Chapter 1 - Using iterators in PythonLand

Here, you'll learn all about iterators and iterables, which you have already worked with before when writing for loops! You'll learn about some very useful functions that will allow you to effectively work with iterators and finish the chapter with a use case that is pertinent to the world of Data Science - dealing with large amounts of data - in this case, data from Twitter that you will load in chunks using iterators!

Iterators vs Iterables

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.

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.

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

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


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

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

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

In [ ]:
```
superspeed
```

Out[ ]:
```
<list_iterator at 0x2b36b146560>
```

Iterating over iterables (2)

One of the things you learned about in this chapter is that not all iterables are actual lists. A couple of examples that we looked at are strings and the use of the range() function. In this exercise, we will focus on the range() function.

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

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

Recall that range() doesn't actually create the list; instead, it creates a range object with an iterator that produces the values until it reaches the limit (in the example, until the value 4). If range() created the actual list, calling it with a value of 10100 may not work, especially since a number as big as that may go over a regular computer's memory. The value 10100 is actually what's called a **Googol** which is a 1 followed by a hundred 0s. That's a huge number!

Your task for this exercise is to show that calling range() with 10100 won't actually pre-create the list.

In [ ]:
```
range(3)
```

Out[ ]:
```
range(0, 3)
```

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

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

# Loop over range(3) and print the values
for 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))
print(next(googol))
print(next(googol))
print(next(googol))
print(next(googol))
```

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

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.

In [ ]:
```
# 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
```

Using enumerate

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

You've just gained several new ideas on iterators and one of them is the `enumerate()` function. 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.

In [ ]:
```
# 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
```

Using zip

Another interesting function 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.

In [ ]:
```
aliases = ['prof x', 'iceman', 'nightcrawler', 'magneto', 'shadowcat']
powers = ['telepathy', 'thermokinesis', 'teleportation', 'magnetokinesis', 'intangibility']
```

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

# Print the list of tuples
print(mutant_data)

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

# Print the zip object
print(mutant_zip)

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

```
[('charles xavier', 'prof x', 'telepathy'), ('bobby drake', 'iceman', 'thermokinesis'), ('kurt wagner', 'nightcrawler', 'teleport
ation'), ('max eisenhardt', 'magneto', 'magnetokinesis'), ('kitty pryde', 'shadowcat', 'intangibility')]
<zip object at 0x000002B36D0E14C0>
charles xavier prof x telepathy
bobby drake iceman thermokinesis
kurt wagner nightcrawler teleportation
max eisenhardt magneto magnetokinesis
kitty pryde shadowcat intangibility
```

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.

In [ ]:
```
mutants = tuple(mutants)
powers = tuple(powers)
```

In [ ]:
```
# 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', 'magnetokine
sis') ('kitty pryde', 'intangibility')
True
True
```

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 exercises of the prequel course, but this time, working on it in chunks of 10 entries at a time.

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

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

# Iterate over the file chunk by chunk
```

```
file_path = 'C:\\Users\\yeiso\\OneDrive - Douglas College\\0. DOUGLAS COLLEGE\\3. Fund Machine Learning\\0. Python Course DataCamp
for chunk in pd.read_csv(file_path, 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}
```

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.

