Nick Bowman, Sonja Johnson-Yu, Kylie Jue
CS 106AP

Assignment #2
July 2, 2019

Assignment #2: Khan-sole Academy

On-time deadline: 11:59 PM on Wednesday, July 10
Extended deadline: 11:59 PM on Thursday, July 11
This assignment should be done individually.

This assignment consists of six problems. You can download the starter code for this project under the "Assignments" tab on the CS106AP website. The starter project will provide Python files for you to write your programs in.

In the spirit of learning how to code, in this assignment we're going to use code to help others learn! Specifically, we'll practice using control flow, variables, and functions to look at patterns in math and biology. We'll also use the `input()` built-in to get user input through the console for each problem. Each of the math and biology sections will include both a problem that we can solve as humans and also a problem that is made much easier when we have programming on our side. For extra credit, you'll get the chance to put everything together to build Khan-sole Academy: a tool for our friends to practice math and biology problems!

AN IMPORTANT NOTE ON TESTING:

For each problem, we give you specific guidelines on how to begin decomposing your solution. While you can (and should) add additional functions for decomposition, **you should not change any of the function names or parameter requirements that we already provide to you in the starter code.** Since we include doctests for these pre-decomposed functions, editing the function headers can cause existing tests to fail.

As requirement for this assignment, you must adhere to the following testing guidelines:

- For each pre-decomposed function that we provide you, you must write **at least 3 additional doctests per function**.
  - The exceptions to this are that you do not have to write doctests for the following functions: `liftoff()`, any of the `main()` functions, and any of the functions that are part of the bonus problem starter code
- Any new functions that you write should have **at least 1 doctest**.

All tests for a given function should cover completely separate cases, and we encourage you to consider both common use cases and edge cases as you write your doctests. Using good testing practices and thinking through possible inputs/outputs for your code will help increase your likelihood of getting full functionality credit for your work!

Based on handouts and problems by Chris Piech, Jerry Cain, and Eric Roberts.

Warmups

Before we get started on Khan-sole Academy, let's do a couple of warm-up problems to practice control flow, variables, and functions. The first will require you to trace through a program by hand to demonstrate your understanding of functions and strings. You'll write your answer in the **`warmup.txt`** file. The second problem will be written in **`liftoff.py`**.

1. As a warmup, trace the code below. **Please try doing this by hand rather than plugging it into an interpreter.** What is printed to the console when the following function call is made? Put your answer in **`warmup.txt`**.

`mystery()`

```
"""
This program is not useful but is meant to test your
understanding of functions, parameters, and return values!
"""

def cupid(s1, ch):
    return s1 + ch.lower()

def valentine(s1, s2):
    num = len(s1[1:2])
    s1 = s2[num:]
    s2 = cupid(s1, s2[0])
    return s2

def mystery():
    s1 = 'Heart'
    s2 = valentine('candy', s1)
    print('s1 = ' + s1)
    print('s2 = ' + s2)
```

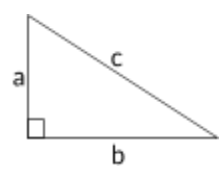
2. As a warmup, write a program that prints out the calls for a spaceship that is about to launch. Countdown the numbers from 10 to 1 and then write "Liftoff." You must use a **`range()`** for loop, and the output should look as it does in Figure 1.

```
10
9
8
7
6
5
4
3
2
1
Liftoff
```

Figure 1: The output of your program in **`liftoff.py`**.

Patterns in math

3. In geometry, you learned the Pythagorean theorem for the relationship among the lengths of the three sides of a right triangle:



$$a^2 + b^2 = c^2$$

which can alternatively be written as:

$$c = \sqrt{a^2 + b^2}$$

Most of this expression contains operators that we have already covered. The one piece that's missing is taking square roots, which you can do by calling the standard function **`math.sqrt()`**. For example, the statement

```
y = math.sqrt(x)
```

assigns **`y`** to the square root of **`x`**.

In **`pythagorean.py`**, write a program that accepts values for **`a`** and **`b`** as floats (you can assume that **`a`** and **`b`** will be positive) from the user and then calculates the solution of **`c`** and prints it on the screen. Your program should be able to duplicate the sample run shown in Figure 2.

```
Enter values to compute the Pythagorean theorem.
a: 3.5
b: 4.2
c = 5.4671747731346585
```

Figure 2: A sample output of your program in **`pythagorean.py`**.

Your code should be decomposed so that it includes a function called **`pythag()`**, which takes in **`a`** and **`b`** as parameters and returns **`c`**. You should call your function after using **`input()`** to get the values for **`a`** and **`b`** from the user and before printing out **`c`** on the screen.

