ReFactory: Recursion Part 1

Table of Contents

- 1) Preparation
- 2) Introduction
  - 2.1) Programming Exercise Instructions
- 3) Factory
  - 3.1) Counting Subparts
  - 3.2) Base Parts
  - 3.3) Max Depth

1) Preparation

This lab assumes you have Python 3.6 or later installed on your machine (3.11 recommended).

The following file contains code and other resources as a starting point for this practice exercise: ZIP FOLDER

This practice exercise is optional and ungraded but it is designed to help you prepare for this week's Recipe's lab.

These problems are also a good way to practice writing code with good style. Upon submitting a correct solution to a problem, you will get access to staff solutions and explanations for each problem.

While we encourage students to collaborate on the concept questions, please refrain from collaborating on the coding problems except with staff members. To allow all students the benefit of working through the problems individually, the course academic integrity policy applies to your solution and the official solution code-- meaning no sharing or posting of answers allowed, especially on public or shared internet spaces.

2) Introduction

Many of our future labs and quizzes will require a solid understanding of recursion. While thinking recursively may seem difficult at first, after enough practice breaking problems into recursive steps and base cases, recursion will become easier. The following problems are intended to be completed recursively (even though all recursive problems can be done iteratively and vice versa.)

In particular, this week's practice exercises are meant to give you opportunities to practice the following skills (which will be useful for this week's lab!):

- Testing using doctests
- Making recursive functions that navigate nested structures
- Using recursive helper functions

2.1) Programming Exercise Instructions

The following sections will contain programming exercises and related concept questions. For each programming exercise, we recommend the following approach:

1. Read the function docstring.
2. Answer the concept questions by hand (you should be able to answer these without writing / running code!)
3. Write the function in the provided practice.py file.
4. Once your code in practice.py passes the test case associated with the function in the test.py file, paste it into the code submission box. Submitting a fully correct solution to the code submission box will unlock the ability to view the staff solutions.

If you have any questions about the code solutions or want to verify your answers to the concept questions, we highly encourage you to check with your peers and/or ask a staff member!

3) Factory

As an intern for Refactory, a parts manufacturing company, you are tasked with writing code that determines various properties of a factory's part manifest. A part manifest is represented as a dictionary, where the keys are the part numbers of finished parts and the values are a list of part numbers that are needed in order to assemble the finished part.

Take for example a pen factory that produces the following items:

- blue pens (assembled using a pen casing and blue ink)
- a small pack of blue pens (assembled using packaging and four assembled blue pens)
- black pens (assembled using a pen casing and black ink)
- a large pack of blue pens (assembled using three packs of blue pens and packaging)

This could be represented as the following part manifest:

```
pen_manifest = {
  5: [1, 2],
  3: [6, 5, 5, 5, 5],
  7: [1, 9],
   10: [3, 3, 3, 6]
 }
```

where the corresponding part numbers to part names are:

```
    1 -> pen casing
    2 -> blue ink
    3 -> small pack of blue pens
    5 -> blue pen
    6 -> packaging
    7 -> black pen
    9 -> black ink
    10 -> large pack of blue pens
```

Note that all the keys of the dictionary are compound parts, meaning that they need to be assembled using one or more sub-parts. There are also atomic parts included in the list of values, which are raw materials that do not need assembly because they have no sub-parts.

> **Check Yourself 1:**
>
> **A.** Which part numbers in the pen_manifest are compound parts? Which part numbers are atomic parts?

3.1) Counting Subparts

Last year's intern, Pye Thon, started implementing a function that determines the number of total subparts required to assemble a part. For example, a blue pen has 2 supbarts. Making part 3, (small pack of blue pens) requires 5 initial sub-parts to make (one part 6 and four part 5s). Each part 5 (blue pen), requires an additional two parts two make (one part 1 and one part 2), for a total of 5 + 4 * 2 = 13 parts. Note making packing or other atomic parts requires 0 additional subparts.

