How to think like a programmer

Genome 559: Introduction to Statistical and Computational Genomics

Seungsoo Kim

The keys to programming

- By now you know enough Python to solve almost any problem:
  - variables and types
  - loops
  - conditional statements
  - lists and dictionaries
  - arguments
  - file input and output
  - functions
- The challenge is figuring out how to break down complex problems into these basic elements

Three keys to good programming

1. Understand how things work
2. And then don’t worry about the details (abstraction)
3. Consider the tradeoffs between run time, memory efficiency, and your time (to write the program)

We often do fairly complicated tasks *intuitively*

- What's the biggest number in this list?
  - 0, 1, 2, 3, 4, 10, 5, 6

But actually, we're using algorithms

- What's the biggest number in this list?
  - 0, 1, 2, 3, 4, 10, 5, 6
- What's the biggest number in this list so far?
  - 0
- Is the next number bigger than the biggest one so far?
  - 1 > 0

Python makes it intuitive to solve some problems

- What's the biggest number in this list?
  - 0, 1, 2, 3, 4, 10, 5, 6
- `max([0, 1, 2, 3, 4, 10, 5, 6])`

But the computer is breaking this down into very basic operations

- What's the biggest number in this list?
  - 0, 1, 2, 3, 4, 10, 5, 6
- `max([0, 1, 2, 3, 4, 10, 5, 6])`
- How is the max function written?

Exercise #1: Write a `max` function

Using the logic we discussed, write a function `my_max` that takes a list of integers and returns the largest number in that list. You can assume the list is not empty and only contains integers.

Exercise #1 solution

```
def my_max(list1):
    max_so_far = list1[0]
    for element in list1:
        if element > max_so_far:
            max_so_far = element
    return max_so_far
```

Exercise #1 continued

- Write a function that returns the sum of the elements in a list of integers
- Write a function that returns the smallest value from a list of integers
- Remember, to calculate something over a list, create a variable to keep track of your calculations so far as you loop over the list

Exercise #1 continued solutions

```
def my_sum(list1):
    sum_so_far = 0
    for element in list1:
        sum_so_far += element
    return sum_so_far

def my_min(list1):
    min_so_far = list1[0]
    for element in list1:
        if element < min_so_far:
            min_so_far = element
    return min_so_far
```

Wait, how is that for loop working?

- Python conveniently lets you write:

```
for element in list:
    ...
```

- This is a shortcut for:

```
for i in range(len(list)):
    element = list[i]
    ...
```

- In a while loop:

```
i = 0
while i < len(list):
    element = list[i]
    ...
    i += 1
```

But you don't always want to break everything down

What does this function do?

```
def my_function(a, b):
    x = 0
    i = 0
    j = 0
    while i < a:
        while j < b:
            x += 1
            j += 1
        i += 1
        j = 0
    return x
```

The power of programming comes from abstraction

- For example, you don’t have to worry about how the computer is actually reading a file, which is ultimately a string of 0s and 1s
- Functions can serve as black boxes – as long as you know what it takes in and what it outputs, you don’t need to worry about how it’s doing what it’s doing

But - you have to know enough about what your computer is actually doing under the hood

- Some algorithms are more efficient than others
  - Runtime efficiency
  - Memory efficiency

A case study of efficiency

- You have a dictionary and want to look up the definition of “program”
- One way is to open up your dictionary to page 1, and check the first word.
- If it’s “program”, you’re done. If not, check the next word.
- Repeat until you find it!
- This is how Python lists work (list.index)
- How long does this take, on average? In the worst case scenario?

Big *O* notation

- How does the runtime depend on the size(s) of the input(s)?
- Specifically, in the worst case scenario, e.g. if you wanted to look up “zyzzogeton”
- In this case, how does the runtime depend on the number of words in the dictionary, n?
  - *O*(n) – linear
  - This means “Order of” – don’t worry about constants

A better way to look up a word in the dictionary

- Open up to about the middle of the dictionary
- Check whether the word you're looking for is before or after the point you opened to
- Repeat until you find it!
- How long does this take?
  - At worst, $\log_2 n$, or $O(\log n)$

Python dictionaries, or hash tables or hash maps, are even better!