4. What about math problems that are more difficult for humans to solve? Douglas Hofstadter's Pulitzer-prize-winning book *Gödel*, *Escher*, *Bach* contains many interesting mathematical puzzles, many of which can be expressed in the form of computer programs. In Chapter XII, Hofstadter mentions a problem that is well within the scope of the control statements we have learned so far. The problem can be expressed as follows:

   Pick some positive integer and call it *n*.
   If *n* is even, divide it by two.
   If *n* is odd, multiply it by three and add one.
   Continue this process until *n* is equal to one.

   On page 401 of the Vintage edition, Hofstadter illustrates this process with the following example, starting with the number 15:

15	is odd, so I make 3*n*+1:	46
46	is even, so I take half:	23
23	is odd, so I make 3*n*+1:	70
70	is even, so I take half:	35
35	is odd, so I make 3*n*+1:	106
106	is even, so I take half:	53
53	is odd, so I make 3*n*+1:	160
160	is even, so I take half:	80
80	is even, so I take half:	40
40	is even, so I take half:	20
20	is even, so I take half:	10
10	is even, so I take half:	5
5	is odd, so I make 3*n*+1:	16
16	is even, so I take half:	8
8	is even, so I take half:	4
4	is even, so I take half:	2
2	is even, so I take half:	1

   As you can see from this example, the numbers go up and down, but eventually—at least for all numbers that have ever been tried—comes down to end in 1. In some respects, this process is reminiscent of the formation of hailstones, which get carried upward by the wind over and over again before they finally descend to the ground. Because of this analogy, this sequence of numbers is usually called the **Hailstone sequence**, although it goes by many other names as well.

   In **`hailstone.py`**, write a program that reads in a number from the user and then displays the Hailstone sequence for that number, just as in Hofstadter's book, followed by a line showing the number of steps taken to reach 1. Note that all of the

numbers in the program are integers, not floats. Your code should be able to produce a sample run that looks like Figure 3.

```
Enter a number: 17
17 is odd, so I make 3n + 1: 52
52 is even, so I take half: 26
26 is even, so I take half: 13
13 is odd, so I make 3n + 1: 40
40 is even, so I take half: 20
20 is even, so I take half: 10
10 is even, so I take half: 5
5 is odd, so I make 3n + 1: 16
16 is even, so I take half: 8
8 is even, so I take half: 4
4 is even, so I take half: 2
2 is even, so I take half: 1
The process took 12 step(s) to reach 1.
```

Figure 3: A sample output of your program in **`hailstone.py`**.

The fascinating thing about this problem is that no one has yet been able to prove that it always stops. The number of steps in the process can certainly get very large. How many steps, for example, does your program take when *n* is 27?

Patterns in biology

5. In biology, you might have learned that the fundamental unit of DNA is a nucleotide, or base. The four possible bases for DNA are Guanine (G), Cytosine (C), Adenine (A), and Thymine (T). These nucleotides form "base pairs" that make up complementary strands of DNA (which create its double-helix structure). As shown in Figure 4, A pairs with T, and G pairs with C.

Figure 4: DNA is made up of complementary strands of nucleotides.

In **`complement.py`**, write a program that prompts the user for one DNA strand and then prints out the complementary strand. Figure 5 shows a potential sample output of your program.

```
Please give me a DNA strand and I'll find the complement: ATGCAAG
The complement of ATGCAAG is TACGTTC
```

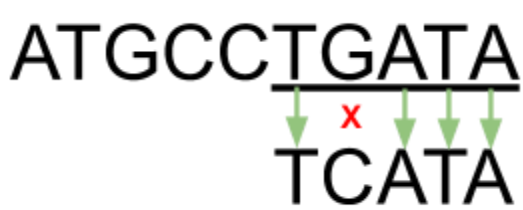
Figure 5: A sample output of your program in **`complement.py`**.

Your code should be decomposed so that it includes a function called **`build_complement()`**, which takes in a strand of DNA as a string and returns its complement as a string. You should call your function after using **`input()`** to get the strand from the user and before printing out the complement on the screen. Your function's output should be case-insensitive; that is, **`build_complement('ATG')`** and **`build_complement('aTg')`** should return the same result. Your function can assume that all of the base pairs of the input string are valid DNA base pairs– that is, the string consists only of the following characters: **`'a', 'A', 'g', 'G', 't', 'T', 'c', 'C'`**

6. As humans, we can easily look at short segments of a DNA strand and generate the complementary bases. But what about identifying the similarity between two different strands of DNA? A common use of programming in biology is to determine segments of DNA that have the highest similarity when aligning nucleotides. In particular, this problem will look at how we might measure **homology**, or the alignment of matching bases.

