CSC148 - Introducing List Comprehensions

A **list comprehension** is a special type of Python expression that can be used to succinctly create new lists. Instead of writing:

```
result = []
for x in lst:
    result.append(f(x))  # where f is some helper
```

we can simply write:

```
result = [f(x) for x in lst]
```

List comprehensions can often make standard loop patterns more concise, so that our code is both easier to understand and has less possibility for error.

1. Recall that the Python `sum` function takes a list as an argument, and returns its sum. Using this, we can rewrite loops of the form:

   ```
   s = 0
   for x in lst:
       s += x
   ```

   into simply:

   ```
   s = sum(lst)
   ```

   Use `sum` and a list comprehension to implement `sum_nested`, which adds up all the numbers in a nested list.

   ```
   def sum_nested(obj: Union[int, List]) -> int:
       """Return the sum of the numbers in <obj> (or 0 if there are no numbers)."""
       if isinstance(obj, int):
           return obj
       else:
           return sum([sum_nested(elem) for elem in obj])
   ```

2. But `sum` can be used to add more than just numbers! It takes a second argument, `start`, which is the "initial" value to add on to. More generally,

   ```
   s = init
   for x in lst:  # x isn't necessarily a number!
       s += x
   ```

   can be written as

   ```
   s = sum(lst, init)
   ```

   Using this idea and a list comprehension, implement the recursive function `flatten` for nested lists.

   ```
   def flatten(obj: Union[int, List]) -> List[int]:
       """Return a (non-nested) list of the integers in <obj>."""
       if isinstance(obj, int):
           return [obj]
       else:
           flattened = [flatten(sublist) for sublist in obj]
           return sum(flattened, [])
   ```

3. In addition to `sum`, there are two other useful Python built-in functions for simplifying loop patterns: `any` and `all`. Each of these takes a *list of booleans* as an argument. `any(lst)` returns `True` if *at least* one boolean is `True` (and returns `False` otherwise), while `all(lst)` returns true if *every* boolean is `True` (and returns `False` otherwise).

   For example, we can use `any` to rewrite:

   lst[0] OR lst[1] OR . . . OR lst[n]

   lst[0] AND lst[1] AND . . . AND lst[n]

```
s = False
for x in lst:  # x is a boolean
    if x:
        s = True
```

   into simply:

```
s = any(lst)
```

   Use this idea, plus a well-chosen list comprehension, to implement `nested_list_contains`, which searches for a number in a nested list.

```
def nested_list_contains(obj: Union[int, List], item: int) -> bool:
    if isinstance(obj, int):
        return obj == item
    else:
        return any([nested_list_contains(s, item) for s in obj])
```

4. Finally, use some combination of list comprehensions, `any`, and `all` to implement `semi_homogeneous` from last lab's quiz.

```
def semi_homogeneous(obj: Union[int, List]) -> bool:
    """Return whether the given nested list is semi-homogeneous.

    A single integer and empty list are semi-homogeneous.
    In general, a list is semi-homogeneous if and only if:
        - all of its sub-nested-lists are integers or all of them are lists
    AND - all of its sub-nested-lists are semi-homogeneous
    """
```

   (A or B) and C

   A: all of its sub-nested-lists are integers
   B: all of them are lists
   C: all of its sub-nested-lists are semi-homogeneous (recursive)

```
    if isinstance(obj, int) or obj == []:
        return True
    else:
        all_ints = all([isinstance(elm, int) for elm in obj])
        all_lists = all([isinstance(elm, list) for elm in obj])
        all_semis = all([semi_homogeneous(elm) for elm in obj])
        return (all_ints or all_lists) and all_semis
```