In [ ]:
```
# 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)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[27], line 23
     20     return counts_dict
     22 # Call count_entries(): result_counts
---> 23 result_counts = count_entries('tweets.csv', 10, 'lang')
     25 # Print result_counts
     26 print(result_counts)

Cell In[27], line 10, in count_entries(csv_file, c_size, colname)
      7 counts_dict = {}
      9 # Iterate over the file chunk by chunk
---> 10 for chunk in pd.read_csv(csv_file, chunksize = c_size):
     11
     12     # Iterate over the column in DataFrame
     13     for entry in chunk[colname]:
     14         if entry in counts_dict.keys():

File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\parsers\readers.py:948, in read_csv(filepath_or_buffer, sep, delimiter, header, names, index_col, usecols, dtype, engine, converters, true_values, false_values, skipinitialspace, skiprows, skipfooter, nrows, na_values, keep_default_na, na_filter, verbose, skip_blank_lines, parse_dates, infer_datetime_format, keep_date_col, date_parser, date_format, dayfirst, cache_dates, iterator, chunksize, compression, thousands, decimal, lineterminator, quotechar, quoting, doublequote, escapechar, comment, encoding, encoding_errors, dialect, on_bad_lines, delim_whitespace, low_memory, memory_map, float_precision, storage_options, dtype_backend)
    935 kwds_defaults = _refine_defaults_read(
    936     dialect,
    937     delimiter,
   (...)
    944     dtype_backend=dtype_backend,
    945 )
    946 kwds.update(kwds_defaults)
--> 948 return _read(filepath_or_buffer, kwds)

File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\parsers\readers.py:611, in _read(filepath_or_buffer, kwds)
    608 _validate_names(kwds.get("names", None))
    610 # Create the parser.
--> 611 parser = TextFileReader(filepath_or_buffer, **kwds)
    613 if chunksize or iterator:
    614     return parser

File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\parsers\readers.py:1448, in TextFileReader.__init__(self, f, engine, **kwds)
   1445     self.options["has_index_names"] = kwds["has_index_names"]
   1447 self.handles: IOHandles | None = None
-> 1448 self._engine = self._make_engine(f, self.engine)

File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\parsers\readers.py:1705, in TextFileReader._make_engine(self, f, engine)
```

```
   1703     if "b" not in mode:
   1704         mode += "b"
-> 1705 self.handles = get_handle(
   1706     f,
   1707     mode,
   1708     encoding=self.options.get("encoding", None),
   1709     compression=self.options.get("compression", None),
   1710     memory_map=self.options.get("memory_map", False),
   1711     is_text=is_text,
   1712     errors=self.options.get("encoding_errors", "strict"),
   1713     storage_options=self.options.get("storage_options", None),
   1714 )
   1715 assert self.handles is not None
   1716 f = self.handles.handle

File c:\Users\yeiso\AppData\Local\Programs\Python\Python312\Lib\site-packages\pandas\io\common.py:863, in get_handle(path_or_buf, mode, encoding, compression, memory_map, is_text, errors, storage_options)
    858 elif isinstance(handle, str):
    859     # Check whether the filename is to be opened in binary mode.
    860     # Binary mode does not support 'encoding' and 'newline'.
    861     if ioargs.encoding and "b" not in ioargs.mode:
    862         # Encoding
--> 863         handle = open(
    864             handle,
    865             ioargs.mode,
    866             encoding=ioargs.encoding,
    867             errors=errors,
    868             newline="",
    869         )
    870     else:
    871         # Binary mode
    872         handle = open(handle, ioargs.mode)

FileNotFoundError: [Errno 2] No such file or directory: 'tweets.csv'
```

Chapter - List comprehensions and generators

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

Writing list comprehensions

Your job in this exercise is to write a list comprehension that produces a list of the squares of the numbers ranging from 0 to 9.

In [ ]:
```
# Create list comprehension: squares
squares = [x**2 for x in range(0,10)]
squares
```

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

Nested list comprehensions

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

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

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

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

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

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

In [ ]:
```
matrix = [[x for x in range(5)] for x in range(5)]
```

In [ ]:
```
matrix
```

Out[ ]:
```
[[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]]
```

In [ ]:
```
# Create a 5 x 5 matrix using a list of lists: matrix
matrix = [[col for col in range(0,5)] for row in range(0,5) ]

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

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

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!

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.

In [ ]:
```
# 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']
```

Using conditionals in comprehensions (2)

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

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

In [ ]:
```
# 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', '']
```

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`.

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 : .

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.

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

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

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

List comprehensions vs generators

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.

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

A list comprehension produces a list as output, a generator produces a generator object.

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.

In [ ]:
```
# 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
```

In [ ]:
```
result
```

Out[ ]:
```
<generator object <genexpr> at 0x01354E70>
```

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.

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

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

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

```
6
5
5
6
7
```

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!

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

In [ ]:
```
# 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
```

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.

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

In [ ]:
```
df.head()
```

Out[ ]:

	contributors	coordinates	created_at	entities	extended_entities	favorite_count	favorited	filter_level	geo	id	...	quoted_st
0	NaN	NaN	Tue Mar 29 23:40:17 +0000 2016	{'hashtags': [], 'user_mentions': [{'screen_na...	{'media': [{'sizes': {'large': {'w': 1024, 'h'...	0	False	low	NaN	714960401759387648	...	
1	NaN	NaN	Tue Mar 29 23:40:17 +0000 2016	{'hashtags': [{'text': 'cruzsexscandal', 'indi...	{'media': [{'sizes': {'large': {'w': 500, 'h':...	0	False	low	NaN	714960401977319424	...	
2	NaN	NaN	Tue Mar 29 23:40:17 +0000 2016	{'hashtags': [], 'user_mentions': [], 'symbols...	NaN	0	False	low	NaN	714960402426236928	...	
3	NaN	NaN	Tue Mar 29 23:40:17 +0000 2016	{'hashtags': [], 'user_mentions': [], 'symbols...	NaN	0	False	low	NaN	714960402367561730	...	7.1492
4	NaN	NaN	Tue Mar 29 23:40:17 +0000 2016	{'hashtags': [], 'user_mentions': [{'screen_na...	NaN	0	False	low	NaN	714960402149416960	...	

5 rows × 31 columns

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

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

# 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', '23:40:1
8', '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', '23:4
0:18', '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', '2
3:40:18', '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', '23:40:1
8', '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', '23:4
0:19', '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', '2
3:40:18', '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', '23:40:1
9', '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'.

In [ ]:
```
# 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', '23:40:1
9', '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', '23:4
0:19', '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', '2
3:40:19', '23:40:19']
```

Chapter 3 - Bringing it all together!

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

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. 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.

In [ ]:
```
row_vals = ['Arab World', 'ARB', 'Adolescent fertility rate (births per 1,000 women ages 15-19)', 'SP.ADO.TFRT', '1960', '133.5609
feature_names = ['Arab World', 'ARB', 'Adolescent fertility rate (births per 1,000 women ages 15-19)', 'SP.ADO.TFRT', '1960', '133
```

In [ ]:
```
# 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)
```

