UMpy: map() and filter()

Goals

1. Awareness/Literacy: work with functions that accept another callable as an argument.
2. Awareness/Literacy: modify sequences and dictionaries using the built-in `map()` function.
3. Awareness/Literacy: return subsets of sequences using the built-in `filter()` function.
4. Awareness: work with built-in `zip()` function.
5. Awareness: work with the `statistics` and `string` modules.
6. Review: Work with list and dictionary comprehensions.
7. Review: Work with `lambda` functions.

Glossary

source: https://docs.python.org/3/glossary.html

1. **iterable**: an object capable of returning its members one at a time.

2. **iterator**: an object representing a stream of data. An iterator is provisioned with a `__next__()` method that can be called to iterate over the object.

   💡 An iterator is also an iterable but not all iterables are iterators (e.g., a `list`). Both the built-in functions `map()` and `filter()` return iterator objects.

map()

The built-in `map()` function applies a specified function to each element in of a passed in iterable or list of iterables and returns an iterator (type `map`) that *yields* each transformed element on demand.

```
map(< function >, iterable[, iterable_1 ... iterable_N])
```

💡 The `map()` function is often paired with the built-in `list()` function in order to return to the caller a list of transformed elements.

```
var = list(map(< function >, iterable[, iterable_1 ... iterable_N]))
```

The `map()` function leverages a "callable" (e.g., a function) passed to it in order to *transform* sequence elements or dictionary values, *mapping* each source value to a new value in the iterator that it returns.

filter()

The built-in `filter()` function returns an iterator comprising those elements of a passed in iterable that meet the condition or conditions imposed by the specified function (i.e., returns `True`).

```
filter(< function >, iterable)
```

💡 The `filter()` function is often paired with the built-in `list()` function in order to return to the caller a list of transformed elements.

```
var = list(filter(< function >, iterable))
```

The `filter()` function utilizes a "callable" (e.g., a function) passed to it in order to apply a filtering condition or conditions against an iterable in order to return an iterator comprising a subset of the iterable's elements.

Challenges

Challenge 01

South African life expectancy data reveals a slow but steady increase in life expectancy for both females and males. Let's clean the values using `map()`.

1. Open the file `south_africa-life_expectancy-1960_2019.csv` containing South African life expectancy data sourced from the World Bank and assign the list returned to a variable named `data`.

2. Access the headers, female life expectancy numbers (by year), and male life expectancy numbers (by year) and assign to the variables `headers`, `female_life_exp`, and `male_life_exp`.

3. Utilize `map()` to convert the elements in `female_life_exp` from `str` to `float` and assign to a new list named `female_life_exp_flt`.

4. Bonus: use the built-in functions `dict()` and `zip()` to create a dictionary using `headers` as the keys and `female_life_exp` as the values, converting each value using `map()` to a `float`.

5. Bonus: utilize a dictionary comprehension to return a dictionary with `headers` as the keys and `male_life_exp` as the values, with each value converted from a `str` to a `float`. Assign the new dictionary to a variable named `male_life_exp_flt`.

   💡 employ `len` and `range()` to sync the `headers` and `male_life_exp` indexes.

Challenge 02

Cape Town's winter season is June to August. Let's work with Cape Town temperature data for June 2021 using `map()` to convert the Fahrenheit values to Celsius.

1. Open the file `cape_town-temperature_readings-202106.csv` containingCape Town temperature data for June 2021 and assign to a variable named `data`.

2. Access the headers, daily max temperatures, and daily min temperatures (and assign to the variables `headers`, `temp_max`, and `temp_min`.

3. Using the `statistics` module compute the mean (average) max temperature for June. Employ `map()` to convert the temperature values from `str` to `int`. Assign the return value to a variable named `mean_max_temp_fahr`.

4. Again, use the `statistics` module to compute the mean (average) max temperature for June. But this time convert the Fahrenheit values to Celsuis using `map()` and a passed in `lambda` function, rounding to the third decimal place. Assign the return value to `mean_max_temp_cels`.

   💡 Formula: `< Celsius value > = (< Fahrenheit value > - 32) * .5556`

5. Convert all the max temperature Fahrenheit values to Celsius using `map()`. Assign the iterator returned to a new list named `temp_max_cels`.

6. Convert all the min temperature Fahrenheit values to Celsius using a list comprehension. Assign the new list to a variable named `temp_min_cels`.

7. Process both the max and min temperature values together, converting each from Fahrenheit to Celsius. Use a list comprehension to create the list of min and max values and assign it to a variable named `temp_max_min`. Then write another list comprehension that loops over `temp_max_min` and converts the nested list elements using either a `map()` and `lambda` function or another approach. Assign the new list to a variable named `temp_cels`.

8. Utilize a list comprehension to create a new list of city and date values based on `data`. Assign the new list to a variable named `city_days`. "Rejoin" `city_days` and `temp_cels` employing comprehension that utilizes `len()` and `range()` to sync the indexes.

Challenge 03

1. Employ the built-in function `filter()` and a `lambda` function to return an iterator that can be converted to a `list` comprising only elements from `temp_max` with a value >= 70 degrees Fahrenheit. Assign the new list to a variable named `high_temps`.

2. Use `filter()` and the custom function `is_temp_extreme` to return an iterator that can be converted to a `list` comprising only elements from `temp_max_min` with a max temperature value >= 70 degrees Fahrenheit *and* min temperature value <= 50 degrees Fahrenheit. Assign the new list to a variable named `extreme_temps`.

3. Bonus: use `filter()` and the custom function `is_temp_extreme` to return an iterator that can be converted to a `list` comprising only elements from `temp_max_min` with a max temperature value >= 68 degrees Fahrenheit *and* min temperature value <= 48 degrees Fahrenheit. Assign the new list to a variable named `extreme_temps`.

   💡 this problem requires use of a `lambda` function to pass the needed arguments to the function `is_temp_extreme`.

Challenge 04

1. Open the file `mandela-prepared_speech.txt` containing the statement that Nelson Mandela read from the defendant's dock on 20 April 1964 during the Rivonia Trial (Oct 1963 - June 1964). His

statement has come to known as the "I am prepared to die" speech. Skip all blank lines encountered and return a list of paragraphs assigned to a variable named `data_loop`.

2. Two alternative implementations using 1) `list(map(...))` and 2) a list comprehension will also be shared.

Challenge 05

1. Remove all punctuation from the text using `str.make_trans()` and the `string.punctuation` constant. Use a `for` loop and the accumulator pattern to accomplish the task. Assign the new list to a variable named `data_cleaned_loop`.

2. Two alternative implementations using 1) `list(map(...))` and 2) a list comprehension will also be shared.

Challenge 06

1. Utilize `filter()` to search `data_cleaned_loop` for specific words and phrases (one word or phrase at a time). Return the list of matched elements (text) to a variable named `lines`.

   Search terms: 'apartheid', 'white supremacy', 'communist', 'freedom charter'

2. Implement using `filter()` and a `lambda` function.

3. Implement using a list comprehension.

Sources

Nelson Mandela, "I am prepared to die", prepared speech, 20 April 1964.

Weather Underground, Cape Town, South Africa, June 2021.

World Bank, South Africa.