Grade 6 Math Circles

November 24, 2021

Computer Science Part 2 - Solutions

Do not worry about your solutions exactly matching what it written here. In computer science, there are multiple ways to write programs depending on your preference. The efficiency of your program is not a concern of these lessons, just that they work the way they are supposed to. To make sure your code is correct, test it out with many different values to make sure it is outputting the correct results.

Exercise Solutions

Activity 1

Write a program called *evens* that inputs any integer, and outputs the following:

- *True*, if the integer is even
- *False*, if the integer is odd

Activity 1 Solution

Our approach for this program is to use a conditional statement. If the integer is even, then the remainder when its divided by 2 is 0. So that is our first condition, and it is true then we will return $True$. If the integer is not even, then it must be odd, so we write **else** and return $False$. This gives us the following code:

```
def evens(i):

    if i % 2 == 0:    # remainder when i is divided by 2 is 0
        return True
    else:             # remainder when i is divided by 2 is not 0
        return False
```

An even neater way this code could be written is by simply returning the **bool** value of the first condition above, as shown below:

```
def evens(i):

    return i % 2 == 0
```

Activity 2

Write a program called *number_sum* using recursion, that inputs two integer values, where the second integer is greater than or equal to the first integer, and outputs the sum of every integer between and including the integers.
(For example: $number\_sum(-1, 5)$ outputs 14)

Activity 2 Solution

Our approach for this program is to start with the value of a, and then add each integer following a until we get to b, using recursion. To do this, we will have a conditional statement with the first condition being our base condition, so we can stop the recursion. We know that we want the value of b to be the last integer we add, so when the value of a is equal to the value of b, we will return a (or b), which will end the recursion. This also covers the case where the initial values of a and b are equal, so the result of the program will just be that value. Our **else** condition will then return the value of a plus the value of $number\_sum(a + 1, b)$. This gives us the following code:

```
def number_sum(a, b):

    if a == b:
        return a
    else:
        return(a + number_sum(a + 1, b))
```

Activity 3

Let $a =$ "computer" and $b =$ "science". Determine the following.

(a) $a[2]$

(b) $b[3 : 6]$

(c) $\text{len}(b)$

(d) b not in a

Activity 3 Solution

(a) $a[2] \implies$ "computer"$[2] \implies$ "m"

(b) $b[3:6] \implies$ "science"$[3:6] \implies$ "enc"

(c) len(b) $\implies$ len("science") $\implies 7$

(d) b not in $a \implies$ "science" not in "computer" $\implies True$

Activity 4

Write the program *number_sum* from Activity 2 using **while** loops.

Activity 4 Solution

Our approach for this program is very similar to our approach for Activity 2, where we will start with the value of a, and then add each following integer until we reach b, but using a **while** loop this time. First, we define a variable *sum* with the value 0, since we haven't added any numbers yet, to represent the sum of all the numbers. Then we want our loop to continue as long as $a \leq b$, because we don't want values greater than b. Within the loop, we want to increase the value of *sum* each iteration by the new value of a. Additionally, we want to increase the value of a by 1 each time so that we get each value and the loop eventually ends. After the loop has ended, we return the value of *sum*. This gives us the following code:

```
def number_sum(a, b):

    sum = 0

    while a <= b:

        sum = sum + a
        a = a + 1

    return sum
```

Activity 5

Write a program called *occurences* using **for** loops, that inputs a string of any length and a character (string of length 1), and outputs the number of times the character appears in the string.

As a bonus exercise, try also writing this program using **while** loops instead of **for** loops.

(For example: *occurences*("math circles", "c") outputs 2)

Activity 5 Solution

Our approach using either **for** or **while** loops is to go through each character in *string* and compare it with *char*. If the values are the same, then we will increase our counter, *total*, by 1. If they are not equal, then nothing happens.

With a **for** loop, the value of x will automatically equal a new character in *string* during each iteration, and will end the loop when there are no more characters. Similar to **while** loops, we define $total = 0$ before the loop, instead of within the loop, so that the value won't reset to 0 during each iteration. Within the loop, if the values of x and *char* are equal, then we increase the value of *total* by 1, and then we don't need any more conditions. After the loop has ended, we return the value of *total*, which is the number of times *char* appears in *string*. This gives us the following code:

```
def occurences(string, char):

    total = 0

    for x in string:

        if x == char:
            total = total + 1

    return total
```

With a **while** loop, we do nearly the same thing, the only difference is how we loop through the characters in *string*. Unlike **for** loops, **while** loops don't just automatically loop through sequences or strings, so we have to be more specific. We do this by defining a new variable i, which will represent the index of *string*, with the value 0 since the first character in strings have the index 0. We then run the loop as long as the value of i is less than the length of *string*, since the index of the last character of a string is 1 less than the length. This gives us the following

code:

```
def occurences(string, char):

    i = 0                          # index of the string
    total = 0

    while i < len(string):

        if string[i] == char:
            total = total + 1

        i = i + 1

    return total
```