```
{'Arab World': 'Arab World', 'ARB': 'ARB', 'Adolescent fertility rate (births per 1,000 women ages 15-19)': 'Adolescent fertility
rate (births per 1,000 women ages 15-19)', 'SP.ADO.TFRT': 'SP.ADO.TFRT', '1960': '1960', '133.56090740552298': '133.5609074055229
8'}
```

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!

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

```
    the keys and list2 provides the values."""

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

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

    # Return the dictionary
    return rs_dict

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

# Print rs_fxn
print(rs_fxn)
```

```
{'Arab World': 'Arab World', 'ARB': 'ARB', 'Adolescent fertility rate (births per 1,000 women ages 15-19)': 'Adolescent fertility rate (births per 1,000 women ages 15-19)', 'SP.ADO.TFRT': 'SP.ADO.TFRT', '1960': '1960', '133.56090740552298': '133.56090740552298'}
```

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.

In [ ]:
```
row_lists = [['Arab World', 'ARB', 'Adolescent fertility rate (births per 1,000 women ages 15-19)', 'SP.ADO.TFRT', '1960', '133.56
```

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

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

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

```
['Arab World', 'ARB', 'Adolescent fertility rate (births per 1,000 women ages 15-19)', 'SP.ADO.TFRT', '1960', '133.56090740552298']
['Arab World', 'ARB', 'Age dependency ratio (% of working-age population)', 'SP.POP.DPND', '1960', '87.7976011532547']
{'Arab World': 'Arab World', 'ARB': 'ARB', 'Adolescent fertility rate (births per 1,000 women ages 15-19)': 'Adolescent fertility rate (births per 1,000 women ages 15-19)', 'SP.ADO.TFRT': 'SP.ADO.TFRT', '1960': '1960', '133.56090740552298': '133.56090740552298'}
{'Arab World': 'Arab World', 'ARB': 'ARB', 'Adolescent fertility rate (births per 1,000 women ages 15-19)': 'Age dependency ratio (% of working-age population)', 'SP.ADO.TFRT': 'SP.POP.DPND', '1960': '1960', '133.56090740552298': '87.7976011532547'}
```

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 of all 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.

In [ ]:
```
# 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
df.head()
```

Out[ ]:

	133.56090740552298	1960	ARB	Adolescent fertility rate (births per 1,000 women ages 15-19)	Arab World	SP.ADO.TFRT
0	133.56090740552298	1960	ARB	Adolescent fertility rate (births per 1,000 wo...	Arab World	SP.ADO.TFRT
1	87.7976011532547	1960	ARB	Age dependency ratio (% of working-age populat...	Arab World	SP.POP.DPND
2	6.634579191565161	1960	ARB	Age dependency ratio, old (% of working-age po...	Arab World	SP.POP.DPND.OL
3	81.02332950839141	1960	ARB	Age dependency ratio, young (% of working-age ...	Arab World	SP.POP.DPND.YG
4	3000000.0	1960	ARB	Arms exports (SIPRI trend indicator values)	Arab World	MS.MIL.XPRT.KD

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_ind_pop_data.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.

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

    # Skip the column names
    file.readline()

    # Initialize an empty dictionary: counts_dict
    counts_dict = {}

    # Process only the first 1000 rows
    for j in range(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': 5, 'Caribbean small states': 5, 'Central Europe and the Baltics': 5, 'East Asia & Pacific (all income levels)': 5,
'East Asia & Pacific (developing only)': 5, 'Euro area': 5, 'Europe & Central Asia (all income levels)': 5, 'Europe & Central Asi
a (developing only)': 5, 'European Union': 5, 'Fragile and conflict affected situations': 5, 'Heavily indebted poor countries (HI
PC)': 5, 'High income': 5, 'High income: nonOECD': 5, 'High income: OECD': 5, 'Latin America & Caribbean (all income levels)': 5,
'Latin America & Caribbean (developing only)': 5, 'Least developed countries: UN classification': 5, 'Low & middle income': 5, 'L
ow income': 5, 'Lower middle income': 5, 'Middle East & North Africa (all income levels)': 5, 'Middle East & North Africa (develo
ping only)': 5, 'Middle income': 5, 'North America': 5, 'OECD members': 5, 'Other small states': 5, 'Pacific island small state
s': 5, 'Small states': 5, 'South Asia': 5, 'Sub-Saharan Africa (all income levels)': 5, 'Sub-Saharan Africa (developing only)':
5, 'Upper middle income': 5, 'World': 4, 'Afghanistan': 4, 'Albania': 4, 'Algeria': 4, 'American Samoa': 4, 'Andorra': 4, 'Angol
a': 4, 'Antigua and Barbuda': 4, 'Argentina': 4, 'Armenia': 4, 'Aruba': 4, 'Australia': 4, 'Austria': 4, 'Azerbaijan': 4, '"Baham
as': 4, 'Bahrain': 4, 'Bangladesh': 4, 'Barbados': 4, 'Belarus': 4, 'Belgium': 4, 'Belize': 4, 'Benin': 4, 'Bermuda': 4, 'Bhuta
