e

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

* 4 hours
* 12 Videos
* 46 Exercises
* 232,585 Participants
* 3,800 XP

**Course Description**



In this 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.

1. 1

**Using iterators in PythonLand**

0%

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**

50 xp

**Iterators vs. Iterables**

50 xp

**Iterating over iterables (1)**

100 xp

**Iterating over iterables (2)**

100 xp

**Iterators as function arguments**

100 xp

**Playing with iterators**

50 xp

**Using enumerate**

100 xp

**Using zip**

100 xp

**Using \* and zip to 'unzip'**

100 xp

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

50 xp

**Processing large amounts of Twitter data**

100 xp

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

100 xp

**Congratulations!**

50 xp

[Hide Chapter Details](https://www.datacamp.com/courses/python-data-science-toolbox-part-2?embedded=true)

  2

**List comprehensions and generators**

0%

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**

50 xp

**Write a basic list comprehension**

50 xp

**List comprehension over iterables**

50 xp

**Writing list comprehensions**

100 xp

**Nested list comprehensions**

100 xp

**Advanced comprehensions**

50 xp

**Using conditionals in comprehensions (1)**

100 xp

**Using conditionals in comprehensions (2)**

100 xp

**Dict comprehensions**

100 xp

**Introduction to generator expressions**

50 xp

**List comprehensions vs. generators**

50 xp

**Write your own generator expressions**

100 xp

**Changing the output in generator expressions**

100 xp

**Build a generator**

100 xp

**Wrapping up comprehensions and generators.**

50 xp

**List comprehensions for time-stamped data**

100 xp

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

100 xp

[Hide Chapter Details](https://www.datacamp.com/courses/python-data-science-toolbox-part-2?embedded=true)

  3

**Bringing it all together!**

0%

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!**

50 xp

**Dictionaries for data science**

100 xp

**Writing a function to help you**

100 xp

**Using a list comprehension**

100 xp

**Turning this all into a DataFrame**

100 xp

**Using Python generators for streaming data**

50 xp

**Processing data in chunks (1)**

100 xp

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

100 xp

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

100 xp

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

50 xp

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

100 xp

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

100 xp

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

100 xp

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

100 xp

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

100 xp

**Final thoughts**

**Daily XP150**

# Introduction to iterators

**50 XP**

This course is also available on the mobile app

## 1. Introduction to iterators

Welcome to the course! My name is Hugo Bowne-Anderson and I am a Data Scientist at DataCamp. In this course, the second of the Python Data Science toolbox courses, you'll learn about iterables, iterators, list comprehensions and generators - all essential components in the Pythonista's Data Science toolbox. We will conclude with an entire chapter devoted to a case study in which you'll apply time and time again techniques learned in both of these courses. Let's talk about iterators.

## 2. Iterating with a for loop

There's no reason to be scared of iterators because you have actually been working with them for some time now! When you use a for loop to print out each element of a list, you're iterating over the list.

## 3. Iterating with a for loop

You can also use a for loop to iterate over characters in a string such as you see here. You can also use a for loop to iterate a over

## 4. Iterating with a for loop

a sequence of numbers produced by a special range object. The reason that we can loop over such objects is that they are special objects

## 5. Iterators vs. iterables

called iterables: lists, strings and range objects are all iterables, as are many other Python objects, such as dictionaries and file connections! The actual definition of an iterable is an object that has an associated iter method. Once this iter method is applied to an iterable, an iterator object is created. Under the hood, this is actually what a for loop is doing: it takes an iterable, creates the associated iterator object, and iterates over it! An iterator is defined as an object that has an associated next method that produces the consecutive values. To create an iterator from an iterable,

## 6. Iterating over iterables: next()

all we need to do is use the function iter and pass it the iterable. Once we have the iterator defined, we pass it to the function next and this returns the first value. Calling next again on the iterator returns the next value until there are no values left to return and then it throws us a StopIteration error.

## 7. Iterating at once with \*

You can also print all values of an iterator in one fell swoop using the star operator, referred to as the splat operator in some circles. This star operator unpacks all elements of an iterator or an iterable. Be warned, however, once you do so, you cannot do it again as there are no more values to iterate through! We would have to redefine our iterator to do so.

## 8. Iterating over dictionaries

We mentioned before that dictionaries and file connections are iterables as well. To iterate over the key-value pairs of a Python dictionary, we need to unpack them by applying the items method to the dictionary as you can see here.

## 9. Iterating over file connections

With respect to file connections, here you can see how to use the iter and next methods to return the lines from a file, file dot txt. This has been your crash course in the fundamentals of iterables and iterators.

## 10. Let's practice!

Use your gained knowledge wisely and have fun iterating! -

**Daily XP200**

**Exercise**

**Exercise**

**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*.

**Instructions**

**50 XP**

**Possible Answers**

* 

Both flash1 and flash2 are iterators.

* 

Both flash1 and flash2 are iterables.

* 

**flash1 is an iterable and flash2 is an iterator.**

In [1]:

print(flash1)

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

In [2]:

print(next(flash1))

Traceback (most recent call last):

File "<stdin>", line 72, in exceptionCatcher

raise exception

File "<stdin>", line 3361, in run\_ast\_nodes

if (await self.run\_code(code, result, async\_=asy)):

File "<stdin>", line 3458, in run\_code

self.showtraceback(running\_compiled\_code=True)

File "<stdin>", line 2066, in showtraceback

self.\_showtraceback(etype, value, stb)

File "<stdin>", line 72, in exceptionCatcher

raise exception

File "<stdin>", line 3441, in run\_code

exec(code\_obj, self.user\_global\_ns, self.user\_ns)

File "<stdin>", line 1, in <module>

print(next(flash1))

TypeError: 'list' object is not an iterator

In [3]:

print(flash2)

<list\_iterator object at 0x7fd7aa05b2e0>

In [4]:

print(next(flash2))

jay garrick

**Daily XP250**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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
* # Create an iterator for flash: superhero
* # Print each item from the iterator
* print(\_\_\_\_)
* print(\_\_\_\_)
* print(\_\_\_\_)
* print(\_\_\_\_)

# Create a list of strings: flash

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

# Print each list item in flash using a for loop

for person in flash:

    print(person)

# Create an iterator for flash: superhero

superhero = iter(flash)

# Print each item from the iterator

print(next(superhero))

print(next(superhero))

print(next(superhero))

print(next(superhero))

# Create a list of strings: flash

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

# Print each list item in flash using a for loop

for person in flash:

print(person)

# Create an iterator for flash: superhero

superhero = iter(flash)

# Print each item from the iterator

print(next(superhero))

print(next(superhero))

print(next(superhero))

print(next(superhero))

jay garrick

barry allen

wally west

bart allen

jay garrick

barry allen

wally west

bart allen

**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

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

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

won't actually pre-create the list.

