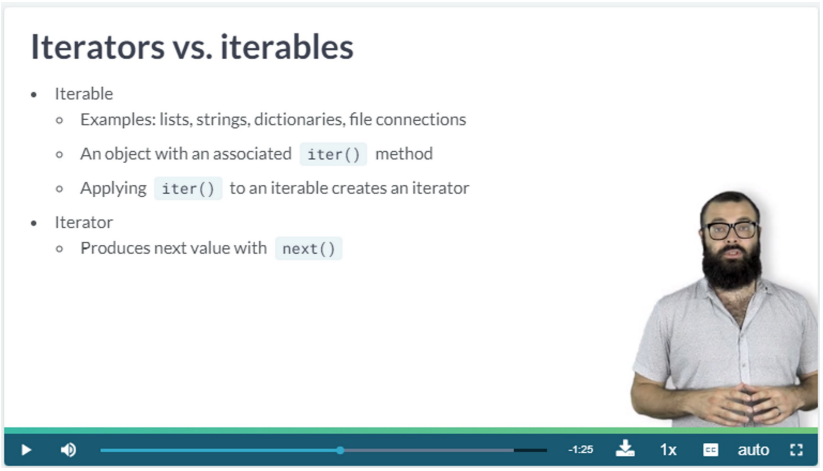
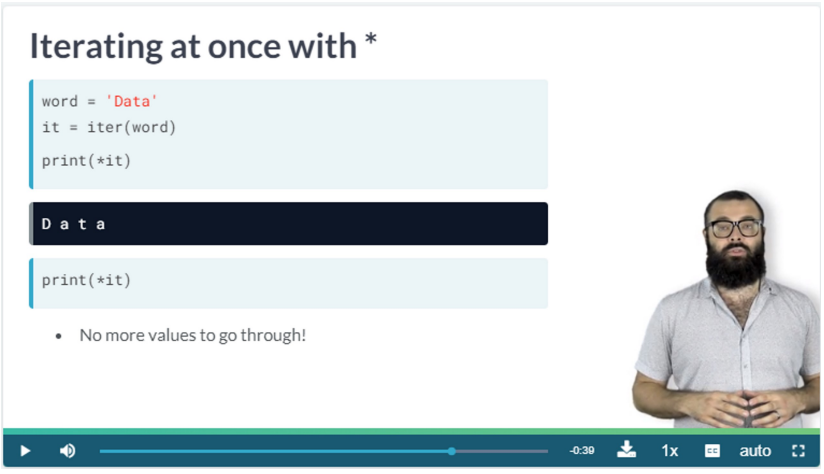
04. Python Data Science Toolbox (Part 2)

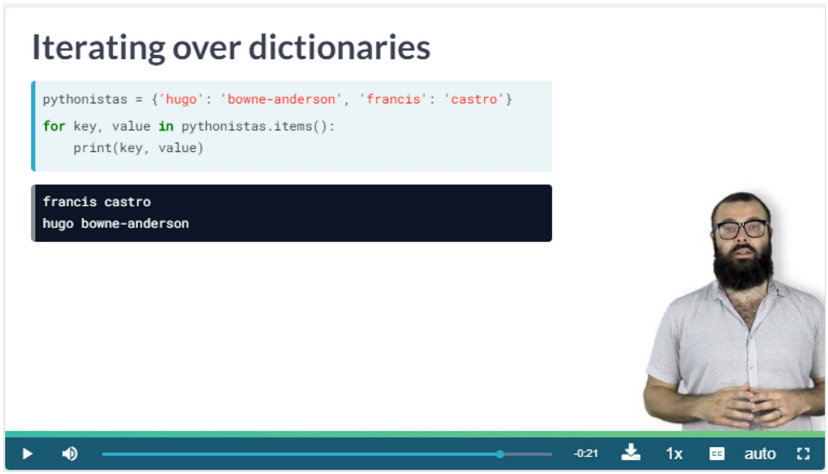
07 January 2020 16:58

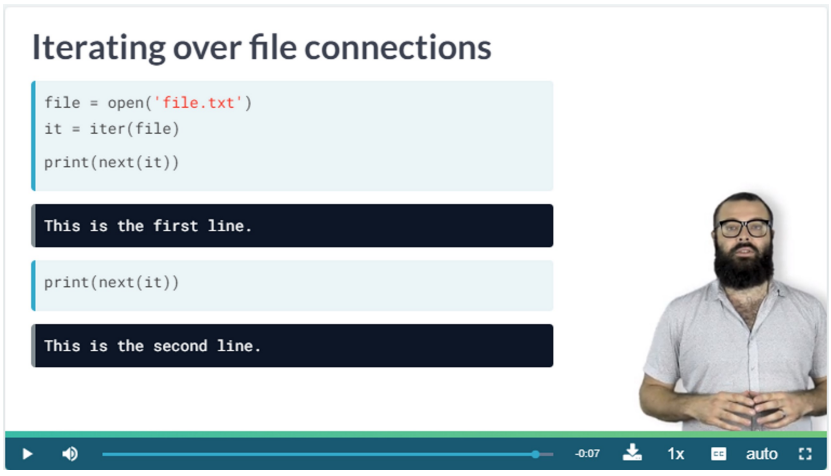
1. Using iterators in PythonLand

1.1 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*.