Problem Set Solutions

1. Let $a =$ "cleveland", $b =$ "level" and $c =$ "thousand". Determine the following.
   (a) $\text{len}(a + b)$
   (b) c not in b
   (c) $a[6:9] == c[5:8]$
   (d) (b in a) and (c in a)

   Solution:
   (a) $\text{len}(a + b) \implies \text{len}(\text{"cleveland"} + \text{"level"}) \implies \text{len}(\text{"clevelandlevel"}) \implies 14$
   (b) c not in $b \implies$ "thousand" not in "level" $\implies True$
   (c) $a[6:9] == c[5:8] \implies \text{"cleveland"}[6:9] == \text{"thousand"}[5:8] \implies$ "and" == "and" $\implies True$
   (d) (b in a) and (c in a) $\implies$ ("level" in "cleveland") and ("thousand" in "cleveland") $\implies True$ and $False \implies False$

2. The grading system for public schools in Ontario is given below:

Percent (%)	Letter Grade
0 – 49	F
50 – 52	D–
53 – 56	D
57 – 59	D+
60 – 62	C–
63 – 66	C
67 – 69	C+
70 – 72	B–
73 – 76	B
77 – 79	B+
80 – 86	A–
87 – 94	A
95 – 100	A+

Write a program called *letter_grade* that inputs an integer percent (between 0 and 100), and outputs the corresponding letter grade.

Solution: Our approach for this program will be to use a conditional statement with 13 conditions (one to represent each percent interval). If the integer *percent* is within an interval, then we return the corresponding letter grade. This gives us the following code:

```
def letter_grade(percent):

    if 0 <= percent <= 49:
        return "F"

    elif 50 <= percent <= 52:
        return "D-"

    elif 53 <= percent <= 56:
        return "D"

    elif 57 <= percent <= 59:
        return "D+"

    elif 60 <= percent <= 62:
        return "C-"

    elif 63 <= percent <= 66:
        return "C"

    elif 67 <= percent <= 69:
        return "C+"

    elif 70 <= percent <= 72:
        return "B-"

    elif 73 <= percent <= 76:
        return "B"

    elif 77 <= percent <= 79:
        return "B+"

    elif 80 <= percent <= 86:
        return "A-"

    elif 87 <= percent <= 94:
        return "A"

    elif 95 <= percent <= 100:
        return "A+"
```

Note that since *percent* must be an integer between 0 and 100, we could replace the final condition with:

```
else:
    return "A+"
```

3. Suppose we want a program called *find_sevens* that inputs a positive 4-digit integer and outputs the number of times that 7 appears in the integer.

   (For example: *find_sevens*(7017) outputs 2, *find_sevens*(1234) outputs 0)

   (a) Write the program using exclusively conditional statements (no loops or recursion).

   (b) Write the program using loops.

   (c) Write the program using recursion.

Solution:

(a) The first part of the program is similar to *sum_digits* from Example 4 in the previous lesson, because we want to isolate each digit of the integer. After that, we have 4 individual conditional statements because we want compare the value of each digit to 7. If the values are equal, then we increase our counter, *sevens*, by 1. If the values are not equal, then we do nothing. At the end, we return the value of *sevens*, which is the number of times 7 appears in the integer *num*. This gives us the following code:

```
def find_sevens(num):

    ones = num % 10
    tens = (num // 10) % 10
    hundreds = (num // 100) % 10
    thousands = num // 1000

    sevens = 0

    if ones == 7:
        sevens = sevens + 1

    if tens == 7:
        sevens = sevens + 1
                                # all separate conditional statements
    if hundreds == 7:
        sevens = sevens + 1

    if thousands == 7:
        sevens = sevens + 1

    return sevens
```

(b) To write this program using loops, our approach is to cycle through each digit in the integer and compare them to 7. Since we are dealing with an integer, we will use a **while** loop. For each iteration of the loop, we take the digit in the ones place, which we already know how to do. If the digit is equal 7, then we increase the value of our counter *sevens* by 1. At the end of each iteration, we then take the integer quotient when the integer is divided by 10, so that we have a new digit in the ones place. The loop ends when the value of *num* is 0, and then we return *sevens*. This gives us the following code:

```
def find_sevens(num):

    sevens = 0

    while num != 0:

        digit = num % 10

        if digit == 7:
            sevens = sevens + 1

        num = num // 10

    return sevens
```

(c) To write this program using recursion, our approach is very similar to part (b). We take the digit in the ones place and then run it through our conditional statement. If the digit is equal to 7, we return 1 plus the result of *find_sevens* with the integer quotient of *num* divided by 10. If the digit is not equal to 7 (**else**), then we just return the result of *find_sevens* with the integer quotient of *num* divided by 10. The recursion ends when the value of *num* is 0, which is our base condition. This gives us the following code:

```
def find_sevens(num):

    digit = num % 10

    if num == 0:
        return 0

    elif digit == 7:
        return 1 + find_sevens(num // 10)

    else:
        return find_sevens(num // 10)
```