n': 4, 'Bolivia': 4, 'Bosnia and Herzegovina': 4, 'Botswana': 4, 'Brazil': 4, 'Brunei Darussalam': 4, 'Bulgaria': 4, 'Burkina Fas
o': 4, 'Burundi': 4, 'Cabo Verde': 4, 'Cambodia': 4, 'Cameroon': 4, 'Canada': 4, 'Cayman Islands': 4, 'Central African Republic':
4, 'Chad': 4, 'Channel Islands': 4, 'Chile': 4, 'China': 4, 'Colombia': 4, 'Comoros': 4, '"Congo': 8, 'Costa Rica': 4, "Cote d'Iv
oire": 4, 'Croatia': 4, 'Cuba': 4, 'Curacao': 4, 'Cyprus': 4, 'Czech Republic': 4, 'Denmark': 4, 'Djibouti': 4, 'Dominica': 4, 'D
ominican Republic': 4, 'Ecuador': 4, '"Egypt': 4, 'El Salvador': 4, 'Equatorial Guinea': 4, 'Eritrea': 4, 'Estonia': 4, 'Ethiopi
a': 4, 'Faeroe Islands': 4, 'Fiji': 4, 'Finland': 4, 'France': 4, 'French Polynesia': 4, 'Gabon': 4, '"Gambia': 4, 'Georgia': 4,
'Germany': 4, 'Ghana': 4, 'Greece': 4, 'Greenland': 4, 'Grenada': 4, 'Guam': 4, 'Guatemala': 4, 'Guinea': 4, 'Guinea-Bissau': 4,
'Guyana': 4, 'Haiti': 4, 'Honduras': 4, '"Hong Kong SAR': 4, 'Hungary': 4, 'Iceland': 4, 'India': 4, 'Indonesia': 4, '"Iran': 4,
'Iraq': 4, 'Ireland': 4, 'Isle of Man': 4, 'Israel': 4, 'Italy': 4, 'Jamaica': 4, 'Japan': 4, 'Jordan': 4, 'Kazakhstan': 4, 'Keny
a': 4, 'Kiribati': 4, '"Korea': 8, 'Kuwait': 4, 'Kyrgyz Republic': 4, 'Lao PDR': 4, 'Latvia': 4, 'Lebanon': 4, 'Lesotho': 4, 'Lib
eria': 4, 'Libya': 4, 'Liechtenstein': 4, 'Lithuania': 4, 'Luxembourg': 4, '"Macao SAR': 4, '"Macedonia': 4, 'Madagascar': 4, 'Ma
lawi': 4, 'Malaysia': 4, 'Maldives': 4, 'Mali': 4, 'Malta': 4, 'Marshall Islands': 4, 'Mauritania': 4, 'Mauritius': 4, 'Mexico':
4, '"Micronesia': 4, 'Moldova': 4, 'Monaco': 4, 'Mongolia': 4, 'Montenegro': 4, 'Morocco': 4, 'Mozambique': 4, 'Myanmar': 4, 'Nam
ibia': 4, 'Nepal': 4, 'Netherlands': 4, 'New Caledonia': 4, 'New Zealand': 4, 'Nicaragua': 4, 'Niger': 4, 'Nigeria': 4, 'Northern
Mariana Islands': 4, 'Norway': 4, 'Oman': 4, 'Pakistan': 4, 'Palau': 4, 'Panama': 4, 'Papua New Guinea': 4, 'Paraguay': 4, 'Per
u': 4, 'Philippines': 4, 'Poland': 4, 'Portugal': 4, 'Puerto Rico': 4, 'Qatar': 4, 'Romania': 4, 'Russian Federation': 4, 'Rwand
a': 4, 'Samoa': 4, 'San Marino': 4, 'Sao Tome and Principe': 4, 'Saudi Arabia': 4, 'Senegal': 4, 'Seychelles': 4, 'Sierra Leone':
4, 'Singapore': 4, 'Slovak Republic': 4, 'Slovenia': 4, 'Solomon Islands': 4, 'Somalia': 4, 'South Africa': 4, 'South Sudan': 4,
'Spain': 4, 'Sri Lanka': 4, 'St. Kitts and Nevis': 4, 'St. Lucia': 4, 'St. Vincent and the Grenadines': 4, 'Sudan': 4, 'Surinam
e': 4, 'Swaziland': 4, 'Sweden': 4, 'Switzerland': 4, 'Syrian Arab Republic': 4, 'Tajikistan': 4, 'Tanzania': 4, 'Thailand': 4,
'Timor-Leste': 4, 'Togo': 4, 'Tonga': 4, 'Trinidad and Tobago': 4, 'Tunisia': 4, 'Turkey': 4, 'Turkmenistan': 4, 'Turks and Caico
s Islands': 4, 'Tuvalu': 4, 'Uganda': 4, 'Ukraine': 4, 'United Arab Emirates': 4, 'United Kingdom': 4, 'United States': 4, 'Urugu
ay': 4, 'Uzbekistan': 4, 'Vanuatu': 4, '"Venezuela': 4, 'Vietnam': 4, 'Virgin Islands (U.S.)': 4, '"Yemen': 4, 'Zambia': 4, 'Zimb
abwe': 4}

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 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_ind_pop_data.csv' is in your current directory for your use.

Note that when you open a connection to a file, the resulting file object is already a generator! So out in the wild, you won't have to explicitly create generator objects in cases such as this. However, for pedagogical reasons, we are having you practice how to do this here with the read_large_file() function. Go for it!

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

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

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

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

        # Yield the line of data
        yield data

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

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

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

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

Arab World,ARB,1960,92495902.0,31.285384211605397

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

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

Writing a generator to load data in chunks (3)

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

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

1000 rows of data, you'll process the entire dataset!

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

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

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

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

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

# Print
print(counts_dict)
```

