# Week 12 - Iterable, Iterator, and Generator

In week 9, we implemented the recursive function **find\_subcategories** by finding the user-input category, flattening its subcategories, and recursively returning the result. Now that you have learned about generators in Python, you might find that a generator could be more concise and straightforward for this function.

## Required Steps

1. Rewrite the **find\_subcategories** method of the **Categories** class using a generator.
   1. Define an inner function **find\_subcategories\_gen**, which is a recursive generator, in the **find\_subcategories** method.
   2. For **find\_subcategories** itself, simply return a list generated by **find\_subcategories\_gen**.
2. Remove the **\_flatten** method, which is no longer needed.

## Hint for find\_subcategories\_gen

You are encouraged to try by yourself before referring to this hint.

Here's the template:

| class Categories:  def \_\_init\_\_(self):  ...  def view(self, ...):  ...  def is\_category\_valid(self, ...):  ...    def find\_subcategories(self, category):  def find\_subcategories\_gen(category, categories):  # A generator that yields the target category and its subcategories    return \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  # A list generated by find\_subcategories\_gen(category, self.\_categories)  # The \_flatten method is no longer needed. Remove it! |
| --- |

1. First of all, you can refer to the **atom\_gen()** example in the lecture of week 12, which generates all the elements in a nested list.

| def find\_subcategories\_gen(category, categories):  if type(categories) == list: # recursive case  for child in categories:  for atom in find\_subcategories\_gen(category, child):  yield atom  else: # base case  yield categories |
| --- |

For now, this generator simply yields all of the categories in the nested list, no caring which **category** is to be found (i.e. the parameter **category**).

1. Before going further, here's a tip to simplify the above code: use a **yield from** statement instead of the inner for-loop.

| def find\_subcategories\_gen(category, categories):  if type(categories) == list: # recursive case  for child in categories:  yield from find\_subcategories\_gen(category, child)  else: # base case  yield categories |
| --- |

Same as the inner for-loop did, **yield from** yields the results yielded from **find\_subcategories\_gen(...)** one by one.

1. Since we don't want to yield every category, we have to set some conditions before the **yield** statement in the base case. The first condition is when **categories** is the same as the **category** we want to find.

| def find\_subcategories\_gen(category, categories):  if type(categories) == list: # recursive case  for child in categories:  yield from find\_subcategories\_gen(category, child)  else: # base case  if categories == category:  yield categories |
| --- |

This way, the generator only yields the target **category**, and all the other categories are skipped.

1. We also want to yield all the subcategories under **category**. The idea is to make a boolean flag **found** in the parameter list. The flag stays as **False** at the beginning and is set to **True** when the function recurs to the subcategories of **category**. In the base case, the category can also be yielded when the flag **found** is True.

| def find\_subcategories\_gen(category, categories, found=False):  if type(categories) == list:  for index, child in enumerate(categories):  yield from find\_subcategories\_gen(category, child, \_\_\_\_\_)  if child == category and index + 1 < len(categories) \  and type(categories[index + 1]) == list:  # When the target category is found,  # recursively call this generator on the subcategories  # with the flag set as True.  yield from \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  else:  if categories == category or \_\_\_\_\_:  yield categories |
| --- |

Fill the blanks yourself!

## Related Knowledge

* Recursive generator