Possible Answers

- [ ] Both flash1 and flash2 are iterators.
- [ ] Both flash1 and flash2 are iterables.
- [x] flash1 is an iterable and flash2 is an iterator.

1.2 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:

- Create a for loop to loop over flash and print the values in the list. Use person as the loop variable.
- Create an *iterator* for the list flash and assign the result to superhero.
- Print each of the items from superhero using next() 4 times.

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

# Print each list item in flash using a for loop
```

```
for person in flash :
    print(person)

# 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))
```

1.3 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 $10^{100}10100$ may not work, especially since a number as big as that may go over a regular computer's memory. The value $10^{100}10100$ 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 $10^{100}10100$ won't actually pre-create the list.

Instructions:

- Create an **iterator** object small_value over range(3) using the function iter().
- Using a for loop, iterate over range(3), printing the value for every iteration. Use num as the loop variable.
- Create an **iterator** object googol over range(10 ** 100).

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

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

1.4 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.

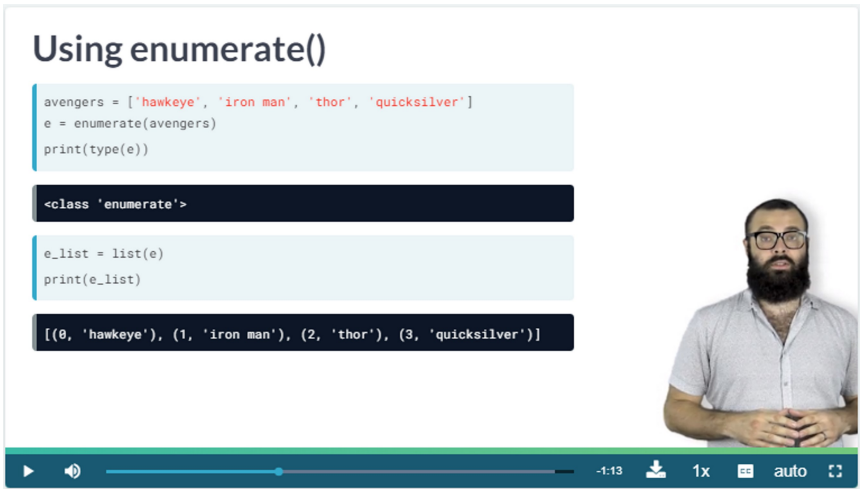
In this exercise, you will use these functions by passing an iterable from range() and then printing the results of the function calls.

Instructions:

- Create a range object that would produce the values from 10 to 20 using range(). Assign the result to values.
- Use the list() function to create a list of values from the range object values. Assign the result to values_list.
- Use the sum() function to get the sum of the values from 10 to 20 from the range object values. Assign the result to values_sum.

```
# Create a range object: values
values = range(10, 21)

# Print the range object
print(values)

# 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)
```

1.5 Using enumerate



enumerate() and unpack

```
avengers = ['hawkeye', 'iron man', 'thor', 'quicksilver']
for index, value in enumerate(avengers):
    print(index, value)
```

```
0 hawkeye
1 iron man
2 thor
3 quicksilver
```

```
for index, value in enumerate(avengers, start=10):
    print(index, value)
```

```
10 hawkeye
11 iron man
12 thor
13 quicksilver
```

-0:51 1x auto

Using zip()

```
avengers = ['hawkeye', 'iron man', 'thor', 'quicksilver']
names = ['barton', 'stark', 'odinson', 'maximoff']
z = zip(avengers, names)
print(type(z))
```