- Python dictionaries are *O*(1), or in constant time
- To an approximation, it takes no longer to search a 1,000,000-word dictionary than a 100-word dictionary
- How?
- It uses a mathematical function that maps every possible key (e.g. a word) to a location in memory

Revisiting the kmer problem

- Recall PS4 #7, in which you were asked to count the number of occurrences of each kmer of length k in a given sequence file.
- Many of you enumerated all possible kmers of length k, and then searched for the number of occurrences of each kmer

```
for kmer in possible_kmers:
    kmer_count = seq.count(kmer)
```

- But wait, what is **`seq.count`** doing?

Exercise #2: Write a `string.count` function

- Write a function that takes two strings, and finds how many times the second occurs in the first.
- string_count(“ABCXYZABC”,”ABC”) => 2

Exercise #2: pseudocode

```
# for position in string1
#     initialize counter
#     if substring starting at position matches string2
#         increment counter
# print counter
```

Exercise #2: solution

```
def string_count(str1, str2):
    hits = 0
    for i in range(len(str1)-len(str2)+1):
        if str1[i:i+len(str2)] == str2:
            hits += 1
    return hits
```

Exercise #2b: Write a `string.find` function

- Write a function that takes two strings, and finds the index of the first occurrence of the second string within the first.
- string_find(“ABCXYZABC”,”XYZ”) => 3
- Remember, Python is 0-indexed:

```
0 1 2 3 4 5 6 7 8
A B C X Y Z A B C
```

Exercise #2b: solution

```
def string_find(str1, str2):
    for i in range(len(str1)-len(str2)+1):
        if str1[i:i+len(str2)] == str2:
            return i
```

Revisiting the kmer problem

- Recall PS4 #7, in which you were asked to count the number of occurrences of each kmer of length k in a given sequence file.
- Many of you enumerated all possible kmers of length k, and then searched for the number of occurrences of each kmer

```
for kmer in possible_kmers:
    kmer_count = seq.count(kmer)
```

- `seq.count` goes through each substring of length k in seq and checks if it matches the kmer you're looking for
- How long will this take?
  - (number of kmers) * (time to find all occurrences of one kmer)
  - $(5^k)*O(n)$ or $O(e^k n)$

Revisiting the kmer problem

- Instead, you could go through each kmer in the sequence seq once (which you previously did 5^k times)

```
for i in range(len(seq)-k+1):
    kmer = seq[i:i+k]
    if kmer in kmer_dict:
        kmer_dict[kmer] += 1
    else:
        kmer_dict[kmer] = 1
```

Exercise #3: spell checker

- Write a program that reads in a file containing a list of words (“dictionary”) each on one line (e.g. words.txt), and a second file containing the query text, and then prints the words in the second file that are not in the “dictionary”.
- Do you need to keep the whole “dictionary” in memory? What about the query text?
- Remember to remove punctuation

Exercise #3 pseudocode

```
# read first file and save each line
as an entry in a dictionary
# read the second file
#     for each word, check if in dict
#         if not in dict, print word
```

Exercise #3 solution

```
import sys
dict = open(sys.argv[1])
text = open(sys.argv[2])

worddict = {}
for line in dict:
    worddict[line.strip()]=None
dict.close()
for line in text:
    words = line.strip().lower().translate(None,'.,;:"()?!').split()
    for word in words:
        if word not in worddict:
            print word
text.close()
```

Exercise #3 continued

- Modify your program to time how long the program takes. Remember that if you **`import time`**, time.time() prints the current time in seconds (after some fixed point in time a long time ago).
- Rewrite your program to use a list instead of a dictionary (save both versions)
- Run both versions of the program with various “dictionaries” and “queries” and compare the runtimes

Exercise #3 solution with list and timing

```
import sys
import time

starttime = time.time()
dict = open(sys.argv[1])
text = open(sys.argv[2])

worddict = []
for line in dict:
    worddict.append(line.strip())
dict.close()
for line in text:
    words = line.strip().lower().translate(None,'.,;:"()?!').split()
    for word in words:
        if word not in worddict:
            print word
text.close()
print time.time()-starttime
```

Three keys to good programming

1. Understand how things work
2. And then don’t worry about the details (abstraction)
3. Consider the tradeoffs between run time, memory efficiency, and your time (to write the program)

Extra challenge problems

- Implement Fitch's algorithm if you haven't already (see PS5)
- Implement the algorithm we used to search the dictionary (sorted list of words), using Python lists