{'CountryName': 1, 'Arab World': 55, 'Caribbean small states': 55, 'Central Europe and the Baltics': 55, 'East Asia & Pacific (al
l income levels)': 55, 'East Asia & Pacific (developing only)': 55, 'Euro area': 55, 'Europe & Central Asia (all income levels)':
55, 'Europe & Central Asia (developing only)': 55, 'European Union': 55, 'Fragile and conflict affected situations': 55, 'Heavily
indebted poor countries (HIPC)': 55, 'High income': 55, 'High income: nonOECD': 55, 'High income: OECD': 55, 'Latin America & Car
ibbean (all income levels)': 55, 'Latin America & Caribbean (developing only)': 55, 'Least developed countries: UN classificatio
n': 55, 'Low & middle income': 55, 'Low income': 55, 'Lower middle income': 55, 'Middle East & North Africa (all income levels)':
55, 'Middle East & North Africa (developing only)': 55, 'Middle income': 55, 'North America': 55, 'OECD members': 55, 'Other smal
l states': 55, 'Pacific island small states': 55, 'Small states': 55, 'South Asia': 55, 'Sub-Saharan Africa (all income levels)':
55, 'Sub-Saharan Africa (developing only)': 55, 'Upper middle income': 55, 'World': 55, 'Afghanistan': 55, 'Albania': 55, 'Algeri
a': 55, 'American Samoa': 55, 'Andorra': 55, 'Angola': 55, 'Antigua and Barbuda': 55, 'Argentina': 55, 'Armenia': 55, 'Aruba': 5
5, 'Australia': 55, 'Austria': 55, 'Azerbaijan': 55, '"Bahamas': 55, 'Bahrain': 55, 'Bangladesh': 55, 'Barbados': 55, 'Belarus':
55, 'Belgium': 55, 'Belize': 55, 'Benin': 55, 'Bermuda': 55, 'Bhutan': 55, 'Bolivia': 55, 'Bosnia and Herzegovina': 55, 'Botswan
a': 55, 'Brazil': 55, 'Brunei Darussalam': 55, 'Bulgaria': 55, 'Burkina Faso': 55, 'Burundi': 55, 'Cabo Verde': 55, 'Cambodia': 5
5, 'Cameroon': 55, 'Canada': 55, 'Cayman Islands': 55, 'Central African Republic': 55, 'Chad': 55, 'Channel Islands': 55, 'Chil
e': 55, 'China': 55, 'Colombia': 55, 'Comoros': 55, '"Congo': 110, 'Costa Rica': 55, "Cote d'Ivoire": 55, 'Croatia': 55, 'Cuba':
55, 'Curacao': 55, 'Cyprus': 55, 'Czech Republic': 55, 'Denmark': 55, 'Djibouti': 55, 'Dominica': 55, 'Dominican Republic': 55,
'Ecuador': 55, '"Egypt': 55, 'El Salvador': 55, 'Equatorial Guinea': 55, 'Eritrea': 55, 'Estonia': 55, 'Ethiopia': 55, 'Faeroe Is
lands': 55, 'Fiji': 55, 'Finland': 55, 'France': 55, 'French Polynesia': 55, 'Gabon': 55, '"Gambia': 55, 'Georgia': 55, 'German