**Instructions**

**100 XP**

* 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 = \_\_\_\_

# Print the values in small\_value

print(next(small\_value))

print(next(small\_value))

print(next(small\_value))

# Loop over range(3) and print the values

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

googol = \_\_\_\_

# Print the first 5 values from googol

print(next(googol))

print(next(googol))

print(next(googol))

print(next(googol))

print(next(googol))

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

small\_value = iter(range(3))

#print(\*small\_value)

# Print the values in small\_value

print(next(small\_value))

print(next(small\_value))

print(next(small\_value))

# Loop over range(3) and print the values

for num in range(3):

    print(num)

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

googol = iter(range(10\*\*100))

# Print the first 5 values from googol

#print(next(googol))

#rint(next(googol))

#print(next(googol))

#print(next(googol))

#print(next(googol))

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

small\_value = iter(range(3))

#print(\*small\_value)

# Print the values in small\_value

print(next(small\_value))

print(next(small\_value))

print(next(small\_value))

# Loop over range(3) and print the values

for num in range(3):

print(num)

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

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

# Print the first 5 values from googol

#print(next(googol))

#rint(next(googol))

#print(next(googol))

#print(next(googol))

#print(next(googol))

0

1

2

0

1

2

**Daily XP450**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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 = \_\_\_\_

# Print the range object

print(values)

# Create a list of integers: values\_list

values\_list = \_\_\_\_

# Print values\_list

print(values\_list)

# Get the sum of values: values\_sum

values\_sum = \_\_\_\_

# Print values\_sum

print(values\_sum)

# Create a range object: values

values = range(10,21)

# Print the range object

print(values)

# Create a list of integers: values\_list

values\_list = list(values)

# Print values\_list

print(values\_list)

# Get the sum of values: values\_sum

values\_sum = sum(values)

# Print values\_sum

print(values\_sum)

# Create a range object: values

values = range(10,21)

# Print the range object

print(values)

# Create a list of integers: values\_list

values\_list = list(values)

# Print values\_list

print(values\_list)

# Get the sum of values: values\_sum

values\_sum = sum(values)

# Print values\_sum

print(values\_sum)

range(10, 21)

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

165

# Playing with iterators

**50 XP**

This course is also available on the mobile app

## 1. Playing with iterators

We're now going to dive a bit deeper into the world of iterables and iterators by checking out some very cool, very useful functions. The first function, enumerate, will allow us to add a counter to any iterable while the second function, zip, will allow us to stitch together an arbitrary number of iterables. Let's begin:

## 2. Using enumerate()

enumerate is a function that takes any iterable as argument, such as a list, and returns a special enumerate object, which consists of pairs containing the elements of the original iterable, along with their index within the iterable. We can use the function list to turn this enumerate object into a list of tuples and print it to see what it contains. The enumerate object itself

## 3. enumerate() and unpack

is also an iterable and we can loop over it while unpacking its elements using the clause for index, value in enumerate(avengers). It is the default behavior of enumerate to begin indexing at 0. However, you can alter this with a second argument, start, which you can see here.

## 4. Using zip()

Now let's move on to zip, which accepts an arbitrary number of iterables and returns an iterator of tuples. Here we have two lists, one of the avengers, the other of their names. Zipping them together creates a zip object which is an iterator of tuples. We can turn this zip object into a list and print the list. The first element is a tuple containing the first elements of each list that was zipped. The second element is a tuple containing the second elements of each list that was zipped and so on.

## 5. zip() and unpack

Alternatively, we could use a for loop to iterate over the zip object and print the tuples. We could also

## 6. Print zip with \*

have used the splat operator to print all the elements! Now it's your turn to play with these useful functions.

## 7. Let's practice!

Enjoy! -

**Daily XP600**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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 = \_\_\_\_

# Print the list of tuples

print(mutant\_list)

# Unpack and print the tuple pairs

for \_\_\_\_ in \_\_\_\_:

    print(index1, value1)

# Change the start index

for \_\_\_\_ in \_\_\_\_:

    print(index2, value2)

# Create a list of strings: mutants

mutants = ['charles xavier',

            'bobby drake',

            'kurt wagner',

            'max eisenhardt',

            'kitty pryde']

# Create a list of tuples: mutant\_list

mutant\_list = (list(enumerate(mutants)))

# Print the list of tuples

print(mutant\_list)

# Unpack and print the tuple pairs

for index1, value1 in enumerate(mutants):

    print(index1, value1)

# Change the start index

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

    print(index2, value2)

# Create a list of strings: mutants

mutants = ['charles xavier',

'bobby drake',

'kurt wagner',

'max eisenhardt',

'kitty pryde']

# Create a list of tuples: mutant\_list

mutant\_list = (list(enumerate(mutants)))

# Print the list of tuples

print(mutant\_list)

# Unpack and print the tuple pairs

for index1, value1 in enumerate(mutants):

print(index1, value1)

# Change the start index

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

print(index2, value2)

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

0 charles xavier

1 bobby drake

2 kurt wagner

3 max eisenhardt

4 kitty pryde

1 charles xavier

2 bobby drake

3 kurt wagner

4 max eisenhardt

5 kitty pryde

**Daily XP700**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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.
* # Create a list of tuples: mutant\_data
* mutant\_data = \_\_\_\_
* # Print the list of tuples
* print(mutant\_data)
* # Create a zip object using the three lists: mutant\_zip
* mutant\_zip = \_\_\_\_
* # Print the zip object
* print(mutant\_zip)
* # Unpack the zip object and print the tuple values
* for \_\_\_\_ in \_\_\_\_:
* print(value1, value2, value3)

# Create a list of tuples: mutant\_data

mutant\_data = list(zip(mutants, aliases, powers))

# Print the list of tuples

print(mutant\_data)

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

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

# Print the zip object

print(mutant\_zip)

# Unpack the zip object and print the tuple values

for value1, value2, value3 in zip(mutants, aliases, powers):

    print(value1, value2, value3)

or this:

# Create a list of tuples: mutant\_data

mutant\_data = list(zip(mutants, aliases, powers))