```
<class 'zip'>
```

```
z_list = list(z)
print(z_list)
```

```
[('hawkeye', 'barton'), ('iron man', 'stark'),
('thor', 'odinson'), ('quicksilver', 'maximoff')]
```

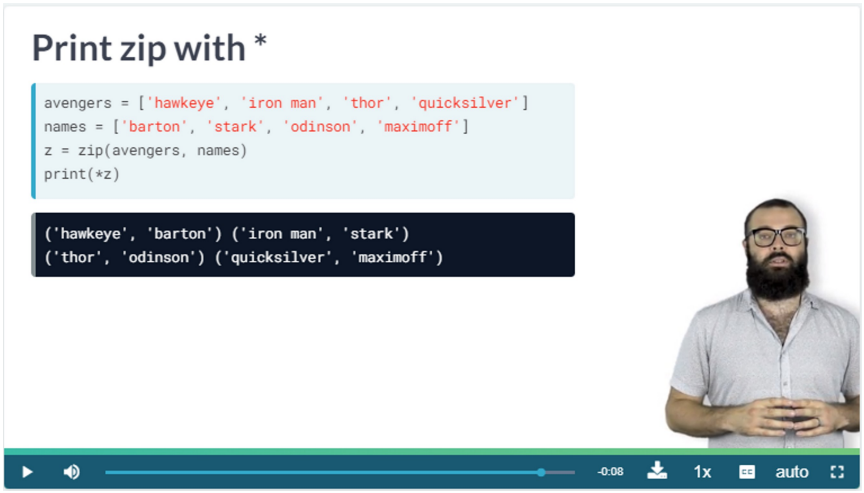
-0:22 1x auto

zip() and unpack

```
avengers = ['hawkeye', 'iron man', 'thor', 'quicksilver']
names = ['barton', 'stark', 'odinson', 'maximoff']
for z1, z2 in zip(avengers, names):
    print(z1, z2)
```

```
hawkeye barton
iron man stark
thor odinson
quicksilver maximoff
```

-0:12 1x auto



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:

- Create a list of tuples from mutants and assign the result to mutant_list. Make sure you generate the tuples using enumerate() and turn the result from it into a list using list().
- Complete the first for loop by unpacking the tuples generated by calling enumerate() on mutants. Use index1 for the index and value1 for the value when unpacking the tuple.
- Complete the second for loop similarly as with the first, but this time change the starting index to start from 1 by passing it in as an argument to the start parameter of enumerate(). Use index2 for the index and value2 for the value when unpacking the tuple.

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

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

# Print the list of tuples
print(mutant_list)

# 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)
```

1.6 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:

- 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 = 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)
```

*1.7 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.

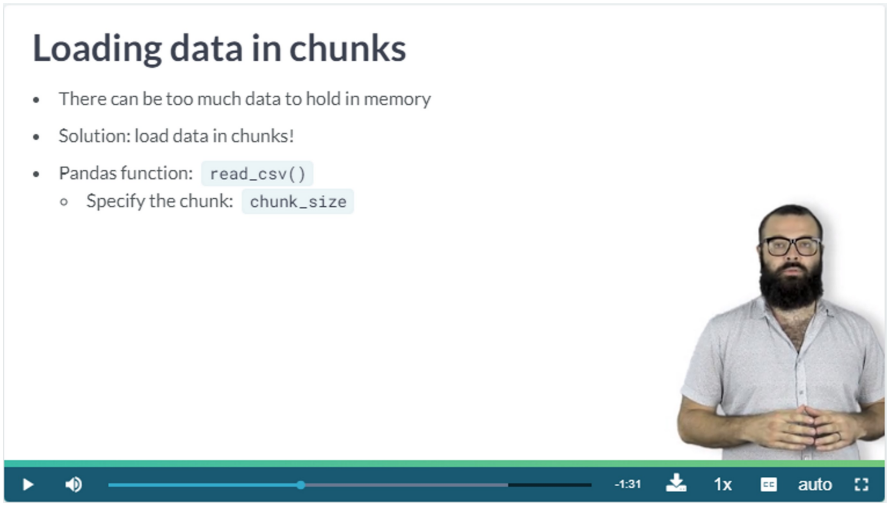
Instructions:

- Create a zip object by using zip() on mutants and powers, in that order. Assign the result to z1.
- Print the tuples in z1 by unpacking them into positional arguments using the * operator in a print() call.
- Because the previous print() call would have exhausted the elements in z1, recreate the zip object you defined earlier and assign the result again to z1.
- 'Unzip' the tuples in z1 by unpacking them into positional arguments using the * operator in a zip() call. Assign the results to result1 and result2, in that order.
- The last print() statements prints the output of comparing result1 to mutants and result2 to powers. Click Submit Answer to see if the unpacked result1 and result2 are equivalent to mutants and powers, respectively.

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

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

# 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)
```

1.8 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.