y': 55, 'Ghana': 55, 'Greece': 55, 'Greenland': 55, 'Grenada': 55, 'Guam': 55, 'Guatemala': 55, 'Guinea': 55, 'Guinea-Bissau': 5
5, 'Guyana': 55, 'Haiti': 55, 'Honduras': 55, '"Hong Kong SAR': 55, 'Hungary': 55, 'Iceland': 55, 'India': 55, 'Indonesia': 55,
'"Iran': 55, 'Iraq': 55, 'Ireland': 55, 'Isle of Man': 55, 'Israel': 55, 'Italy': 55, 'Jamaica': 55, 'Japan': 55, 'Jordan': 55,
'Kazakhstan': 55, 'Kenya': 55, 'Kiribati': 55, '"Korea': 110, 'Kuwait': 52, 'Kyrgyz Republic': 55, 'Lao PDR': 55, 'Latvia': 55,
'Lebanon': 55, 'Lesotho': 55, 'Liberia': 55, 'Libya': 55, 'Liechtenstein': 55, 'Lithuania': 55, 'Luxembourg': 55, '"Macao SAR': 5
5, '"Macedonia': 55, 'Madagascar': 55, 'Malawi': 55, 'Malaysia': 55, 'Maldives': 55, 'Mali': 55, 'Malta': 55, 'Marshall Islands':
55, 'Mauritania': 55, 'Mauritius': 55, 'Mexico': 55, '"Micronesia': 55, 'Moldova': 55, 'Monaco': 55, 'Mongolia': 55, 'Montenegr
o': 55, 'Morocco': 55, 'Mozambique': 55, 'Myanmar': 55, 'Namibia': 55, 'Nepal': 55, 'Netherlands': 55, 'New Caledonia': 55, 'New
Zealand': 55, 'Nicaragua': 55, 'Niger': 55, 'Nigeria': 55, 'Northern Mariana Islands': 55, 'Norway': 55, 'Oman': 55, 'Pakistan':
55, 'Palau': 55, 'Panama': 55, 'Papua New Guinea': 55, 'Paraguay': 55, 'Peru': 55, 'Philippines': 55, 'Poland': 55, 'Portugal': 5
5, 'Puerto Rico': 55, 'Qatar': 55, 'Romania': 55, 'Russian Federation': 55, 'Rwanda': 55, 'Samoa': 55, 'San Marino': 55, 'Sao Tom
e and Principe': 55, 'Saudi Arabia': 55, 'Senegal': 55, 'Seychelles': 55, 'Sierra Leone': 55, 'Singapore': 55, 'Slovak Republic':
55, 'Slovenia': 55, 'Solomon Islands': 55, 'Somalia': 55, 'South Africa': 55, 'South Sudan': 55, 'Spain': 55, 'Sri Lanka': 55, 'S
t. Kitts and Nevis': 55, 'St. Lucia': 55, 'St. Vincent and the Grenadines': 55, 'Sudan': 55, 'Suriname': 55, 'Swaziland': 55, 'Sw
eden': 55, 'Switzerland': 55, 'Syrian Arab Republic': 55, 'Tajikistan': 55, 'Tanzania': 55, 'Thailand': 55, 'Timor-Leste': 55, 'T
ogo': 55, 'Tonga': 55, 'Trinidad and Tobago': 55, 'Tunisia': 55, 'Turkey': 55, 'Turkmenistan': 55, 'Turks and Caicos Islands': 5
5, 'Tuvalu': 55, 'Uganda': 55, 'Ukraine': 55, 'United Arab Emirates': 55, 'United Kingdom': 55, 'United States': 55, 'Uruguay': 5
5, 'Uzbekistan': 55, 'Vanuatu': 55, '"Venezuela': 55, 'Vietnam': 55, 'Virgin Islands (U.S.)': 55, '"Yemen': 55, 'Zambia': 55, 'Zi
mbabwe': 55, 'Serbia': 25, 'West Bank and Gaza': 25, 'Sint Maarten (Dutch part)': 17}

