euro area ... 62.096947

## 6 Europe & Central Asia (all income levels) ... 55.378977

## 7 Europe & Central Asia (developing only) ... 38.066129

## 8 European Union ... 61.212898

## 9 Fragile and conflict affected situations ... 17.891972

##

## [10 rows x 5 columns]

print(next(df\_reader))

## CountryName ... Urban population (% of total)

## 10 Heavily indebted poor countries (HIPC) ... 12.236046

## 11 High income ... 62.680332

## 12 High income: nonOECD ... 56.107863

## 13 High income: OECD ... 64.285435

## 14 Latin America & Caribbean (all income levels) ... 49.284688

## 15 Latin America & Caribbean (developing only) ... 44.863308

## 16 Least developed countries: UN classification ... 9.616261

## 17 Low & middle income ... 21.272894

## 18 Low income ... 11.498396

## 19 Lower middle income ... 19.810513

##

## [10 rows x 5 columns]

Great work!

**Writing an iterator to load data in chunks (2)**

In the previous exercise, you used read\_csv() to read in DataFrame chunks from a large dataset. In this exercise, you will read in a file using a bigger DataFrame chunk size and then process the data from the first chunk.

To process the data, you will create another DataFrame composed of only the rows from a specific country. You will then zip together two of the columns from the new DataFrame, ‘Total Population’ and ‘Urban population (% of total)’. Finally, you will create a list of tuples from the zip object, where each tuple is composed of a value from each of the two columns mentioned.

You’re going to use the data from ‘ind\_pop\_data.csv’, available in your current directory. pandas has been imported as pd.

 Use pd.read\_csv() to read in the file in ‘ind\_pop\_data.csv’ in chunks of size 1000. Assign the result to urb\_pop\_reader.

 Get the **first** DataFrame chunk from the iterable urb\_pop\_reader and assign this to df\_urb\_pop.

# Initialize reader object: urb\_pop\_reader

urb\_pop\_reader = pd.read\_csv('datasets/Python-Data-Science-Toolbox-Part-2/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())

## CountryName ... Urban population (% of total)

## 0 Arab World ... 31.285384

## 1 Caribbean small states ... 31.597490

## 2 Central Europe and the Baltics ... 44.507921

## 3 East Asia & Pacific (all income levels) ... 22.471132

## 4 East Asia & Pacific (developing only) ... 16.917679

##

## [5 rows x 5 columns]

 Select only the rows of df\_urb\_pop that have a ‘CountryCode’ of ‘CEB’. To do this, compare whether df\_urb\_pop[‘CountryCode’] is **equal** to ‘CEB’ within the square brackets in df\_urb\_pop[\_\_\_\_].

 Using zip(), zip together the ‘Total Population’ and ‘Urban population (% of total)’ columns of df\_pop\_ceb. Assign the resulting zip object to pops.

# 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)

## [(91401583.0, 44.5079211390026), (92237118.0, 45.206665319194), (93014890.0, 45.866564696018), (93845749.0, 46.5340927663649), (94722599.0, 47.2087429803526)]

Great work! Time to step it up a notch!

**Writing an iterator to load data in chunks (3)**

You’re getting used to reading and processing data in chunks by now. Let’s push your skills a little further by adding a column to a DataFrame.

Starting from the code of the previous exercise, you will be using a *list comprehension* to create the values for a new column ‘Total Urban Population’ from the list of tuples that you generated earlier. Recall from the previous exercise that the first and second elements of each tuple consist of, respectively, values from the columns ‘Total Population’ and ‘Urban population (% of total)’. The values in this new column ‘Total Urban Population’, therefore, are the product of the first and second element in each tuple. Furthermore, because the 2nd element is a percentage, you need to divide the entire result by 100, or alternatively, multiply it by 0.01.

You will also plot the data from this new column to create a visualization of the urban population data.

The packages pandas and matplotlib.pyplot have been imported as pd and plt respectively for your use.

 Write a list comprehension to generate a list of values from pops\_list for the new column ‘Total Urban Population’. The *output expression* should be the product of the first and second element in each tuple in pops\_list. Because the 2nd element is a percentage, you also need to either multiply the result by 0.01 or divide it by 100. In addition, note that the column ‘Total Urban Population’ should only be able to take on integer values. To ensure this, make sure you cast the *output expression* to an integer with int().

