Grade 6 Math Circles
November 24th, 2021
Computer Science Part 2

This lesson is a continuation of Lesson 5 - Computer Science Part 1, so make sure you've gone over that lesson before this one, which is linked here https://www.cemc.uwaterloo.ca/events/mathcircles/2021-22/Fall/Junior6_Computer_Science_Part1_Nov17.pdf. Once again, this lesson deals exclusively with the Python programming language. These concepts exist in other programming languages but are written differently due to syntax. Additionally, the link to Python Tutor is included here for your use.

Conditional Statements

From the previous lesson, we have defined the comparison and boolean/logical operators, and have performed basic computations with them. But, we haven't yet covered much of their applications in computer programs. One of these applications is **conditional statements**.

The essence of conditional statements is that if a condition is *True*, then we perform some computation, and if the condition is *False*, then we perform some other computation. This is written in Python by using the code **if**, **elif** and **else**. Below is a general form for how conditional statements are written in Python.

```
if condition_1:
    # outcome

elif condition_2:
    # outcome

.
.
.

elif condition_n:
    # outcome

else:
    # outcome
```

The first line consists of **if**, followed by a condition that has a **bool** value (e.g. $a == b$, $a < b$, etc.). If the condition is *True*, then Python will run the following indented code (labelled as "*outcome*"). If the condition is *False*, then Python will skip over the following indented code. Additionally, each conditional statement contains only one **if**.

The **elif** keyword is Python's way of saying "if the previous conditions were not true, then try this condition". It works essentially the same as **if**, but it can only be used as a follow-up to **if**. Additionally, a conditional statement can contain any number of **elif**'s.

If every previous condition is *False*, then the indented code following **else** is run. We use **else** to catch any cases that aren't included in any of the previous conditions. Like **elif**, **else** can only be used as a follow-up to **if**, and **elif** (if there's any). Additionally, each conditional statement contains at most one **else**.

We can have multiple conditional statements in a program, but only one of the conditions can be run for each. Once a *True* condition is found, the following indented code is run, but then the remainder of the conditions are skipped, regardless if they are *True* or not. Observe the following code:

```
a = 7
b = 2

if a > b:
    print(a)

elif a >= b:
    print(b)
```

We see that both of the conditions $a > b$ and $a >= b$ are true. But, since the condition $a > b$ occurs first, we run the following indented code, **print(a)**, and then skip over the entire **elif** condition. Also, notice that no **else** statement was included. This is completely valid, as it is not required for a conditional statement to include **else**. The only keyword that is required for a conditional statement is **if**.

Example 1

Recall the lesson on Linear Relations, https://www.cemc.uwaterloo.ca/events/mathcircles/2021-22/Fall/Junior6_Linear_Relations_Nov3.pdf. Suppose we wish to write a program called *slope_class* that inputs a numerical value that represents the slope, m, of a line, and prints if the line is *increasing*, *decreasing* or *horizontal*. (We are excluding vertical lines for this example).

Solution 1

```python
def slope_class(m):

    if m > 0:
        print("The line is increasing")

    elif m < 0:
        print("The line is decreasing")

    else:
        print("The line is horizontal")
```

Note, that instead of writing the **else** statement at the end, we could have instead wrote:

```python
elif m == 0:
    print("The line is horizontal")
```

which would have covered all possible numerical values for m. The way the solution is currently written, if someone were to input a non-numerical value for m, then the program would print "The line is horizontal", which would be incorrect. This is why we say the value must be numerical.

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

We are also able to have conditional statements inside other conditional statements. They are called **nested conditional statements** and an example is given below.

```
a = 7
b = 2
c = 9

if a > b:
    if a > c:
        print(a)
    else:
        print(c)
else:
    print(b)
```

We are able to easily distinguish each condtional statement because of the indentations. Try running this code, and other similar code, through Python Tutor.

Recursion

Recursion is the process of defining something in terms of itself. In Python, this means that we define a program by using the program itself. This may sound a little paradoxical, but let's try to understand it with an example.

Before we get to the example, we first have a definition. The **factorial** of a postive integer x, is the product of x and every integer below it, down to 1. It is denoted by $x!$. For example,

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Example 2

Write a program called $factorial$ using recursion, that inputs a positive integer, and outputs the factorial of the integer.

Solution 2

```
def factorial(x):

    if x == 1:
        return 1
    else:
        return(x * factorial(x - 1))
```

Suppose we wanted to run $factorial(5)$. The process that Python goes through is shown below.

$$\begin{aligned}
factorial(5) &\implies 5 * factorial(4) && (5 == 1 \text{ is } False\text{, so we run } \textbf{else})\\
&\implies 5 * (4 * factorial(3)) && (4 == 1 \text{ is } False\text{, so we run } \textbf{else})\\
&\implies 5 * (4 * (3 * factorial(2))) && (3 == 1 \text{ is } False\text{, so we run } \textbf{else})\\
&\implies 5 * (4 * (3 * (2 * factorial(1)))) && (2 == 1 \text{ is } False\text{, so we run } \textbf{else})\\
&\implies 5 * (4 * (3 * (2 * 1))) && (1 == 1 \text{ is } True\text{, so we run } \textbf{if})\\
&\implies 5 * (4 * (3 * 2))\\
&\implies 5 * (4 * 6)\\
&\implies 5 * 24\\
&\implies 120
\end{aligned}$$

We see the program continues to refer back to itself with a smaller value each time, until it reaches $x = 1$. This is called the **base condition**. Every recursive program must have a base condition that stops the recursion or else the program calls itself infinitely. To prevent infinite recursions, Python limits the **depth** of recursion, which is the number of times a program can call itself.