Writing an iterator to load data in chunks (1)

Another way to read data too large to store in memory in chunks is to read the file in as DataFrames of a certain length, say, 100. For example, with the pandas package (imported as pd), you can do pd.read_csv(filename, chunksize=100). This creates an iterable reader object, which means that you can use next() on it.

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

In [ ]:

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

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

```
                                CountryName CountryCode  Year  \
0                                Arab World         ARB  1960
1                     Caribbean small states         CSS  1960
2              Central Europe and the Baltics         CEB  1960
3    East Asia & Pacific (all income levels)         EAS  1960
4      East Asia & Pacific (developing only)         EAP  1960
5                                  Euro area         EMU  1960
6  Europe & Central Asia (all income levels)         ECS  1960
7    Europe & Central Asia (developing only)         ECA  1960
8                              European Union         EUU  1960
9   Fragile and conflict affected situations         FCS  1960

   Total Population  Urban population (% of total)
0      9.249590e+07                      31.285384
1      4.190810e+06                      31.597490
2      9.140158e+07                      44.507921
3      1.042475e+09                      22.471132
4      8.964930e+08                      16.917679
5      2.653965e+08                      62.096947
6      6.674890e+08                      55.378977
7      1.553174e+08                      38.066129
8      4.094985e+08                      61.212898
9      1.203546e+08                      17.891972
                                    CountryName CountryCode  Year  \
10         Heavily indebted poor countries (HIPC)         HPC  1960
11                                    High income         HIC  1960
12                           High income: nonOECD         NOC  1960
13                              High income: OECD         OEC  1960
14  Latin America & Caribbean (all income levels)         LCN  1960
15    Latin America & Caribbean (developing only)         LAC  1960
16   Least developed countries: UN classification         LDC  1960
17                             Low & middle income         LMY  1960
18                                      Low income         LIC  1960
19                             Lower middle income         LMC  1960

    Total Population  Urban population (% of total)
10      1.624912e+08                      12.236046
11      9.075975e+08                      62.680332
12      1.866767e+08                      56.107863
13      7.209208e+08                      64.285435
14      2.205642e+08                      49.284688
15      1.776822e+08                      44.863308
16      2.410728e+08                       9.616261
17      2.127373e+09                      21.272894
18      1.571884e+08                      11.498396
19      9.429116e+08                      19.810513
```

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.

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

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

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

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

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

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

# Print pops_list
print(pops_list)
```

```
                            CountryName CountryCode  Year  \
0                            Arab World         ARB  1960
1                  Caribbean small states         CSS  1960
2           Central Europe and the Baltics         CEB  1960
3  East Asia & Pacific (all income levels)         EAS  1960
4    East Asia & Pacific (developing only)         EAP  1960

   Total Population  Urban population (% of total)
0      9.249590e+07                      31.285384
1      4.190810e+06                      31.597490
2      9.140158e+07                      44.507921
3      1.042475e+09                      22.471132
4      8.964930e+08                      16.917679
[(91401583.0, 44.5079211390026), (92237118.0, 45.206665319194), (93014890.0, 45.866564696018), (93845749.0, 46.5340927663649), (9
4722599.0, 47.2087429803526)]
```

Writing an iterator to load data in chunks (3)

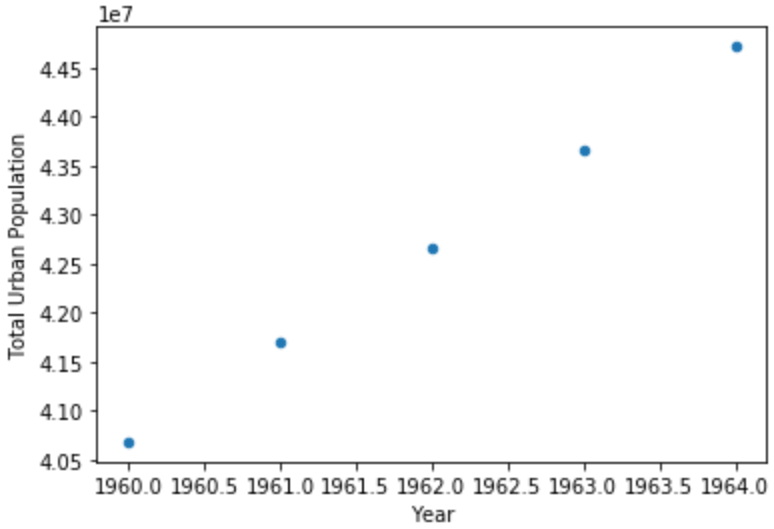
You're getting used to reading and processing data in chunks by now. Let's push your skills a little further by adding a column to a DataFrame.

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.

In [ ]:
```
import warnings
warnings.filterwarnings('ignore')
```

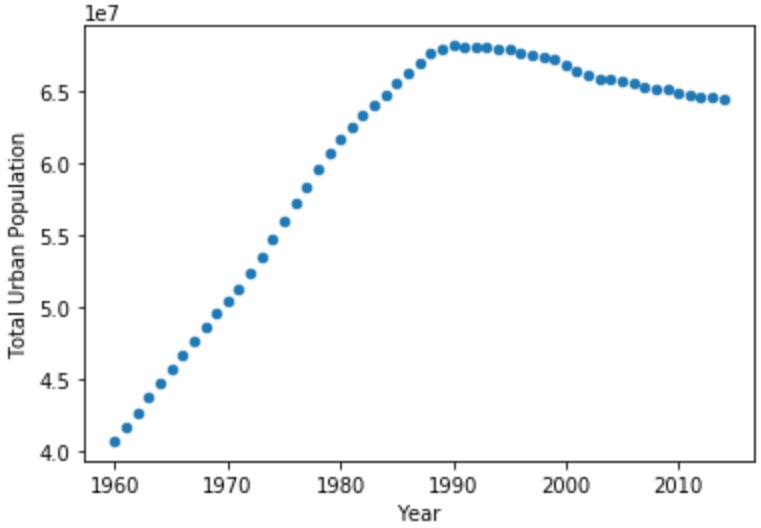
In [ ]:
```
# Code from previous exercise
urb_pop_reader = pd.read_csv('world_ind_pop_data.csv', chunksize=1000)
df_urb_pop = next(urb_pop_reader)
df_pop_ceb = df_urb_pop[df_urb_pop['CountryCode'] == 'CEB']
pops = zip(df_pop_ceb['Total Population'],
           df_pop_ceb['Urban population (% of total)'])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