# Print the list of tuples

print(mutant\_data)

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

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

# Print the zip object

print(mutant\_zip)

# Unpack the zip object and print the tuple values

for value1, value2, value3 in mutant\_zip:

    print(value1, value2, value3)

# Create a list of tuples: mutant\_data

mutant\_data = list(zip(mutants, aliases, powers))

# Print the list of tuples

print(mutant\_data)

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

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

# Print the zip object

print(mutant\_zip)

# Unpack the zip object and print the tuple values

for value1, value2, value3 in zip(mutants, aliases, powers):

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')]

<zip object at 0x7f8ce1118fc0>

charles xavier prof x telepathy

bobby drake iceman thermokinesis

kurt wagner nightcrawler teleportation

max eisenhardt magneto magnetokinesis

kitty pryde shadowcat intangibility

**Daily XP800**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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 = \_\_\_\_

# Print the tuples in z1 by unpacking with \*

print(\_\_\_\_)

# Re-create a zip object from mutants and powers: z1

z1 = \_\_\_\_

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

result1, result2 = \_\_\_\_

# Check if unpacked tuples are equivalent to original tuples

print(result1 == mutants)

print(result2 == powers)

# Create a zip object from mutants and powers: z1

z1 = zip(mutants, powers)

# Print the tuples in z1 by unpacking with \*

print(\*z1)

# Re-create a zip object from mutants and powers: z1

z1 = zip(mutants, powers)

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

result1, result2 = zip(\*z1)

# Check if unpacked tuples are equivalent to original tuples

print(result1 == mutants)

print(result2 == powers)

# Create a zip object from mutants and powers: z1

z1 = zip(mutants, powers)

# Print the tuples in z1 by unpacking with \*

print(\*z1)

# Re-create a zip object from mutants and powers: z1

z1 = zip(mutants, powers)

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

result1, result2 = zip(\*z1)

# Check if unpacked tuples are equivalent to original tuples

print(result1 == mutants)

print(result2 == powers)

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

True

True

**Daily XP900**

# Using iterators to load large files into memory

**50 XP**

This course is also available on the mobile app

## 1. Using iterators to load large files into memory

Now that you're more comfortable with iterables, iterators and how they work, we're going to check out a particular use case that is pertinent to the world of Data Science:

## 2. Loading data in chunks

dealing with large amounts of data. Let's say that you are pulling data from a file, database or API and there's so much of it, just so much data, that you can't hold it in memory. One solution is to load the data in chunks, perform the desired operation or operations on each chuck, store the result, discard the chunk and then load the next chunk; this sounds like a place where an iterator could be useful!To surmount this challenge, we are going to use the pandas function read\_csv, which provides a wonderful option whereby you can load data in chunks and iterate over them. All we need to do is to specify the chunk using the argument yep, you guessed it: chunksize. As with much of what we do in Data Science, this is best illustrated by an example.

## 3. Iterating over data

Let's say that we have a csv with a column called 'x' of numbers and I want to compute the sum of all the numbers in that column. However, the file is too large to store in memory. We first import pandas and then initialize an empty list result to hold the result of each iteration. We then use the read\_csv function, utilizing the argument chunksize, setting it to the size of the chunks I want to read in. In this example, we use a chunk size of 1,000. You can play around with it. The object created by the read\_csv call is an iterable so I can can iterate over it, using a for loop, in which each chunk will be a DataFrame. Within the for loop, that is, on each iteration, we compute the sum of the column of interest and we append it to the list result. Once this is executed, we can take the sum of the list result and this gives us our total sum of the column of interest. Iterators to the rescue!

## 4. Iterating over data

Also note that we need not have used a list to store each result - we could have initialized total to zero before iterating over the file and added each sum during the iteration procedure, as you see here. Now things get really cool: you're going to use an iterator to load Twitter data in chunks and perform a similar computation that you did in the prequel to this course.

## 5. Let's practice!

Then you're going to write a function that does the same. Happy iterating, friend! -

**Daily XP950**

**Exercise**

**Exercise**

**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).*

**Instructions**

**100 XP**

* 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.
* # Initialize an empty dictionary: counts\_dict
* # Iterate over the file chunk by chunk
* for \_\_\_\_ in \_\_\_\_:
* # Iterate over the column in DataFrame
* for \_\_\_\_ in \_\_\_\_:
* if entry in counts\_dict.keys():
* counts\_dict[entry] += 1
* else:
* counts\_dict[entry] = 1
* # Print the populated dictionary
* print(counts\_dict)

# Initialize an empty dictionary: counts\_dict

counts\_dict = {}

# Iterate over the file chunk by chunk

for chunk in pd.read\_csv('tweets.csv', chunksize=10):

    # Iterate over the column in DataFrame

    for entry in chunk['lang']:

        if entry in counts\_dict.keys():

            counts\_dict[entry] += 1

        else:

            counts\_dict[entry] = 1

# Print the populated dictionary

print(counts\_dict)

# Initialize an empty dictionary: counts\_dict

counts\_dict = {}

# Iterate over the file chunk by chunk

for chunk in pd.read\_csv('tweets.csv', chunksize=10):

# Iterate over the column in DataFrame

for entry in chunk['lang']:

if entry in counts\_dict.keys():

counts\_dict[entry] += 1

else:

counts\_dict[entry] = 1

# Print the populated dictionary

print(counts\_dict)

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

**Daily XP1050**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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 \_\_\_\_():
* """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 \_\_\_\_ in \_\_\_\_:
* # Iterate over the column in DataFrame
* for \_\_\_\_ in \_\_\_\_:
* 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 = \_\_\_\_
* # Print result\_counts
* print(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('tweets.csv', 10, 'lang')

# Print result\_counts

print(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('tweets.csv', 10, 'lang')

# Print result\_counts

print(result\_counts)

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

**Daily XP100**

# Congratulations!

**50 XP**

## 1. Congratulations!

Congratulations, you've now grappled with the nuts and bolts of iterators in Python Land and survived. You've even applied your newfound skills to an example from the world of big data: loading and working on a file chunk by chunk!