Recursion can be very complicated to fully understand, so try inputting different numbers into $factorial(x)$ in Python Tutor and follow along as the result is evaluated. Try to avoid inputting large values (> 28) to avoid any problems.

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)

Advantages of Recursion

- Recursive programs make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.

Disadvantages of Recursion

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are inefficient as they take up a lot of memory and time.
- Recursive functions are hard to **debug** (fix).

String Operations

The previous lesson lacked many operations that can be used on **str** data types. The table below lists a few string operations that are commonly used in Python programming. Note, that a **substring** is a string that is a part of a larger string. For example, "ath circ" is a substring of "math circles", but "athcirc" is not, since it is missing the space between "h" and "c". For the following table, let $a =$"hello" .

Code	Description	Example
$[x]$	Returns the value at the index x in the string, which starts at index 0	$a[0] \implies$ "h"
$[x:y]$	Returns the substring starting at the index x and ending at the index y, not including y	$a[1:3] \implies$ "el"
len()	Returns the length of the string	len$(a) \implies 5$
in	Returns $True$ if the left-hand value is a substring of the right-hand value, and $False$ otherwise	"h" in $a \implies True$
not in	Returns $True$ if the left-hand value is not a substring of the right-hand value, and $False$ otherwise	"h" not in $a \implies False$

Activity 3

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

(a) $a[2]$

(b) $b[3:6]$

(c) len(b)

(d) b not in a

Loops

In computer science, a **loop** is a control-flow statement that is used to repeatedly execute a set of code as long as a condition is satisfied. Python has two main loop commands: **while** and **for**.

While Loops

The set-up of a **while** loop is very similar to that of the conditional statments. The keyword **while** is written, followed by some condition with a **bool** value. If the condition is $True$, then the following indented code is executed repeatedly as long as that condition remains true. If the condition is $False$, or becomes $False$, then the following indented code is skipped.

Example 3

Write the program $factorial$ from Example 2 using **while** loops.

Solution 3

```python
def factorial(x):

    result = 1

    while x >= 1:

        result = result * x
        x = x - 1

    return result
```

Aside from certain cases, **while** loops require any relevant variables to be defined before the loop. In the above example, the relevant variables are x and $result$, which are both defined prior to the loop. Had we defined $result = 1$ within the loop, then the value of $result$ would reset back to 1 during each iteration, instead of holding the product of all the values of x. To get a better understanding of this, try running this code through Python Tutor with $result$ defined within the loop.

Additionally, notice the last line of code within the loop, $x = x - 1$. It is extremely important that we include this, because we would receive an error if we did not. If we did not decrease the value of x during each iteration of the loop, then the loop would continue forever because $x >= 1$ would always be $True$. To prevent this from happening, we must always increment the value of the variable in the condition to ensure that the condition will eventually be $False$, thus ending the loop. We also write $x = x - 1$ to make sure that $result$ is being multiplied by the correct value each iteration, instead of the same value every time.

Activity 4

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

For Loops

The **for** loop works differently depending on the programming language used. In Python, a **for** loop is used for iterating over a sequence (like strings). With a **for** loop, we can execute a set of statements once for each character in the sequence. Unlike **while** loops, **for** loops do not require that variables are defined beforehand.

Example 4

Write a program called *characters* using **for** loops, that inputs a string, and prints each character of the string individually.

Solution 4

```
def characters(string):

    for x in string:
        print(x)
```

When we run the above program in Python Tutor, we see that the value of x is a character from *string* for each iteration of the loop. For example, if we ran *characters*("hello"), then $x =$ "h" for the first iteration, $x =$ "e" for the second iteration, and so on, until there are no more characters left. Run this code through Python Tutor for further understanding.

The number of iterations of a **for** loop is equal to the length of the sequence. So, in the above example, the number of iterations of the loop is len(*string*), which is 5 if *string* = "hello".

Unlike **while** loops, **for** loops do not run the risk of continuing forever, unless the sequence in question is infinite, in which case an error will occur. There are commands in Python we can implement to prevent this, such as **break**, which I will leave for you to explore on your own.

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)

Nested Loops

Similar to conditional statements, we are also able to write **nested loops** in Python, which are loops within loops. This means that we can have **for** loops within **while** loops, and vice versa.

On top this, we can also have loops within conditional statements, and conditional statements within loops, which is necessary for a few of the activity questions above.

Additionally, there is no limit to how many times we can do this, but in order to preserve the efficiency of your programs, it is advised that you try to minimize how "deep" you go with the nesting, especially with loops.

An additional resource for a more in-depth look at computer science can be found here:

https://cscircles.cemc.uwaterloo.ca/