 Create a *scatter* plot where the x-axis are values from the ‘Year’ column and the y-axis are values from the ‘Total Urban Population’ column.

# Code from previous exercise

urb\_pop\_reader = pd.read\_csv('datasets/Python-Data-Science-Toolbox-Part-2/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(tup[0] \* tup[1] \* 0.01) for tup in pops\_list]

## <string>:1: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

import matplotlib.pyplot as plt

# Plot urban population data

df\_pop\_ceb.plot(kind='scatter', x='Year', y='Total Urban Population')

plt.show()

Chart, scatter chart

Description automatically generated

Great work!

**Writing an iterator to load data in chunks (4)**

In the previous exercises, you’ve only processed the data from the first DataFrame chunk. This time, you will aggregate the results over all the DataFrame chunks in the dataset. This basically means you will be processing the **entire** dataset now. This is neat because you’re going to be able to process the entire large dataset by just working on smaller pieces of it!

You’re going to use the data from ‘ind\_pop\_data.csv’, available in your current directory. The packages pandas and matplotlib.pyplot have been imported as pd and plt respectively for your use.

 Initialize an empty DataFrame data using pd.DataFrame().

 In the for loop, iterate over urb\_pop\_reader to be able to process all the DataFrame chunks in the dataset.

 Using the method append() of the DataFrame data, append df\_pop\_ceb to data.

# Initialize reader object: urb\_pop\_reader

urb\_pop\_reader = pd.read\_csv('datasets/Python-Data-Science-Toolbox-Part-2/world\_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\_list]

# Append DataFrame chunk to data: data

data = data.append(df\_pop\_ceb)

## <string>:14: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:17: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:14: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:17: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:14: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:17: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:14: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:17: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:14: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:17: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:14: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:17: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:14: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:17: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:14: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:17: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:14: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:17: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:14: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:17: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:14: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:17: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:14: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:17: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:14: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:17: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:14: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:17: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

# Plot urban population data

data.plot(kind='scatter', x='Year', y='Total Urban Population')

plt.show()

Chart, line chart

Description automatically generated

Great work!

**Writing an iterator to load data in chunks (5)**

This is the last leg. You’ve learned a lot about processing a large dataset in chunks. In this last exercise, you will put all the code for processing the data into a single function so that you can reuse the code without having to rewrite the same things all over again.

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!

You’re going to use the data from ‘ind\_pop\_data.csv’, available in your current directory. The packages pandas and matplotlib.pyplot has been imported as pd and plt respectively for your use.

After you are done, take a moment to look at the plots and reflect on the new skills you have acquired. The journey doesn’t end here! If you have enjoyed working with this data, you can continue exploring it using the pre-processed version available on Kaggle.

 Define the function plot\_pop() that has two arguments: first is filename for the file to process and second is country\_code for the country to be processed in the dataset.

# 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 Population'

df\_pop\_ceb['Total Urban Population'] = [int(tup[0] \* tup[1] \* 0.01) for tup in pops\_list]

# 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()

 Call plot\_pop() to process the data for country code ‘CEB’ in the file ‘ind\_pop\_data.csv’.

 Call plot\_pop() to process the data for country code ‘ARB’ in the file ‘ind\_pop\_data.csv’.

# Set the filename: fn

fn = 'datasets/Python-Data-Science-Toolbox-Part-2/world\_ind\_pop\_data.csv'

# Call plot\_pop for country code 'CEB'

plot\_pop(fn, 'CEB')

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

Chart, line chart

Description automatically generated

# Call plot\_pop for country code 'ARB'

plot\_pop(fn, 'ARB')

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

## <string>:22: SettingWithCopyWarning:

## A value is trying to be set on a copy of a slice from a DataFrame.

## Try using .loc[row\_indexer,col\_indexer] = value instead

##

## See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy

## <string>:25: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future version. Use pandas.concat instead.

Chart, line chart

Description automatically generated

Great work!

**Final thoughts**

**Final thoughts**

Wow, you sure have come a long way! Congratulations on making it through the case study, in which you had to apply skills from both of these Data Science toolbox courses.

**You’ve applied your skills in:**

You have just used your new skills of writing user-defined functions, iterators, list comprehensions and generators to explore and extract results from the World Bank World Development Indicators dataset. These are all tools that you’ll utilize time and time again in your work as a Data Scientist and you have worked hard to earn your Pythonista stripes.

**Congratulations!**

Good work and keep on coding in