## 2. What’s next?

What's up next, I hear you ask? Well, that's a great question. We've got some of the most important Pythonista tools in store: comprehensions and generators. These terms may not mean much to you yet, but soon you won't be able to forget them. In fact, you may even want more because they are that useful. List comprehensions, for example, allow us to create lists from other lists or from columns of DataFrames, among many other objects. Why would we want to do this? Check this out, for example. Let's say that we have a DataFrame with a column of time-stamped data and we want to extract the year from it. We could extract the column from the DataFrame, loop over its entries using, for example, a for loop, and within the for loop, extract the desired information. In that case, we would have had to initialize an empty list in which to store the retrieved data. A list comprehension, however, allows us to perform this entire operation, in a single line of code and it's waaaaaaay more efficient than the equivalent for loop code. This is one of a number of Pythonista Data Science chops that you'll learn in the rest of this course.

## 3. Let's practice!

Moreover, the entire final chapter will be dedicated to using the skills you're developing to extract meaningful information from a real-world dataset, the World Bank World Development Indicators dataset which, according to the World Bank, 'presents the most current and accurate global development data available, and includes national, regional and global estimates.' See you in the next Chapter!

**Daily XP150**

# List comprehensions

**50 XP**

## 1. List comprehensions

Let's say that you have a list of numbers and you want to create a new list of numbers that's the same as the old list, except that each number has 1 added to it.

## 2. Populate a list with a for loop

You could initialize a new empty list, loop through the old list, add 1 to each entry and append all new values to the new list, but for loops are inefficient, both computationally and in terms of coding time and space, particularly when you could do this in one line of code. "One line of code?" I hear you asking.

## 3. A list comprehension

Welcome to the wonderful world of list comprehensions! The syntax is as follows: within square brackets, you write the values you wish to create, otherwise known as the output expression, followed by the for clause referencing the original list. So in our case, you open the square bracket, followed by num + 1 for num in nums and then you close the square bracket. This is a list comprehension and creates precisely the desired list!

## 4. For loop and list comprehension syntax

See here the relationship between the for loop syntax and the list comprehension syntax. The power of list comprehensions is not merely relegated to the world of lists, however, you can write a list comprehension over any iterable.

## 5. List comprehension with range()

Here's an example of a list comprehension using a range object. To summarize,

## 6. List comprehensions

list comprehensions collapse for loops for building lists into a single line and the required components are 1) an iterable, 2) an iterator variable that represents the members of the iterable and 3) an output expression. That's it. You can also use list comprehensions in place of nested for loops.

## 7. Nested loops (1)

For example, lets say that we wanted to create a list of all pairs of integers where the first integer is between 0 and 1 and the second between 6 and 7. This nested for loop would produce the required result. The question is, can we do the same with a list comprehension? And the answer is, yes, as follows.

## 8. Nested loops (2)

Once again, within the square brackets, place the desired output expression followed by the two required for loop clauses. You may observe that while it keeps to a single line of code, we sacrifice some readability of the code as a tradeoff, so you'll have to consider if you'd like to use list comprehensions in cases such as this. The more often you use this, the more you get used to reading list comprehensions, so readability may not be a problem for you later on. But do remember that others may have to read your code as well!

## 9. Let's practice!

Now you've seen a few list comprehensions, it's your turn to write some! -

**Daily XP200**

**Exercise**

**Exercise**

**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?

**Instructions**

**50 XP**

**Possible Answers**

* 

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] for doc 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'].

In [1]:

[doc[0] for doc in doctor]

Out[1]:

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

**Daily XP250**

**Exercise**

**Exercise**

**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

**Instructions**

**50 XP**

**Possible Answers**

* 

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**.

**Daily XP250**

**Exercise**

**Exercise**

**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

**Instructions**

**50 XP**

**Possible Answers**

* 

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.

# Create list comprehension: squares

squares = [\_\_\_\_ for \_\_\_\_ in \_\_\_\_]

# Create list comprehension: squares

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

print(squares)

# Create list comprehension: squares

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

# Create list comprehension: squares

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

print(squares)

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

<script.py> output:

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

**Daily XP400**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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 = [[\_\_\_\_] \_\_\_\_]
* # Print the matrix
* for row in matrix:
* print(row)

# 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)

#matrix = [[j for j in range(5)] for i in range(5)]

# 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)

matrix = [[j for j in range(5)] for i in range(5)]

[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]

**Daily XP400**

# Advanced comprehensions

**50 XP**

This course is also available on the mobile app

## 1. Advanced comprehensions

Now that you know the basics of list comprehensions, lets check out some more

## 2. Conditionals in comprehensions

advanced comprehension capabilities, such as conditionals! Here we see that we can filter the output of a list comprehension using a conditional on the iterable: in this example, the resulting list is the square of the values in range(10) under the condition that the value itself is even. If you have not seen it before, the percent operation that you see being used in the comprehension is called the modulo operator. We can look at the Python documentation to see how the modulo operator is used and it shows that it produces the remainder from the division of the first argument by the second. Thus an integer modulo two is equal to zero if and only if the integer is even.

## 3. Conditionals in comprehensions

We can also condition the list comprehension on the output expression. Here, for an even integer we output its square. In any other case, signified by the else clause, that is for odd integers, we output 0.

## 4. Dict comprehensions

Now we can also write dictionary comprehensions to create new dictionaries from iterables. The syntax is almost the same as in list comprehensions and there are 2 differences. One, we use curly braces instead of square brackets. Two, the key and value are separated by a colon in the output expression as we can see here. In this example, we are creating a dictionary with keys positive integers and corresponding values the respective negative integers.

## 5. Let's practice!

Now that we have a grasp on advanced list comprehensions and dictionary comprehensions, it's time to practice! -

**Daily XP450**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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 = [\_\_\_\_ for \_\_\_\_ in fellowship \_\_\_\_]

# Print the new list

print(new\_fellowship)

# 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)

# 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']

**Daily XP650**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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 = [\_\_\_\_ for \_\_\_\_ in fellowship]

# Print the new list

print(new\_fellowship)

# 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)

# 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', '']

**Daily XP750**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

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 = \_\_\_\_

# Print the new dictionary

print(new\_fellowship)

# 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)

# 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}

**Daily XP850**

# Introduction to generator expressions

**50 XP**

This course is also available on the mobile app

## 1. Introduction to generator expressions

Now that you're getting a bit more comfortable with comprehensions, we're going to check out generators, which are related to comprehensions in way that will soon become evident.

## 2. Generator expressions

Recall that this list comprehension will create a list of the first 10 even numbers. Now lets replace the square brackets with round parentheses and voila! Something called a generator object has been created.

## 3. List comprehensions vs. generators

Now the question on everybody's lips is, "What is this generator object?" Well, a generator is like a list comprehension except it does not store the list in memory: it does not construct the list, but is an object we can iterate over to produce elements of the list as required.

