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

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

we can simply write:

```python
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:

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

   into simply:

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

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

   ```python
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_nested(sum, init)
```

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,

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

   can be written as

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

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

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

For example, we can use \texttt{any} to rewrite:

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

into simply:

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

Use this idea, plus a well-chosen list comprehension, to implement \texttt{nested\_list\_contains}, which searches for a number in a nested list.

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

4. Finally, use some combination of list comprehensions, \texttt{any}, and \texttt{all} to implement \texttt{semi\_homogeneous} from last lab's quiz.

```python
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
    - all of its sub-nested-lists are semi-homogeneous
    """
```