Most Common

As in Word Order, we are keeping track of the counts of some keys, so a dictionary is an obvious structure. Now the keys will be individual characters, not whole words, but the basic idea is the same.

There is a slight wrinkle to this, which we didn't spend much time on yesterday: how do you ensure there's an entry in the dictionary for each key? You could create a new entry for every possible key prior to reading the string, but that would waste a lot of space, at least if we decided we had to deal with unicode. We could check whether the character was in the dictionary and create or increment the count as appropriate, but that would be a bit of a pain. (This is what yesterday's solution did). Luckily, the Python standard library gives us the tool we need: defaultdict, from the collections module. defaultdict takes a function as an argument, and when a non-existent key is referenced, the output of that function is used as the value. Here we use the `int` method, which returns `0` when called without an argument.

```
1   from collections import defaultdict
2
3   s = input()
4   d = defaultdict(int)
5   for c in s:
6       d[c] += 1
7   counts = sorted(d.items(), key=lambda x: (-x[1], x[0]))
8
9   for c, n in counts[:3]:
10      print('{c} {n}'.format(c=c, n=n))
```

Let's take a second to unpack what's going on in line 7. For a dictionary `d`, `d.items()` returns a list of (key, value) tuples of the dictionary's entries. `sorted` takes a list and returns a copy with the list elements sorted. In this case, the list elements are (character, count) tuples. Python will order tuples by their first element, breaking ties with the second element, and so on, so this would sort the list alphabetically.

We can change this by passing the `key` keyword to `sorted`. This specifies a function which, given a list value, returns the value that will be used for sorting. In this case, we specify an anonymous function, using `lambda`, that takes the (character, count) tuple and specifies that it should be sorted by (-count, character). The negative sign ensures that the largest count will sort first. Having the character be the second element of the sort tuple ensures

that the characters with the same count will be sorted alphabetically, as the challenge requested.

Compartmentalization

It is good coding practice to split up your code into functions, each of which does only a single thing. Here, there are three main tasks, counting the letters, getting the top three, and printing the results. We make a function for each. To make our code more general, we make the second step accept an optional argument of the number of top items to return. This way, if the requirements change to specify the top five, we can easily update the code.

```
1   from collections import defaultdict
2
3   def count_letters(s):
4       d = defaultdict(int)
5       for c in s:
6           d[c] += 1
7       return d
8
9   def get_top_n(d, n=3):
10      return sorted(d.items(), key=lambda x: (-x[1], x[0]))[:n]
11
12  def print_counts(counts):
13      for c, n in counts:
14          print('{c} {n}'.format(c=c, n=n))
15
16  s = input()
17  print_counts(get_top_n(count_letters(s)))
```

With code like this, we can test each of the functions in the interpreter with test input. This will help us isolate which part of the code is failing. Only once we have it working can we try to hook it up to stdin. If it fails at this point, the problem is probably in parsing the input, not in the processing.

What's in a __name__?

You've probably noticed that the solutions here don't have the same boilerplate code that you were given as a starting position. There are two bits of that which you may not recognize:

1. The "shabang" line at the top (`#!/bin/python3`), which just tells the (Unix-based) operating system which interpreter to run this file through.

2. The `__name__ == '__main__'` test at the bottom. This is a standard guard in Python modules, so we'll explore it further.

There are two ways for a .py file to be used: as a script or as a module loaded by another file. In either case, what happens is about the same: the python code in it is executed. This ensures that all functions and classes are defined, but it also means that any procedural code at the top level will be run. If you wanted to use this `count_letters()` function in another script, you'd find importing this file would cause stdin to be read and that whole function pipeline to be called on it. This is probably not what you want.

To avoid this, we check on the value of a special variable, `__name__`. When a Python file is run as a script, it is set to be the string "__main__". When it is loaded as a module, `__name__` is set to the module name. Thus, by testing `if __name__ == '__main__'`, we can ensure that code will run only when the file is being run as a script.

```
1   from collections import defaultdict
2
3   def count_letters(s):
4       d = defaultdict(int)
5       for c in s:
6           d[c] += 1
7       return d
8
9   def get_top_n(d, n=3):
10      return sorted(d.items(), key=lambda x: (-x[1], x[0]))[:n]
11
12  def print_counts(counts):
13      for c, n in counts:
14          print('{c} {n}'.format(c=c, n=n))
15
16  if __name__ == '__main__':
17      s = input()
18      print_counts(get_top_n(count_letters(s)))
```

All of the procedural code has been moved inside of this guard. Thus, when this file is imported as a module, only the function definitions are evaluated.

Using this guard is unnecessary for HackerRank challenges, but you will see this mechanism often so it's good to know what it does. It's not a bad idea to get in the habit of using it yourself.