## 4. Printing values from generators (1)

Here we can see that looping over a generator expression produces the elements of the analogous list. We can also pass a generator to the function list to create the list. Moreover,

## 5. Printing values from generators (2)

like any other iterator, we can pass a generator to the function next in order to iterate through its elements. For the geeks like me, this is an example of something called lazy evaluation, whereby the evaluation of the expression is delayed until its value is needed. This can help a great deal when working with extremely large sequences as you don't want to store the entire list in memory, which is what comprehensions would do; you want to generate elements of the sequence on the fly.

## 6. Generators vs. list comprehensions

Let's say that we wanted to iterate over a very large sequence of numbers, such as from 0 up to 10 to the power of a million, or at least wanted to do so until another condition was satisfied. Look what happens when I try to build such an iterable list using a comprehension on DataCamp's servers.

## 7. Generators vs. list comprehensions

My colleagues disconnect me because the list I'm trying to create can't even be stored in memory! Be warned though, don't try this at home, on our servers or yours!

## 8. Generators vs. list comprehensions

Check this out, however: I can easily create the analogous generator object because it does not yet create the entire list.

## 9. Conditionals in generator expressions

What's really cool is that anything we can do in a list comprehension such as filtering and applying conditionals, we can also do in a generator expression, such as you see here. You'll get a whole bunch of practice with this in the upcoming exercises.

## 10. Generator functions

The last thing to discuss before you get coding is the ability to write generator functions. Generator functions are functions that, when called, produce generator objects. Generator functions are written with the syntax of any other user-defined function, however instead of returning values using the keyword return, they yield sequences of values using the keyword yield.

## 11. Build a generator function

Here I have defined a generator function that, when called with a number n, produces a generator object that generates integers 0 though n. We can see within the function definition that i is initialized to 0 and that the first time the generator object is called, it yields i equal to 0. It then adds one to i and will then yield one on the next iteration and so on. The while loop is true until i equals equals n and then the generator ceases to yield values.

## 12. Use a generator function

This generator function can be called as you do any other function. Here I call the generator function with the argument, 5. We see that it produces a generator object and that we can iterate over this generator object with a for loop to print the values it yields. Generator functions are a powerful and customizable way to create generators.

## 13. Let's practice!

You'll have much practice with these in the coming exercises. Happy generating! -

**Daily XP900**

**Exercise**

**Exercise**

**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.

**Instructions**

**50 XP**

**Possible Answers**

* 

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.

**Daily XP950**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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 = \_\_\_\_
* # Print the first 5 values
* print(\_\_\_\_)
* print(\_\_\_\_)
* print(\_\_\_\_)
* print(\_\_\_\_)
* print(\_\_\_\_)
* # Print the rest of the values
* for value in \_\_\_\_:
* print(value)

# 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)

# Create generator object: result

result = (num for num in range(0,31))

# Print the first 5 values

print(next(result))

print(next(result))

print(next(result))

print(next(result))

print(next(result))

# Print the rest of the values

for value in result:

    print(value)

# Create generator object: result

result = (num for num in range(0,31))

# Print the first 5 values

print(next(result))

print(next(result))

print(next(result))

print(next(result))

print(next(result))

# Print the rest of the values

for value in result:

print(value)

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

**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.

**Instructions**

**100 XP**

* 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 = \_\_\_\_
* # Iterate over and print the values in lengths
* for value in \_\_\_\_:
* print(value)

# 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)

# 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)

**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)

**Instructions**

**100 XP**

* 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 \_\_\_\_:
* """Generator function that yields the
* length of the strings in input\_list."""
* # Yield the length of a string
* for person in input\_list:
* \_\_\_\_
* # Print the values generated by get\_lengths()
* for value in \_\_\_\_:
* print(value)

# 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)

# 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

**Daily XP1300**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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!

# Extract the created\_at column from df: tweet\_time

tweet\_time = \_\_\_\_

# Extract the clock time: tweet\_clock\_time

tweet\_clock\_time = [\_\_\_\_]

# Print the extracted times

print(tweet\_clock\_time)

# Extract the created\_at column from df: tweet\_time

tweet\_time = df['created\_at']

#print(tweet\_time)

# 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)

# Extract the created\_at column from df: tweet\_time

tweet\_time = df['created\_at']

#print(tweet\_time)

# 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'

|  |
| --- |
| Instructions |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| 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'. | |
|  | |
| Instructions |
|  |

|  |
| --- |
| -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! |
|  |

|  |
| --- |
| ''' |
|  |

|  |
| --- |
| # Import packages |
|  |

|  |
| --- |
| import pandas as pd |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| df = pd.read\_csv('../\_datasets/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)

|  |
| --- |
| ''' |
|  |

|  |
| --- |
| # Import packages |
|  |

|  |
| --- |
| import pandas as pd |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| df = pd.read\_csv('../\_datasets/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 if entry[17:19] == '19'] |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Print the extracted times |
|  |

print(tweet\_clock\_time)

# Extract the created\_at column from df: tweet\_time

tweet\_time = df['created\_at']

print(tweet\_time)

# 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)

# Extract the created\_at column from df: tweet\_time

tweet\_time = df['created\_at']

print(tweet\_time)

# 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)

0 Tue Mar 29 23:40:17 +0000 2016

1 Tue Mar 29 23:40:17 +0000 2016

2 Tue Mar 29 23:40:17 +0000 2016

3 Tue Mar 29 23:40:17 +0000 2016

4 Tue Mar 29 23:40:17 +0000 2016

...