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** 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:

- 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
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)
```

1.9 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:

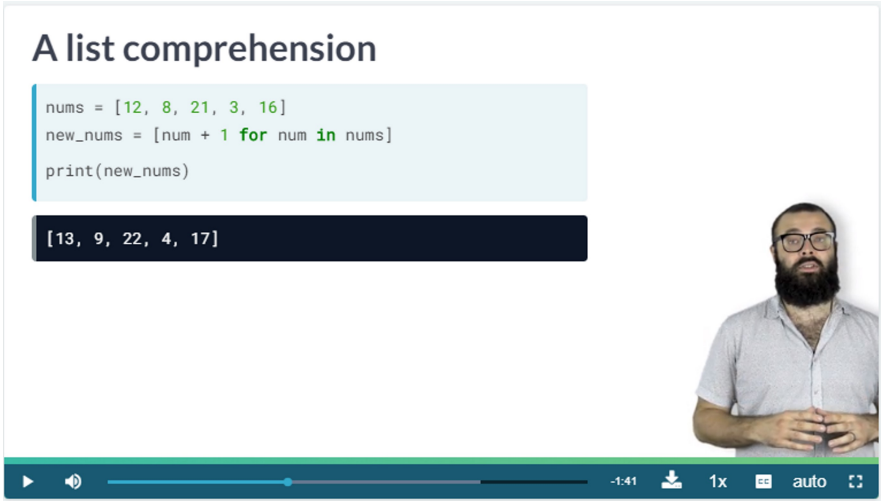
- Define the function count_entries(), which has 3 parameters. The first parameter is csv_file for the filename, the second is c_size for the chunk size, and the last is colname for the column name.
- Iterate over the file in csv_file file by using a for loop. Use the loop variable chunk and iterate over the call to pd.read_csv(), passing c_size to chunksize.
- In the inner loop, iterate over the column given by colname in chunk by using a for loop. Use the loop variable entry.
- Call the count_entries() function by passing to it the filename 'tweets.csv', the size of chunks 10, and the name of the column to count, 'lang'. Assign the result of the call to the variable result_counts.

```
# Define count_entries()
def count_entries(csv_file, c_size, colname):
    """Return a dictionary with counts of
    occurrences as value for each key."""

    # Initialize an empty dictionary: counts_dict
    counts_dict = {}
```

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

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

    # Return counts_dict
    return counts_dict

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

# Print result_counts
print(result_counts)
```

2. List comprehensions and generators

2.1 Write a basic list comprehension



For loop and list comprehension syntax

```
new_nums = [num + 1 for num in nums]
```

```
for num in nums:
    new_nums.append(num + 1)
print(new_nums)
```

```
[13, 9, 22, 4, 17]
```

-1:25 1x auto

List comprehension with range()

```
result = [num for num in range(11)]
print(result)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

-1:21 1x auto

List comprehensions

- Collapse for loops for building lists into a single line
- Components
  - Iterable
  - Iterator variable (represent members of iterable)
  - Output expression

-0:59 1x auto

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?

Possible Answers

- [ ] The list comprehension is `[for doc in doctor: doc[0]]` and produces the list `['h', 'c', 'c', 't', 'w']`.
- [x] 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]: doctor = ['house', 'cuddy', 'chase', 'thirteen', 'wilson']

In [2]: [doc[0] for doc in doctor]
Out[2]: ['h', 'c', 'c', 't', 'w']
```

2.2 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

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).
- [x] You can build list comprehensions over all the objects except the integer object valjean.

2.3 Writing list comprehensions

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

Instructions:

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

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

2.4 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*]

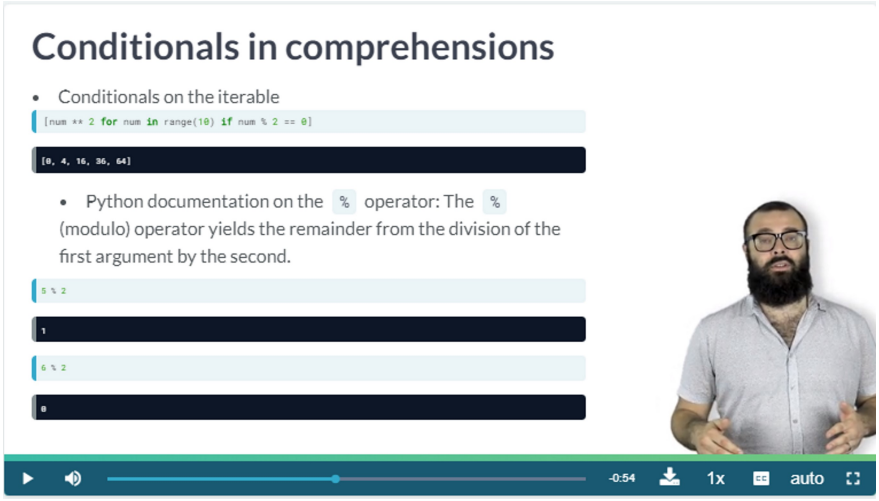
Note that here, the **output expression** is itself a list comprehension.