   In **`similarity.py`**, write a program that asks the user for a strand of DNA to search (we'll refer to this as the "search strand") and a strand of DNA to look for within the search strand (we'll refer to this as the "match strand"). Both of these strands are represented as strings. Your program should return the sub-strand of the search strand that has the most matching nucleotides with the match strand. We define "matching nucleotides" as when the two strands have the same nucleotide at the same index (i.e. if the second nucleotide in both strands is A, then that counts as one matching nucleotide). See Figure 6 for diagrams of how we measure homology and an example run of the program.

A non-optimal substring (2 matching bases)



The optimal substring (4 matching bases)

```
Please give me a DNA sequence to search: ATGCCTGATA
What DNA sequence would you like me to match? TCATA
The best match is TGATA
```

Figure 6: We measure homology by aligning the match strand with a substring of the search strand (previous page). For the search strand "ATGCCTGATA" and the match strand "TCATA," the best match is "TGATA" since it has four matching bases, and running **`similarity.py`** would produce the output shown (above).

You should decompose your program in the following way:

- Write one function called **`search_sequence()`** that takes in as parameters the match strand and an equal length substring of the search strand and returns the number of nucleotides that match between the two.
- Write a second function called **`find_best_substrand()`** that loops over the search strand, generates substrings of equal length to the match strand, and calls your first function to find the sub-strand with the most matching nucleotides.

You can assume that the user will only input strands that include A, T, G, or C and that the strand being searched will be longer than the strand you're trying to match. Note that your program should be **case insensitive** (it should work regardless of whether the user types in capital or lowercase letters for each strand).

Measuring homology has applications in medicine for detecting diseases based on genetic mutations, like sickle cell disease or certain forms of cancer. As the strands of DNA that you're comparing become longer, the problem becomes increasingly more difficult. Computers are an essential tool in helping us find solutions!

Khan-sole Academy (Bonus)

7. Now that you've seen how programming can help us in both math and biology, it's time to put everything together to implement Khan-sole Academy. In this problem, you'll write a program that asks the user for a type of exercise, randomly generates a problem of the given type, reads in the answer from the user, and then checks to see if they got it right or wrong. We'll use some of the helper functions you wrote in your other programs, so make sure you have those parts working correctly first!

   In Khan-sole Academy, a user can choose from four different types of exercises:
   1. Addition
   2. Subtraction
   3. Pythagorean
   4. DNA complement

Note that we won't be asking our users to answer hailstone or similarity problems since those are best solved by a computer anyway.

```
    Enter the number of the type of arithmetic problem you want to practice.
    1) Addition
    2) Subtraction

Your choice: 1
What is 7 + 7?
Your answer: 14
Correct, great work!

    Enter the number of the type of arithmetic problem you want to practice.
    1) Addition
    2) Subtraction

Your choice: 2
What is 19 - 20?
Your answer: 1
Incorrect. The expected answer was -1
```

Figure 7: The start of an example run of **`khansole_academy.py`**. The first question shows a correctly answered addition problem, and the second shows an incorrectly answered subtraction problem. This is what running the program will look like before you've edited the code.

First, try running the program to get a feel for how it works with just addition and subtraction. You should see an output similar to Figure 7. You should also read through the code that's already inside the **`khansole_academy.py`** file to understand how it's working.

To receive extra credit, you'll need to add the last two types of problems, Pythagorean theorem and DNA complement problems.

When writing your code, you should have separate functions for each of the following tasks:

- Generating a Pythagorean problem with random values for **`a`** and **`b`** (the user will input **`c`**)
- Calculating the correct answer for **`c`** in a Pythagorean problem with a given **`a`** and **`b`**
- Generating a DNA complement problem with a random DNA string containing anywhere from 4 to 10 bases, where a single base is either A, T, C, or G
- Determining the correct complement in a DNA problem with a given DNA strand

For randomly generating problems, you'll need to use Python's [random library](), which includes both a **`random.randint()`** function and a **`random.choice()`**

function. Take a look at the linked documentation to understand the arguments and output for each function. The existing code addition and subtraction problems also provides examples for how to **`random.randint()`**.

Your program should first allow the user to select and attempt 5 questions and then should print out the user's score at the end (the total number of correctly answered questions). Since the program already keeps track of and prints out the score, you'll only need to make modifications to let users select your added problem types.

Hint: Can you reuse any of the code you used in problems 3 and 5? Good decomposition is key to solving this problem, and once you understand the general pattern you need to follow, you might even consider adding more problem types beyond Pythagorean theorem and DNA complements (multiplication, division, and more)!

Congrats on finishing! You now have your own mini version of Khan(-sole) Academy to show your friends :)