4. A **divisor** is an integer that divides into another integer with a remainder of 0 (e.g. 3 is a divisor of 12, but 5 is not). Write a program called *divisors* that inputs a positive integer and individually prints each of its positive divisors. (Hint: Use loops)

Solution: Our approach to this program will be to use a **while** loop. Since n must be a positive integer, we define the variable d, and as long as $d \leq n$, we will continue the loop. To see if each value of d is a divisor of n, we just have to see if the remainder of n divided by d is 0, and will print the value of d if it is. We include print(d) within the loop because we want each divisor of n to be printed. This gives us the following code:

```
def divisors(n):

    d = 1

    while d <= n:

        if n % d == 0:
            print(d)

        d = d + 1
```

5. Write a program called *number_of_vowels* using a **for** loop, that inputs a string of any length, and outputs the number of vowels within the string. For this program, we are not counting "y" as a vowel, just "a", "e", "i", "o" and "u". Note, that if we wanted to count "y" as a vowel, then the changes would be quite simple.

   (For example: *number_of_vowels*("math circles") outputs 3)

   Solution: Similar to other problems with **for** loops, we want to run through the entire string and check if each character is a vowel. If the character is a vowel, then we update our counter *total*, which is the value we'll return at the end. So, we define $total = 0$ before the loop, so that the value does not reset to 0 during each iteration. Within the loop, we have 5 conditional statements, one for each vowel. So each character gets checked, and we only increase the value of *total* if one of the conditions is true. This gives us the following code:

```
def number_of_vowels(string):

    total = 0

    for x in string:

        if x == "a":
            total = total + 1

        if x == "e":
            total = total + 1

        if x == "i":
            total = total + 1

        if x == "o":
            total = total + 1

        if x == "u":
            total = total + 1

    return total
```

We could simplify this code by condensing the inside of the loop to only 1 conditional statement. This is shown below:

```
def number_of_vowels(string):

    total = 0

    for x in string:

        if x =="a" or x =="e" or x =="i" or x =="o" or x =="u":
            total = total + 1

    return total
```

Bonus Question

6. The Fibonacci sequence is a sequence of numbers beginning with 0 and 1, where each following number in the sequence is the sum of the previous two numbers. For example, the third number in the sequence would be $0 + 1 = 1$, the fourth number in the sequence would be $1 + 1 = 2$, and so on. The first 10 numbers in the sequence are given below:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Suppose we want a program called $fibonacci$ that inputs a positive integer, n, and outputs the n^{th} number in the Fibonacci sequence.

(For example: $fibonacci(1)$ outputs 0, $fibonacci(7)$ outputs 8)

(a) Write the program using **while** loops.

(b) Write the program using recursion.

Solution:

(a) This problem can seem daunting so it's best to work in steps. We know the first two numbers in the Fibonacci sequence are 0 and 1, so we define two variables to represent them, $n1$ and $n2$.

We then can write a conditional statement to deal with all possible values of n. If the value of n is 1, then we return the first number in the sequence. Similarly, if the value of n is 2, then we return the second number in the sequence. Then, the **else** condition is for any values of $n \geq 3$, since we have to calculate the number, which will require a **while** loop.

We know that we have to define a variable i for the loop condition beforehand, and we set the value to 3 because that is the smallest possible value for n at this point. We want the n^{th} number in the sequence, so we set the loop condition as $i <= n$.

Within the loop, we define a temporary variable nth to represent the next number in the sequence. After this, we update the values of $n1$ and $n2$, and increase the value of i by 1, so we can repeat this process again.

After the loop ends, when $i > n$, we return the value of $n2$, which contains the n^{th} number of the sequence. This gives us the following code:

```
def fibonacci(n):

    n1 = 0           # define the first number in the sequence
    n2 = 1           # define the second number in the sequence

    if n == 1:
        return n1

    elif n == 2:
        return n2

    else:            # n >= 3
        i = 3        # define variable for while condition

        while i <= n:
            nth = n1 + n2  # temporary placeholder for new value
            n1 = n2        # update value of n1
            n2 = nth       # update value of n2

            i = i + 1

    return n2    # return n-th number in the sequence
```

(b) For recursion, the code for this program is short but quite complicated, because it requires 2 recursive commands.

Like all programs that use recursion, we have our conditional statement. In this case, we will 2 base conditions because we already know the first 2 numbers in the sequence. These two conditions, will be nearly identical to the ones above, where we return 0 if the value of n is 1, and we return 1 if the value of n is 2.

For the **else** condition, which is for $n >= 3$, we simply return the sum of $fibonacci(n-1)$ and $fibonacci(n-2)$. This is because any Fibonacci number after the first two numbers is equal to the sum of the previous two numbers. This gives us the following code:

```
def fibonacci(n):

    if n == 1:
        return 0

    elif n == 2:
        return 1

    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```