# Use list comprehension to create new DataFrame column 'Total Urban Population'
df_pop_ceb['Total Urban Population'] = [int(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()
```



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!

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

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

# Iterate over each DataFrame chunk
for df_urb_pop in urb_pop_reader:

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

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

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

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

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

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



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:

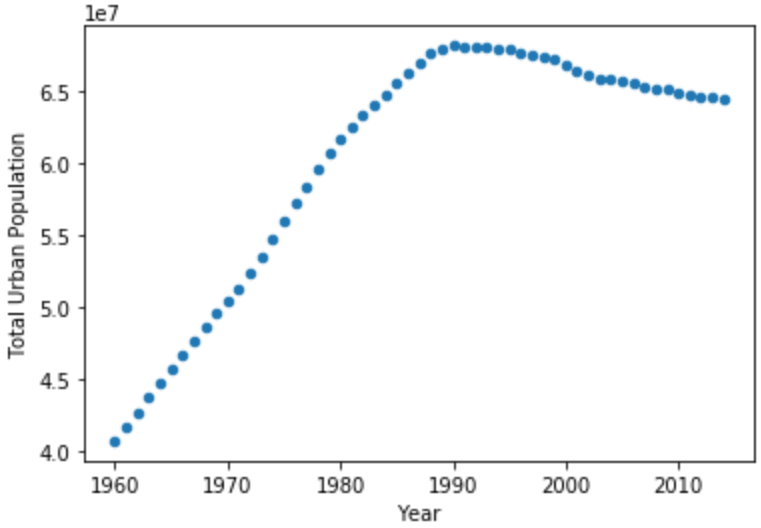
- Loading of the file chunk by chunk,
- Creating the new column of urban population values, and
- Plotting the urban population data.

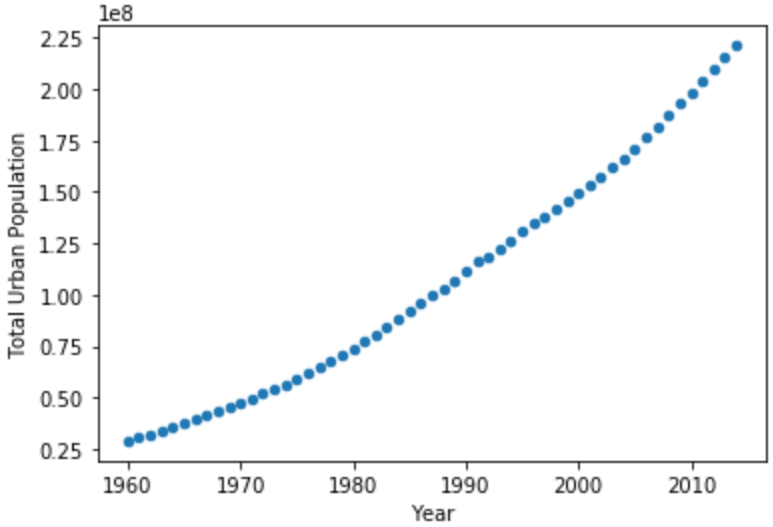
That's a lot of work, but the function now makes it convenient to repeat the same process for whatever file and country code you want to process and visualize!.

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.

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

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

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

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

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

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

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

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

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

# Set the filename: fn
fn = 'world_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')
```


1e7
6.5
6.0
5.5
5.0
4.5
4.0
Total Urban Population
1960
1970
1980
1990
2000
2010
Year


1e8
2.25
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
Total Urban Population
1960
1970
1980
1990
2000
2010
Year