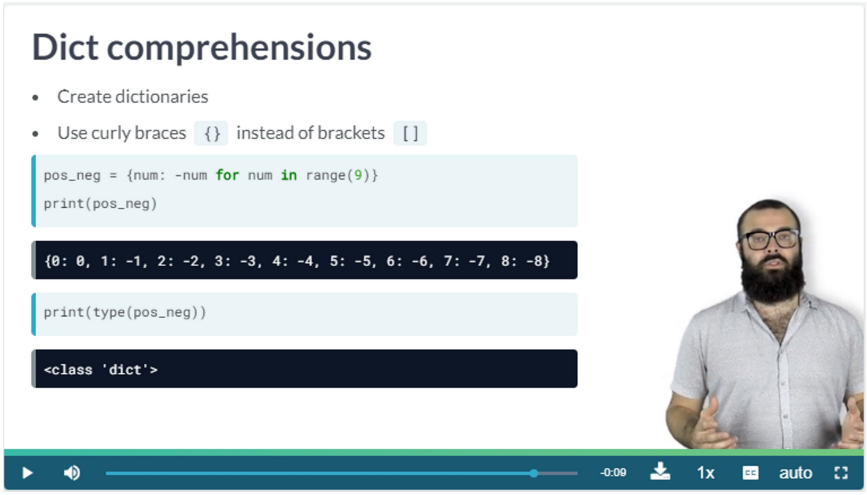
Instructions:

- 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 to create values in the list of lists.

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

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

2.5 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:

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

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

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

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

2.6 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:

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

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

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

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

2.7 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:

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

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

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

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

2.8 List comprehensions vs generators

Generator expressions

- Recall list comprehension

```
[2 * num for num in range(10)]
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

  - Use `( )` instead of `[ ]`

```
(2 * num for num in range(10))
```

```
<generator object <genexpr> at 0x1046bf888>
```

-3:23 1x auto

List comprehensions vs. generators

- List comprehension - returns a list
- Generators - returns a generator object
- Both can be iterated over

-3:07 1x auto

Printing values from generators (1)

```
result = (num for num in range(6))
for num in result:
    print(num)
```

```
0
1
2
3
4
5
```

```
result = (num for num in range(6))
print(list(result))
```

```
[0, 1, 2, 3, 4, 5]
```

-2:53 1x auto

Printing values from generators (2)

```
result = (num for num in range(6))
```

- Lazy evaluation

```
print(next(result))
```

```
0
```

```
print(next(result))
```

```
1
```

```
print(next(result))
```

```
2
```

```
print(next(result))
```

```
3
```

```
print(next(result))
```

```
4
```

-2:25 1x auto

Generators vs list comprehensions

IPython Shell

```
In [1]: [num for num in range(10**1000000)]

In [2]:
```

IPython Shell

```
In [1]: [num for num in range(10**1000000)]

In [2]:
```

Your session has been disconnected.
The performed operation was too resource-intensive.

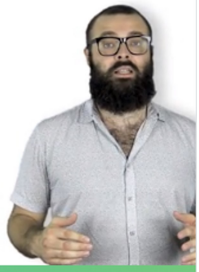
Restart Session

-1:56 1x auto

Generators vs list comprehensions

IPython Shell

```
In [1]: (num for num in range(10**1000000))
Out[1]: <generator object <genexpr> at 0x7f8aca2601f8>

In [2]:
```



-1:44 1x auto

Conditionals in generator expressions

```
even_nums = (num for num in range(10) if num % 2 == 0)
print(list(even_nums))
```