95 Tue Mar 29 23:40:19 +0000 2016

96 Tue Mar 29 23:40:19 +0000 2016

97 Tue Mar 29 23:40:19 +0000 2016

98 Tue Mar 29 23:40:19 +0000 2016

99 Tue Mar 29 23:40:19 +0000 2016

Name: created\_at, Length: 100, dtype: object

['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'

**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.

**Instructions**

**100 XP**

* 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 = \_\_\_\_
* # Extract the clock time: tweet\_clock\_time
* tweet\_clock\_time = [\_\_\_\_ for \_\_\_\_ in \_\_\_\_ if \_\_\_\_ == \_\_\_\_]
* # Print the extracted times
* print(tweet\_clock\_time)

# 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)

# 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']

# Welcome to the case study!

**50 XP**

This course is also available on the mobile app

## 1. Welcome to the case study!

Ok so in this course, along with its prequel, you have learnt a formidable number of serious Pythonista Data Science chops. It's now time to flex those muscles and see what metal you're made of! In this chapter, you'll use your recently-gained skills of writing user-defined functions, iterators, list comprehensions and generators to wrangle and extract meaningful information from a real-world dataset,

## 2. World bank data

the World Bank World Development Indicators dataset. This dataset contains data on 217 world economies for over half a century, from 1960 up until 2015. The data contains hundreds of indicators from population, electricity consumption and CO2 emissions to literacy rates, unemployment and mortality rates. In the following few exercises, you'll use dictionaries, zipping, user-defined functions, and list comprehensions to wrangle some of this wonderfully rich dataset. Let's quickly recap these tools:

## 3. Using zip()

The zip function accepts an arbitrary number of iterables and returns an iterator of tuples. We bring back our two lists from before, one of the avengers, the other of their names. Passing these to the zip function creates a zip object which is an iterator of tuples. We can turn them into a list and then print this list.

## 4. Defining a function

For writing functions, function headers begin with the keyword def, followed by the function name, arguments inside parentheses and a colon. We then have the function body, with the docstrings enclosed in triple quotation marks; the rest of the function body performs the computation that the function does and closes with the keyword return, followed by the value or values to return.

## 5. Re-cap: list comprehensions

Comprehensions, in their most basic forms, are enclosed in square brackets and are structured as output expression for iterator variable in iterable. More advanced comprehensions can include conditionals on the output expression and/or conditionals on the iterable.

## 6. Let's practice!

Now that we've recapped all the tools that you'll need to hack away at this dataset in the next few exercises, let's get hacking! -

**Daily XP1550**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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.
* # Zip lists: zipped\_lists
* zipped\_lists = \_\_\_\_
* # Create a dictionary: rs\_dict
* rs\_dict = \_\_\_\_
* # Print the dictionary
* print(rs\_dict)

# 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)

# 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'}

**Daily XP1650**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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 \_\_\_\_(\_\_\_\_, \_\_\_\_):

    """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

# Call lists2dict: rs\_fxn

rs\_fxn = \_\_\_\_

# Print rs\_fxn

print(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)

# 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'

**Daily XP1750**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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.

# Print the first two lists in row\_lists

print(\_\_\_\_)

print(\_\_\_\_)

# Turn list of lists into list of dicts: list\_of\_dicts

list\_of\_dicts = [\_\_\_\_ for \_\_\_\_ in \_\_\_\_]

# Print the first two dictionaries in list\_of\_dicts

print(\_\_\_\_)

print(\_\_\_\_)

# Print the first two lists in row\_lists

print(row\_lists[0])

print(row\_lists[1])

# Turn list of lists into list of dicts: list\_of\_dicts

list\_of\_dicts = [lists2dict(feature\_names, sublist) for sublist in row\_lists]

# Print the first two dictionaries in list\_of\_dicts

print(list\_of\_dicts[0])

print(list\_of\_dicts[1])

# Print the first two lists in row\_lists

print(row\_lists[0])

print(row\_lists[1])

# Turn list of lists into list of dicts: list\_of\_dicts

#list\_of\_dicts = {feature\_names:row\_lists for sublist in lists2dict(row\_lists[0],row\_lists[1])}

list\_of\_dicts = [lists2dict(feature\_names, sublist) for sublist in row\_lists]

# Print the first two dictionaries in list\_of\_dicts

print(list\_of\_dicts[0])

print(list\_of\_dicts[1])

['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']

{'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'}

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

**Daily XP1850**

**Exercise**

**Exercise**

**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!

**Instructions**

**100 XP**

* 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

# 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 = \_\_\_\_

# Print the head of the DataFrame

# 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 IndicatorName IndicatorCode Year Value

0 Arab World ARB Adolescent fertility rate (births per 1,000 wo... SP.ADO.TFRT 1960 133.56090740552298

1 Arab World ARB Age dependency ratio (% of working-age populat... SP.POP.DPND 1960 87.7976011532547

2 Arab World ARB Age dependency ratio, old (% of working-age po... SP.POP.DPND.OL 1960 6.634579191565161

3 Arab World ARB Age dependency ratio, young (% of working-age ... SP.POP.DPND.YG 1960 81.02332950839141

4 Arab World ARB Arms exports (SIPRI trend indicator values) MS.MIL.XPRT.KD 1960 3000000.0

# 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())

# 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())

# Open a connection to the file

with \_\_\_\_ as \_\_\_\_:

    # Skip the column names

    file.readline()

    # Initialize an empty dictionary: counts\_dict

    counts\_dict = {}

    # Process only the first 1000 rows

    for j in \_\_\_\_:

        # Split the current line into a list: line

        line = file.readline().split(',')

        # Get the value for the first column: first\_col

        first\_col = line[0]

        # If the column value is in the dict, increment its value

        if first\_col in counts\_dict.keys():

            counts\_dict[first\_col] += 1

        # Else, add to the dict and set value to 1

        else:

            counts\_dict[first\_col] = 1

# Print the resulting dictionary

print(counts\_dict)

**Daily XP2000**

**Exercise**

**Exercise**

**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)

**Instructions**

**100 XP**

* 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('world\_dev\_ind.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
* # Print the resulting dictionary
* print(counts\_dict)

# Open a connection to the file

with open('world\_dev\_ind.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

# Print the resulting dictionary

print(counts\_dict)

{'Arab World': 80, 'Caribbean small states': 77, 'Central Europe and the Baltics': 71, 'East Asia & Pacific (all income levels)': 122, 'East Asia & Pacific (developing only)': 123, 'Euro area': 119, 'Europe & Central Asia (all income levels)': 109, 'Europe & Central Asia (developing only)': 89, 'European Union': 116, 'Fragile and conflict affected situations': 76, 'Heavily indebted poor countries (HIPC)': 18}

**Daily XP2100**

**Exercise**

**Exercise**

**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!

**Instructions**

**100 XP**

* 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 = \_\_\_\_

        # Break if this is the end of the file

        if not data:

            break

        # Yield the line of data

# Open a connection to the file

with open('world\_dev\_ind.csv') as file:

    # Create a generator object for the file: gen\_file

    gen\_file = \_\_\_\_

    # Print the first three lines of the file

    print(\_\_\_\_)

    print(\_\_\_\_)

    print(\_\_\_\_)

# Define read\_large\_file()

def read\_large\_file(file\_object):

    """A generator function to read a large file lazily."""