Unfortunately, Pye Thon didn't get the function working before their internship ended, and your manager has assigned you the task of fixing it!

```
def count_subparts(part_manifest, part_num):
    """
    Recursively determine the number of total parts that will be needed
to
    assemble the current part.

    Parameters:
        * part_manifest (dict<int, list<int>>) : a dictionary with
integer part
            numbers mapping to a list of sub-part numbers needed to
assemble the part.
            Each sub-part may or may not themselves be a key in the
dictionary.
        * part_num (int) : the desired part to assemble.

    Returns:
        An integer representing the total number of subparts (including
the
        number of subparts of any compound subparts.

    >>> x = {1: [2, 3, 2, 3], 3: [2, 4, 5], 4: [2, 7]}
    >>> count_subparts(x, 3)
    5
    >>> count_subparts(x, 2)
    0
    >>> count_subparts(x, 1)
    14
    >>> y = {5: [1, 2], 3: [6,5,5,5,5], 7: [1,9], 10: [3,3,3,6]}
    >>> count_subparts(y, 10)
    0 # TODO FIX THIS
    """
```

Check Yourself 2:

A. How many subparts are needed to construct part 10 in the pen factory manifest (copied below)?

```
pen_manifest = {
  5: [1, 2],
  3: [6, 5, 5, 5, 5],
  7: [1, 9],
  10: [3, 3, 3, 6]
}
```

B. Modify the last doctest in count_subparts to check that your code gets the expected answer for part B.

C. Paste your definition of count_subparts below: (click in the box, use ctrl+a to highlight text, ctrl+v to paste)

```python
def count_subparts(part_manifest, part_num):
    pass # Your code here
```

3.2) Base Parts

Your next task is to determine the unique set of atomic parts required to assemble a single part.

```python
def find_base_parts(part_manifest, part_num):
    """
    Recursively determine the unique atomic parts needed to assemble a
part_num.

    Parameters:
        * part_manifest (dict<int, list<int>>) : a dictionary with
integer part
            numbers mapping to a list of sub-part numbers needed to
assemble the part.
            Each sub-part may or may not themselves be a key in the
dictionary.
        * part_num (int) : the desired part to assemble.

    Returns:
        A set of integers representing the atomic parts needed to
assemble the part.

    >>> x = {1: [2, 3, 2, 3, 6], 3: [2, 4, 5]}
    >>> list(sorted(find_base_parts(x, 3))) # returns a set {2, 4, 5}
    [2, 4, 5]
    >>> list(sorted(find_base_parts(x, 2))) # returns a set containing
atomic part {2,}
    [2]
    >>> list(sorted(find_base_parts(x, 1))) # returns a set {2, 4, 5, 6}
    [2, 4, 5, 6]
    """
    pass
```

Check Yourself 3:

A. What are the unique atomic part(s) required to assemble part 10 (large pack of blue pens) in the pen_manifest (copied below)?

```
pen_manifest = {
  5: [1, 2],
  3: [6, 5, 5, 5, 5],
  7: [1, 9],
  10: [3, 3, 3, 6]
}
```

B. What are the base case(s) for the `find_base_parts` function?

C. What are the recursive case(s) for the `find_base_parts` function?

D. Paste your definition of find_base_parts below: (click in the box, use ctrl+a to highlight text, ctrl+v to paste)

```
def find_base_parts(part_manifest, part_num):
    pass # Your code here
```

3.3) Max Depth

Your final task is to determine the maximum number of assembly steps a single part needs. For example a blue pen requires a single assembly step to combine the ink and the casing. A small pack of blue pens requires two steps: first assembling the pens, and then combining the pens with the packaging.

```
def maximum_steps(part_manifest):
    """
    Recursively determine the maximum number of assembly steps it would
take to assemble
    a part in the manifest.

    Parameters:
        * part_manifest (dict<int, list<int>>) : a dictionary with
integer part
            numbers mapping to a list of sub-part numbers needed to
assemble the part.
            Each sub-part may or may not themselves be a key in the
dictionary.
```

```
    Returns:
        An integer representing the maximum depth of the part_manifest.

    >>> maximum_steps({1: [2, 3, 2, 3], 3: [2, 4, 5]})
    2
    >>> maximum_steps({3: [2, 4, 5]})
    1
    >>> maximum_steps({2: [1], 4: [3, 2], 3: [2, 1]})
    3
    """

    def max_helper(part_num):
        """
        Given a part_num (int), return the maximum number of assembly
steps.
        Base parts (parts that have no sub-parts) have 0 assembly steps.
        """
        return 0


    pass
```

Check Yourself 4:

A. How many assembly steps does each compound part require in the pen factory example (copied below)? What is the maximum number of assembly steps?

```
pen_manifest = {
  5: [1, 2],
  3: [6, 5, 5, 5, 5],
  7: [1, 9],
  10: [3, 3, 3, 6]
}
```

B. What are the base case(s) for the `max_steps` function?

C. What are the recursive case(s) for the `max_steps` function?

D. Paste your definition of maximum_steps below: (click in the box, use ctrl+a to highlight text, ctrl+v to paste)

```
def maximum_steps(part_manifest):
    pass # Your code here
```