```
[0, 2, 4, 6, 8]
```

-1:29 1x auto

Generator functions

- Produces generator objects when called
- Defined like a regular function - `def`
- Yields a sequence of values instead of returning a single value
- Generates a value with `yield` keyword

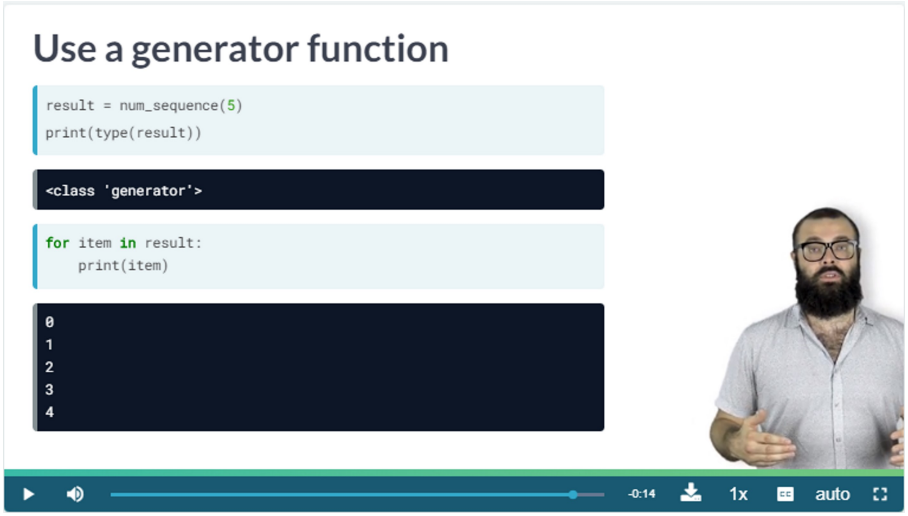
-1:04 1x auto

Build a generator function

- sequence.py

```
def num_sequence(n):
    """Generate values from 0 to n."""
    i = 0
    while i < n:
        yield i
        i += 1
```

-0:38 1x auto



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:

- [ ] List comprehensions and generators are not different at all; they are just different ways of writing the same thing.
- [x] 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.

2.9 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:

- Create a generator object that will produce values from 0 to 30. Assign the result to result and use num as the iterator variable in the generator expression.
- Print the first 5 values by using next() appropriately in print().
- Print the rest of the values by using a for loop to iterate over the generator object.

```
# Create generator object: result
result = (num for num in range(31))
```

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

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

2.10 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:

- Write a generator expression that will generate the **lengths** of each string in `lannister`. Use `person` as the iterator variable. Assign the result to `lengths`.
- Supply the correct iterable in the `for` loop for printing the values in the generator object.

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

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

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

2.11 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)`

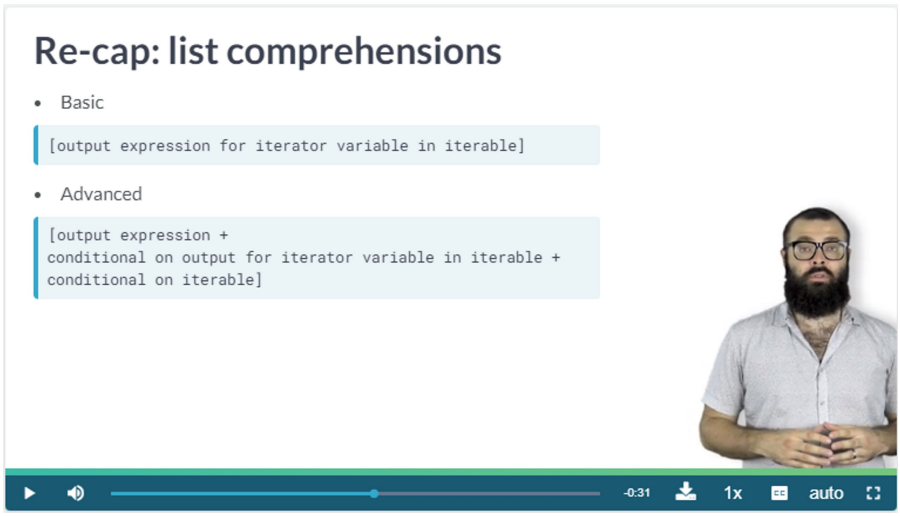
Instructions:

- Complete the function header for the function `get_lengths()` that has a single parameter, `input_list`.
- In the `for` loop in the function definition, `yield` the *length* of the strings in `input_list`.
- Complete the iterable part of the `for` loop for printing the values generated by the `get_lengths()` generator function. Supply the call to `get_lengths()`, passing in the list `lannister`.

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

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

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

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

2.12 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:

- 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 = 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)
```

2.13 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.

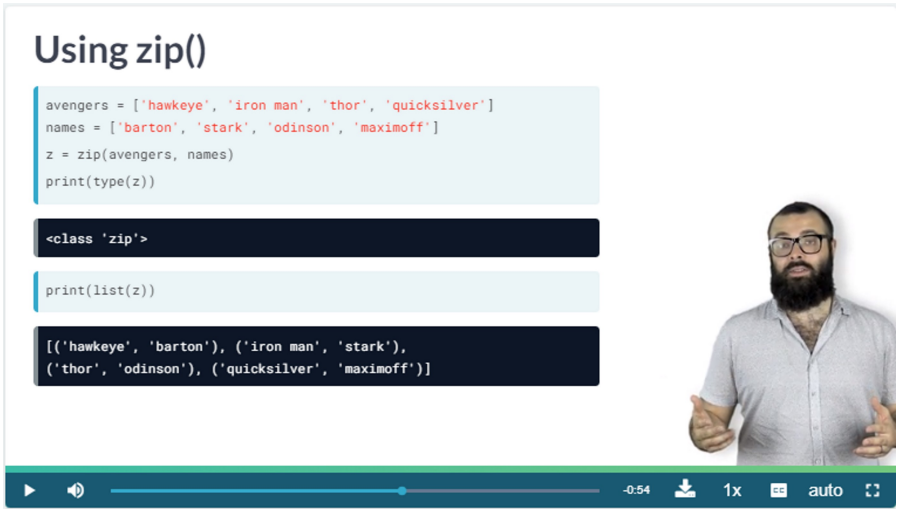
Instructions:

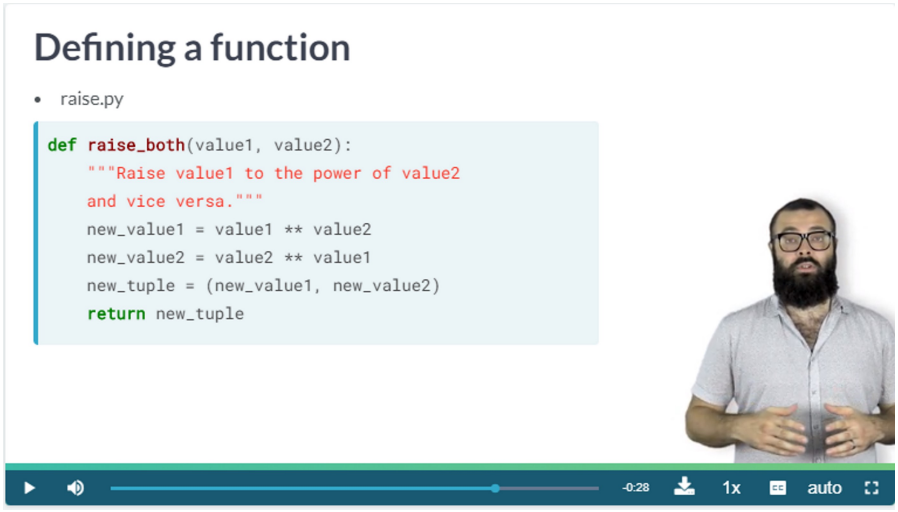
- Extract the column 'created_at' from df and assign the result to tweet_time.
- Create a list comprehension that extracts the time from each row in tweet_time. Each row is a string that represents a timestamp, and you will access the *12th to 19th characters* in the string to extract the time. Use entry as the *iterator variable* and assign the result to tweet_clock_time. Additionally, add a conditional expression that checks whether entry[17:19] is equal to '19'.

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

# Extract the clock time: tweet_clock_time
tweet_clock_time = [entry[11:19] for entry in tweet_time if entry[17:19] == '19']

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

3. Bringing it all together!

3.1 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.

Instructions:

- 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 = zip(feature_names, row_vals)

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

# Print the dictionary
print(rs_dict)
```

3.2 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:

- Define the function lists2dict() with two parameters: first is list1 and second is list2.
- Return the resulting dictionary rs_dict in lists2dict().
- Call the lists2dict() function with the arguments feature_names and row_vals. Assign the result of the function call to rs_fxn.

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

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

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

    # Return the dictionary
    return(rs_dict)

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

# Print rs_fxn
print(rs_fxn)
```

3.3 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:

- 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(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])
```

3.4 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.

The lists2dict() function, feature_names list, and row_lists list have been preloaded for this exercise.

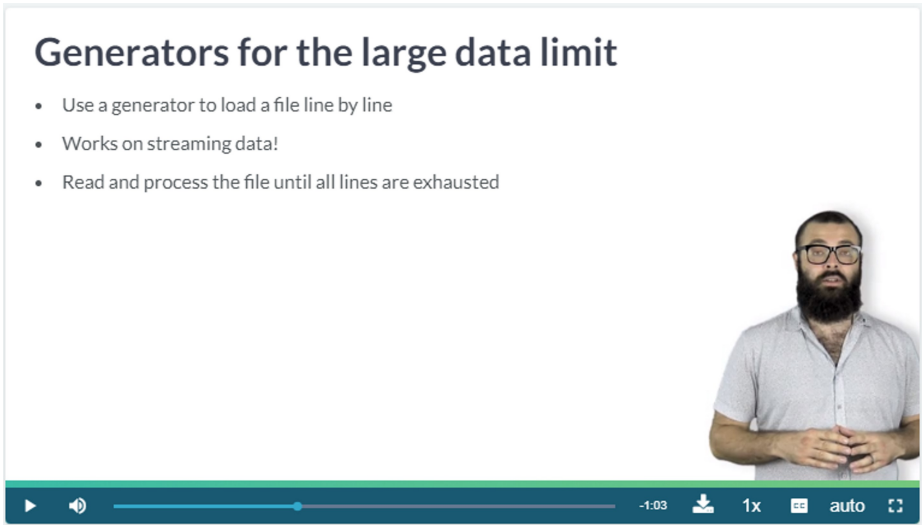
Go for it!

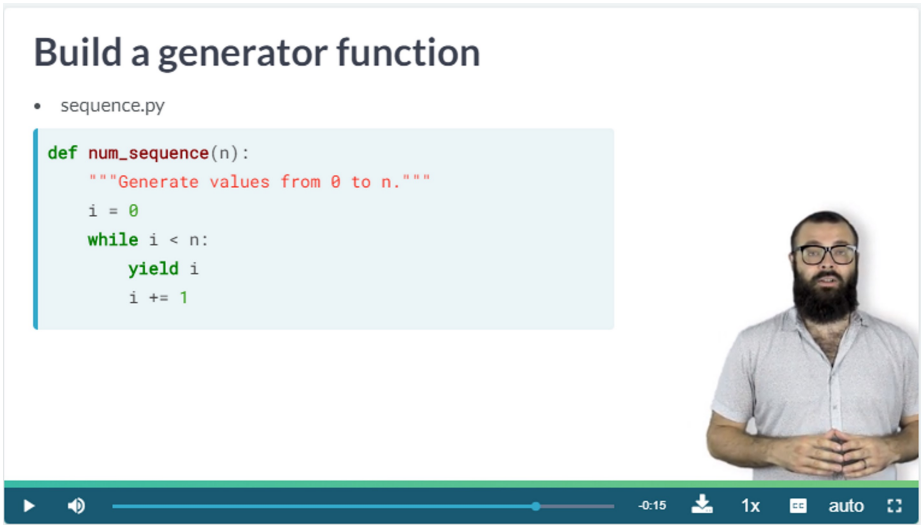
Instructions:

- To use the DataFrame() function you need, first import the pandas package with the alias pd.
- Create a DataFrame from the list of dictionaries in list_of_dicts by calling pd.DataFrame(). Assign the resulting DataFrame to df.
- Inspect the contents of df printing the head of the DataFrame. Head of the DataFrame df can be accessed by calling df.head().

```
# Import the pandas package
import pandas as pd

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

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

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

3.5 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.**

Instructions:

- 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(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)
```

3.6 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_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:

- In the function `read_large_file()`, read a line from `file_object` by using the method `readline()`. Assign the result to `data`.
- In the function `read_large_file()`, `yield` the line read from the file `data`.
- In the context manager, create a generator object `gen_file` by calling your generator function `read_large_file()` and passing `file` to it.
- Print the first three lines produced by the generator object `gen_file` using `next()`.

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

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

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

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

        # Yield the line of data
        yield data

# Open a connection to the file
with open('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))
```

3.7 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:

- Bind the file 'world_dev_ind.csv' to file in the context manager with open().
- Complete the for loop so that it iterates over the generator from the call to read_large_file() to process all the rows of the file.

```
# Initialize an empty dictionary: counts_dict
counts_dict = {}

# Open a connection to the file
with open('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)
```

3.8 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.

Instructions:

- Use pd.read_csv() to read in 'ind_pop.csv' in chunks of size 10. Assign the result to df_reader.
- Print the first two chunks from df_reader.

```
# Import the pandas package
import pandas as pd

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

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

```
print(next(df_reader))
```

3.9 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:

- 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('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)
```

3.10 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:

- 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'] = [int(pops_list[i][0] * pops_list[i][1] * 0.01) for i in

# df_pop_ceb['Total Urban Population'] = [int(tup[0] * tup[1] * 0.01) for tup in pops_list]

range(len(pops_list))]

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

3.11 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:

- Initialize an empty DataFrame data using pd.DataFrame().
- In the for loop, iterate over urb_pop_reader to be able to process all the DataFrame chunks in the dataset.
- Using the method append() of the DataFrame data, append df_pop_ceb to data.

```
# Initialize reader object: urb_pop_reader
urb_pop_reader = pd.read_csv('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()
```

3.12 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:

- 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 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 = 'ind_pop_data.csv'

# Call plot_pop for country code 'CEB'
plot_pop(filename = fn, country_code = 'CEB')

# Call plot_pop for country code 'ARB'
plot_pop(filename = fn, country_code = 'ARB')
```