**Top 3 Pythonic Thinking Tips for Python List Creation**

**Python List Derivation for Data Science**

[Effective Python](https://www.amazon.com/Effective-Python-Specific-Software-Development/dp/0134853989) is a book by Brett Slatkin that covers 59 specific ways to write better python. The book is written in a random-access fashion where each topic has self-contained source code. It is a great resource for intermediate python programmers, whether an engineer or data scientist, as it covers a wide range of topics that can be studied in any order.

Many of the topics covered are very applicable to data science workflows. For example, it covers pythonic thinking with PEP8 which is a style guide that ensures that your python code is readable. It covers best practices for functions, classes and meta classes all of which have important use cases in data science workflows. It also covers best approaches for writing readable comprehensions for list, tuples and dictionaries. This can be applied to task such as feature engineering, data preprocessing and data post processing.

Comprehensions are a useful way to derive one list from another list in a readable way. Effective python cover best practices for comprehensions (also applicable to tuples and dictionaries). It discourages the use of map and filter which can achieve the same tasks as a comprehension but with noisier and harder to read code. It also advises against using multiple expressions in a comprehension. Finally, it suggest using generators for tasks that require comprehensions on large amounts of data.

Writing effective python within a data science workflow can ensure code used for tasks such as feature engineering, data preprocessing and data post processing are efficient and easy to read. Efficiency and readability make it easier for data science and machine learning code bases to be maintained. The easier code is to read, the easier it is for changes to be made without creating bugs. Further, knowing the more efficient approach to performing a task, like list derivation, ultimately helps developers write more effective code.

Here we will see how we can incorporate three effective python practices into a simple data science workflow. We will look at how using list comprehension, over map and filter, can improve readability. We will also see limiting comprehensions to at most two expressions ensures code clarity. Finally, we will compare comprehension and generators when working with a large amount of data.

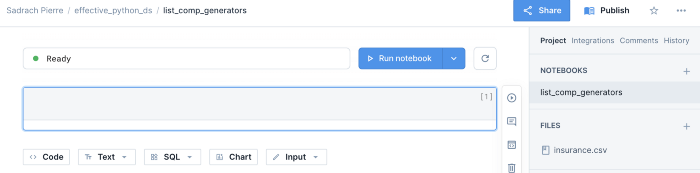
For this work, I will be writing code in [Deepnote](https://deepnote.com/), which is a collaborative data science notebook that makes running reproducible experiments very easy. We will be working with the [Medical Cost dataset](https://www.kaggle.com/datasets/mirichoi0218/insurance). The data is publicly free to use, modify and share under the [Database Contents License](https://opendatacommons.org/licenses/dbcl/1-0/) (DbCL: Public Domain).

**Use list comprehension instead of map & filter**

Map and filter are built in python functions that provide convenient short-hands to the tasks that can be achieved by list comprehensions. To demonstrate the difference between these techniques we will consider two common data tasks. Specifically, we will show how to use map to generate log transform of a column and then show how we can use list comprehension to complete the same task.

To start, let’s navigate to Deepnote and create a new project (you can sign-up for free if you don’t already have an account).

Let’s create a project called ‘effective\_python’ and a notebook within this project called ‘list\_comp\_generators’. Also, lets drag and drop the insurance.csv file on the left hand panel on the page where it says ‘FILES’:



Screenshot taken by Author

Next let’s import the pandas library and read our data into a pandas dataframe:

Embedding Created by Author

Next let’s display the first five rows of data:

Embedding Created by Author

*Using Map for Log-Transfrorm*

The column transformation we will consider first is the log transform. This is a common technique used to transform skewed data to approximately normal data. We can use the map() function to transform a list of numerical values in our data. Let’s use map to take the log transform of the BMI values. We will import the Numpy library and define a function that takes a list as input and returns a list of log transformed elements:

Embedding Created by Author

Next we can use map to apply our function to our BMI list. The map function takes the function name of the function we will be applying and an iterable (in our case a list):

output\_list = map(function, list)

We proceed by defining a variable called bmi\_list and storing the list of bmi values in our variable. We then can pass our function and list to the built-in map function and store the results in a new list that we will call bmi\_lt\_map. We then need to define a new column to store our transformed column:

Embedding Created by Author

Notice this took 5 lines of code to accomplish:

import numpy as np   
def log\_transform(input\_list):  
 return np.log(input\_list)  
bmi\_list = list(df['bmi'])  
bmi\_lt\_map = map(log\_transform, bmi\_list)  
df['bmi\_lt\_map'] = bmi\_lt\_map

*Using List Comprehension for Log-Transfrorm*

Let’s see how using list comprehension can help us achieve the same with more understandable code:

Embedding Created by Author

We see that we can do the same exact thing with only two lines of code. It is also much easier to read. It’s worth noting that apply the log transform to the dataframe column directly is also compact and easy to read, though this isn’t possible for more complex transformations.

Embedding Created by Author

**Avoid more than two expressions in list comprehension**

Let’s say we have a list of lists of prediction probabilities that are produced by a classification machine learning model:

Embedding Created by Author

We can convert this list of lists into a single list (we can “flatten” the list) using list comprehension:

Embedding Created by Author

This is a list comprehension containing two for-loops. This is an easy to read alternative to using two traditional for-loops:

Embedding Created by Author

The list comprehension approach takes up two lines of code while the tradition for-loop takes up 5 lines of code! Despite the convenience of list comprehension, readability and compactness quickly diminishes with more than two expressions. Suppose we want to generate a list of lists with a label ‘Yes’ for probabilities greater than 0.8., ‘Maybe’ for probabilities between 0.5 and 0.8, and ‘No’ for probabilities less than 0.5. We can use list comprehension to do this:

Embedding Created by Author

While this is compact it is pretty difficult to read and understand. If we use traditional for-loops, while it requires more lines of code, it is much easier to understand:

Embedding Created by Author

A good rule of thump is to avoid using two or more expressions with list comprehension including for loops and conditions.

**Generators expressions for large inputs**

While list comprehension is very useful, compact and easy to understand they can require a lot of memory for large inputs. An alternative to list comprehension for large inputs is a generator expression. Generator expressions combine comprehension and generators. They are ideal for large inputs since they yield one item at a time from the expression. To write a generator expression we simply use parenthesis (). Suppose we have a large list of classification prediction probabilities. We will use Numpy’s random.normal method to generate a synthetic list of 200 miillion prediction probabilities with a mean of 0.5 and as standard deviation of 0.1:

Embedding Created by Author

Now suppose we want to generate a list of labels from this using list comprehension. We will assign a label of ‘Yes’ to probabilities greater that 0.5 and ‘No’ to probabilities less than 0.5.

If we try printing the first ten elements in the same cell as our list comprehension we run out of memory:

Embedding Created by Author

So we need to do so in a separate cell:

Embedding Created by Author

Generator expressions can be used to evaluate an iterator that yields one item at a time instead of the whole expression. This can help with avoiding memory allocation issues:

Embedding Created by Author

We see that we are able to create the generator object and print the first ten elements without running into memory issues.

The code in this post is available on [GitHub](https://github.com/spierre91/deepnote/blob/main/list_comp_generators.ipynb).

**Conclusions**

In this post we discussed some useful ways to improve list creation through pythonic thinking. First we discussed how list comprehension should be used over map and filter since it easier to understand especially when collaborating with beginners in python. We then covered how using more than two expressions in a list comprehension should be avoided to maximize clarity and readability. Finally, we showed how to use generator expressions as an alternative to list comprehension for large inputs. I encourage you to apply some of these techniques to your own software engineering and machine learning projects.