    # Loop indefinitely until the end of the file

    while True:

        # Read a line from the file: data

        data = file\_object.readline()

        # Break if this is the end of the file

        if not data:

            break

        # Yield the line of data

        yield data

# Open a connection to the file

with open('world\_dev\_ind.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))

# Define read\_large\_file()

def read\_large\_file(file\_object):

"""A generator function to read a large file lazily."""

# Loop indefinitely until the end of the file

while True:

# Read a line from the file: data

data = file\_object.readline()

# Break if this is the end of the file

if not data:

break

# Yield the line of data

yield data

# Open a connection to the file

with open('world\_dev\_ind.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,IndicatorName,IndicatorCode,Year,Value

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

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!

**Daily XP100**

**Exercise**

**Exercise**

**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!

**Instructions**

**100 XP**

* 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 \_\_\_\_ as \_\_\_\_:

    # Iterate over the generator from read\_large\_file()

    for line in \_\_\_\_:

        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)

# Initialize an empty dictionary: counts\_dict

counts\_dict = {}

# Open a connection to the file

with open('world\_dev\_ind.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)

# Initialize an empty dictionary: counts\_dict

counts\_dict = {}

# Open a connection to the file

with open('world\_dev\_ind.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': 80, 'Caribbean small states': 77, 'Central Europe and the Baltics': 71, 'East Asia & Pacific (all income levels)': 122, 'East Asia & Pacific (developing only)': 123, 'Euro area': 119, 'Europe & Central Asia (all income levels)': 109, 'Europe & Central Asia (developing only)': 89, 'European Union': 116, 'Fragile and conflict affected situations': 76, 'Heavily indebted poor countries (HIPC)': 99, 'High income': 131, 'High income: nonOECD': 68, 'High income: OECD': 127, 'Latin America & Caribbean (all income levels)': 130, 'Latin America & Caribbean (developing only)': 133, 'Least developed countries: UN classification': 78, 'Low & middle income': 138, 'Low income': 80, 'Lower middle income': 126, 'Middle East & North Africa (all income levels)': 89, 'Middle East & North Africa (developing only)': 94, 'Middle income': 138, 'North America': 123, 'OECD members': 130, 'Other small states': 63, 'Pacific island small states': 66, 'Small states': 69, 'South Asia': 36

## 1. Using pandas' read\_csv iterator for streaming data

You're on the home stretch now! That was some great work with generators to read in large and/or streaming files.

## 2. Reading files in chunks

In the remaining exercises, you'll revisit the pandas function read\_csv, along with its chunksize argument in order to read in the World Indicator data in chunks, process each chunk and aggregate. Herein, you'll do so in order to look at specific indicators in specific countries and to plot the results over all the years for which the relevant data exists. You'll then try your hand at writing a function to perform the same task which will save so much work when you want to look at the data for a different country.

## 3. Let's practice!

Good luck, stay cool and I'll see you again after you explore the data further and consolidate the skills that you've built in these courses. Enjoy!

# Import the pandas package

import pandas as pd

# Initialize reader object: df\_reader

df\_reader = pd.read\_csv('ind\_pop.csv', chunksize=10)

# Print two chunks

print(next(df\_reader))

print(next(df\_reader))

# Import the pandas package

import pandas as pd

# Initialize reader object: df\_reader

df\_reader = pd.read\_csv('ind\_pop.csv', chunksize=10)

# Print two chunks

print(next(df\_reader))

print(next(df\_reader))

CountryName CountryCode IndicatorName IndicatorCode Year Value

0 Arab World ARB Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 31.285

1 Caribbean small states CSS Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 31.597

2 Central Europe and the Baltics CEB Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 44.508

3 East Asia & Pacific (all income levels) EAS Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 22.471

4 East Asia & Pacific (developing only) EAP Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 16.918

5 Euro area EMU Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 62.097

6 Europe & Central Asia (all income levels) ECS Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 55.379

7 Europe & Central Asia (developing only) ECA Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 38.066

8 European Union EUU Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 61.213

9 Fragile and conflict affected situations FCS Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 17.892

CountryName CountryCode IndicatorName IndicatorCode Year Value

10 Heavily indebted poor countries (HIPC) HPC Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 12.236

11 High income HIC Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 62.680

12 High income: nonOECD NOC Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 56.108

13 High income: OECD OEC Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 64.285

14 Latin America & Caribbean (all income levels) LCN Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 49.285

15 Latin America & Caribbean (developing only) LAC Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 44.863

16 Least developed countries: UN classification LDC Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 9.616

17 Low & middle income LMY Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 21.273

18 Low income LIC Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 11.498

19 Lower middle income LMC Urban population (% of total) SP.URB.TOTL.IN.ZS 1960 19.811

**Daily XP350**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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.
* 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.

# Initialize reader object: urb\_pop\_reader

urb\_pop\_reader = pd.read\_csv(\_\_\_\_, \_\_\_\_)

# Get the first DataFrame chunk: df\_urb\_pop

df\_urb\_pop = next(\_\_\_\_)

# Check out the head of the DataFrame

print(df\_urb\_pop.head())

# Check out specific country: df\_pop\_ceb

df\_pop\_ceb = df\_urb\_pop[\_\_\_\_]

# Zip DataFrame columns of interest: pops

pops = zip(\_\_\_\_, \_\_\_\_)

# Turn zip object into list: pops\_list

pops\_list = list(pops)

# Print pops\_list

print(pops\_list)

# Initialize reader object: urb\_pop\_reader

urb\_pop\_reader = pd.read\_csv('ind\_pop\_data.csv', chunksize=1000)

# Get the first DataFrame chunk: df\_urb\_pop

df\_urb\_pop = next(urb\_pop\_reader)

# Check out the head of the DataFrame

print(df\_urb\_pop.head())

# Check out specific country: df\_pop\_ceb

df\_pop\_ceb = df\_urb\_pop[df\_urb\_pop['CountryCode']=='CEB']

# Zip DataFrame columns of interest: pops

pops = zip(df\_pop\_ceb['Total Population'],df\_pop\_ceb['Urban population (% of total)'])

# Turn zip object into list: pops\_list

pops\_list = list(pops)

# Print pops\_list

print(pops\_list)

# Initialize reader object: urb\_pop\_reader

urb\_pop\_reader = pd.read\_csv('ind\_pop\_data.csv', chunksize=1000)

# Get the first DataFrame chunk: df\_urb\_pop

df\_urb\_pop = next(urb\_pop\_reader)

