Further Readings for Week 6

This week we started learning Python after a few weeks of Bash. Bash is primarily a shell scripting language and is helpful for handling system operations. Bash makes it easier to work with the operating system directly and is well-suited for interactive, terminal-based tasks.

While Bash is fun to learn and work with(you can disagree!), Python is simpler to read as well as code in, more versatile and user-friendly. It also offers a vast ecosystem of libraries across different domains like data science(you may have heard of or used Numpy, Pandas libraries), machine learning(Scikit-learn, PyTorch), web development(Django, BeautifulSoup, Selenium), artificial intelligence(OpenCV, NLTK) and for our purposes, bioinformatics(BioPython, ScanPy).

Further, there are limitations with what we can do in Bash or rather, easier and more elegant ways to do it in higher-level languages like Python. In the Week 3 reading, we noted such a use case. While we were able to extract the comma-separated numbers enclosed in quotes from a CSV file using a combination of regex and `sed`, we can do that easily in Python using the standard library `csv`'s `reader` function or any CSV-handling function provided by other libraries(like Pandas `read-csv` function).

A few more inconveniences of using shell-scripting languages like Bash are as elaborated here:

- Most scripts fail if filenames contain whitespaces. You can work around this by quoting them, but it can get messy and complicated.
- As you may have experienced, error messages in Bash are not very helpful. There is also no stack trace that most higher-level languages provide when an error is thrown. So it can take a lot of time to analyze why an error occurred.
- Even worse, the Bash interpreter might ignore errors and step silently to the next line. These are called "silent errors" and are harder to debug.
- Shell creates processes for every command you run(for example, `grep`, `sed`, `awk` etc). Depending on how you write your Bash script, creating multiple sub-processes within the script can slow down its execution. You would have experienced the pain of this if you used something like `blast_start=$(echo "$blast_hit" | cut -f9')` inside the nested loops for your assignment 4!

Different counting systems

By now, you already know Python uses a 0-based counting system. It means that for indexing data types like lists, strings and so on, we start from 0 rather than 1. Let's look at how 0-based and 1-based counting systems work and the rationale behind using 0-based system:

1-based counting

This is the counting we all learned in elementary school, start at 1 and count how many ever objects we have. So if there is a list of alphabets `a, b, c, d, e, f`, we count them as:

```
a b c d e f
1 2 3 4 5 6
```

Since we start with 1 (included) and stop at 6 (included), it is also called ***closed-system counting***. What is the problem with using this method? Well, if we want to find the size of our list, we need to do `(6 - 1) + 1 = 6` i.e, `(stop - start) + 1` which means we need an additional operation of (+1) to calculate the size of the interval.

0-based counting

With this counting system, we start counting at 0 and stop at the number of items plus 1. If we take the same example list of alphabets, we can count as below:

```
a b c d e f
0 1 2 3 4 5 6
```

As we start at 0 (included) and stop at 6 (which is excluded), this counting method is called ***half-open counting***. While this may sound unnecessarily complicated and unintuitive if we want to find the size of the list, we can simply do `(6 - 0) = 6` i.e, `(stop - start)`.

To look at a Python-specific example, if we want to slice a list and get first 3 elements, we can use `my_list[0:3]` or simply just `my_list[:3]` by taking advantage of the half-open counting method.

Not just Python, we encounter 0-based counting at other places too. If you remember, the BED files we saw earlier are also 0-based where the start position is inclusive and stop position is not. Since it is pretty common that we want to find the size of features in the BED file, it makes sense to use 0-based counting to save an operation: `(stop - start)` vs `(stop - start) + 1`.

We'll be dealing with 0-based and 1-based counting quite a few times throughout the course and in future assignments, so it is helpful to solidify your understanding to be able to identify what counting system is used and convert between the two when necessary. This is a great article from UCSC Genome Browser blog which explains the two counting systems and this article written by one of Python developers talks about why Python adopted a 0-based counting method.

Interesting case of `break` and `continue`

We saw the use of `break` and `continue` statements during assignment 4 review. To recap, `break` breaks out of the current loop of execution and `continue` skips the rest of the code that comes after it within the current loop and proceeds with the next iteration. Prof. Collins pointed me to an interesting feature of `break` and `continue` that is *supported in Bash but not in Python*. Let's review it below:

Bash supports **labeled** `break` and `continue` statements which allow specifying how many nested loops we want to break out of or continue when a certain condition is met. With these, we can control the flow of execution for not just the immediate loop, but also loops that are further up the hierarchy.

Let's first look at the default behavior with a `break` statement example below:

```
for i in $(seq 3); do
  for j in $(seq 3); do
    if [[ $j -eq 2 ]]; then
      echo $j
      break 1  # Exits only the inner loop
```

```
    fi
    echo "$i $j"
  done
done

1 1
2    <----breaks here
2 1
2    <----breaks here
3 1
2    <----breaks here
```

Whenever the value of `j` becomes two, the `if` condition is satisfied. So it `echo`'s the value of `j` and breaks out of the innermost loop but continues with the next iteration of the outer loop. Python's `break` statement works similarly.

But what additional feature Bash supports can be demonstrated below:

```
for i in $(seq 3); do
  for j in $(seq 3); do
    if [[ $j -eq 2 ]]; then
      echo $j
      break 2  # Exits both the inner and outer loops
    fi
    echo "$i $j"
  done
done

1 1
2    <----breaks here
```

The number after `break` indicates the levels of nesting it should break out of, so we see that when `j` becomes 2 in the first iteration, it `echo`'s `j` and breaks out of both the inner and the outer loop. `continue` also works similarly.

If you think this is a cool feature that Python should have, somebody else also thought so and created a feature request(in Python terms, this is called a Python Enhancement Proposal or PEP for short) that Python supports labeled `break` and `continue`. However, since Python's development is largely community-based, the Python developers review these PEPs and decide if a feature gets implemented. Unfortunately, the PEP to implement labeled `break` and `continue` was rejected. You can view the PEP and why it was rejected here. The gist is that Bash primarily being a system-level scripting language might benefit from such finer level of control on the flow of execution while Python sticks to its principle of keeping things simple and readable.