# Check out the head of the DataFrame

print(df\_urb\_pop.head())

# Check out specific country: df\_pop\_ceb

df\_pop\_ceb = df\_urb\_pop[df\_urb\_pop['CountryCode']=='CEB']

# Zip DataFrame columns of interest: pops

pops = zip(df\_pop\_ceb['Total Population'],df\_pop\_ceb['Urban population (% of total)'])

# Turn zip object into list: pops\_list

pops\_list = list(pops)

# Print pops\_list

print(pops\_list)

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

0 Arab World ARB 1960 9.250e+07 31.285

1 Caribbean small states CSS 1960 4.191e+06 31.597

2 Central Europe and the Baltics CEB 1960 9.140e+07 44.508

3 East Asia & Pacific (all income levels) EAS 1960 1.042e+09 22.471

4 East Asia & Pacific (developing only) EAP 1960 8.965e+08 16.918

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

**Daily XP450**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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('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'] = [\_\_\_\_]
* # Plot urban population data
* df\_pop\_ceb.plot(kind=\_\_\_\_, x=\_\_\_\_, y=\_\_\_\_)
* plt.show()

|  |
| --- |
| Initialize reader object: urb\_pop\_reader |
|  |

|  |
| --- |
| urb\_pop\_reader = pd.read\_csv('ind\_pop\_data.csv', chunksize=1000) |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Get the first DataFrame chunk: df\_urb\_pop |
|  |

|  |
| --- |
| df\_urb\_pop = next(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] |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| # Plot urban population data |
|  |

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

plt.show()

# Code from previous exercise

urb\_pop\_reader = pd.read\_csv('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(tuple[0] \* tuple[1] \* 0.01) for tuple in pops\_list]

# Plot urban population data

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

plt.show()

print(pops\_list)

# Code from previous exercise

urb\_pop\_reader = pd.read\_csv('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(tuple[0] \* tuple[1] \* 0.01) for tuple in pops\_list]

# Plot urban population data

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

plt.show()

**Daily XP550**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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.
* Concatenate data and df\_pop\_ceb by passing a list of the DataFrames to pd.concat().
* # Initialize reader object: urb\_pop\_reader
* urb\_pop\_reader = pd.read\_csv('ind\_pop\_data.csv', chunksize=1000)
* # Initialize empty DataFrame: data
* data = \_\_\_\_
* # Iterate over each DataFrame chunk
* for df\_urb\_pop in \_\_\_\_:
* # 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]
* # Concatenate DataFrame chunk to the end of data: data
* data = \_\_\_\_
* # Plot urban population data
* data.plot(kind='scatter', x='Year', y='Total Urban Population')
* plt.show()

# Initialize reader object: urb\_pop\_reader

urb\_pop\_reader = pd.read\_csv('ind\_pop\_data.csv', chunksize=1000)

# Initialize empty DataFrame: data

data = pd.DataFrame()

# Iterate over each DataFrame chunk

for df\_urb\_pop in urb\_pop\_reader:

    # Check out specific country: df\_pop\_ceb

    df\_pop\_ceb = df\_urb\_pop[df\_urb\_pop['CountryCode'] == 'CEB']

    # Zip DataFrame columns of interest: pops

    pops = zip(df\_pop\_ceb['Total Population'],

                df\_pop\_ceb['Urban population (% of total)'])

    # Turn zip object into list: pops\_list

    pops\_list = list(pops)

    # Use list comprehension to create new DataFrame column 'Total Urban Population'

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

    # Concatenate DataFrame chunk to the end of data: data

    data = pd.concat([data, df\_pop\_ceb])

# Plot urban population data

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

plt.show()

# Initialize reader object: urb\_pop\_reader urb\_pop\_reader = pd.read\_csv('ind\_pop\_data.csv', chunksize=1000) # Initialize empty DataFrame: data data = pd.DataFrame() # Iterate over each DataFrame chunk for df\_urb\_pop in urb\_pop\_reader: # Check out specific country: df\_pop\_ceb df\_pop\_ceb = df\_urb\_pop[df\_urb\_pop['CountryCode'] == 'CEB'] # Zip DataFrame columns of interest: pops pops = zip(df\_pop\_ceb['Total Population'], df\_pop\_ceb['Urban population (% of total)']) # Turn zip object into list: pops\_list pops\_list = list(pops) # Use list comprehension to create new DataFrame column 'Total Urban Population' df\_pop\_ceb['Total Urban Population'] = [int(tup[0] \* tup[1] \* 0.01) for tup in pops\_list] # Concatenate DataFrame chunk to the end of data: data data = pd.concat([data, df\_pop\_ceb]) # Plot urban population data data.plot(kind='scatter', x='Year', y='Total Urban Population') plt.show()

**Daily XP650**

**Exercise**

**Exercise**

**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.

**Instructions**

**100 XP**

* 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.
* 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'.
* # Define plot\_pop()
* def \_\_\_\_(\_\_\_\_, \_\_\_\_):
* # 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]
* # Concatenate DataFrame chunk to the end of data: data
* data = pd.concat([data, df\_pop\_ceb])
* # Plot urban population data
* data.plot(kind='scatter', x='Year', y='Total Urban Population')
* plt.show()
* # Set the filename: fn
* fn = 'ind\_pop\_data.csv'
* # Call plot\_pop for country code 'CEB'
* # Call plot\_pop for country code 'ARB'
* # 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]
* # Concatenate DataFrame chunk to the end of data: data
* data = pd.concat([data, df\_pop\_ceb])
* # Plot urban population data
* data.plot(kind='scatter', x='Year', y='Total Urban Population')
* plt.show()
* # Set the filename: fn
* fn = 'ind\_pop\_data.csv'
* # Call plot\_pop for country code 'CEB'
* plot\_pop('ind\_pop\_data.csv', "CEB")
* # Call plot\_pop for country code 'ARB'
* plot\_pop('ind\_pop\_data.csv', 'ARB')
* # 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]
* # Concatenate DataFrame chunk to the end of data: data
* data = pd.concat([data, df\_pop\_ceb])
* # Plot urban population data
* data.plot(kind='scatter', x='Year', y='Total Urban Population')
* plt.show()
* # Set the filename: fn
* fn = 'ind\_pop\_data.csv'
* # Call plot\_pop for country code 'CEB'
* plot\_pop(fn, 'CEB')
* # Call plot\_pop for country code 'ARB'
* plot\_pop(fn, 'ARB')