

CptS 111 Laboratory 5: Loops, Optional Parameters, and Accumulators

Goal: In this lab we will review various implementations of loops. We will also introduce one new concept concerning optional parameters. Please be patient and carefully read all the material! There are only two “graded” tasks, neither of which requires much code. Only attempt to implement the code for these tasks after you have read and understood the material.

You are free to help your classmates and seek help from your classmates. Collaboration is strongly encouraged. However, please make sure the help you provide takes the form of explaining things and you do not merely provide the answers/solutions.

For this lab you should ensure the TA notes your completion of the following “graded” tasks:

Task 3: Joining a `list` of Strings.

Task 5: Another Accumulator Exercise.

Task 1: Review of Definite Loops.

You have already seen how the `range()` function can be used to create lists of a specified length. You have also seen examples in which the list produced by the `range()` function can be used in a `for`-loop to ensure the loop iterates a specified number of times. For example, the following loop prints `hello` nine times:

```
1 for i in range(9):
2     print("hello")
```

Note that this definite loop (or *counted* loop) has the loop variable `i` in the header.¹ However, we don’t do anything with this variable. This is perfectly fine. In some sense `i` is used as a counter but we never display or use this counter in an expression. You should further note that there is nothing special about the name `i` for this loop variable. Although `i` (or `j`) is often used for the loop variable in counted loops, any valid identifier can be used.

If we want to number each line of output, we certainly could. Here is a slightly modified version of the loop that generates a line number together with the word `hello`.

```
1 for i in range(9):
2     print(i, "hello")
```

You should enter this in Python and ensure it works as claimed. If we prefer to have the line numbers go from 1 to 9 (instead of 0 to 8), then we could enter either of the following

¹The `list()` function has previously been used to display the list of values that `range()` ultimately produces. However, the `list()` function should *not* be used with the `range()` function when the `range()` function appears in the header of a `for`-loop.

```

1 for i in range(9):
2     print(i+1, "hello")

```

or

```

1 for i in range(1, 10):
2     print(i, "hello")

```

In addition to a counted loop, where the loop variable may not be of interest, `for`-loops can also be constructed so that the loop variable takes on the value of the elements of a list. The following code illustrates this where `item` is the loop variable:

```

1 >>> xlist = ["first", 2, 3.0, 2 + 2, "last"]
2 >>> for item in xlist:
3     ...     print(item)
4     ...
5 first
6 2
7 3.0
8 4
9 last

```

Now, what if we want to display each element's index together with the value of the element? Can you do this using the same header as in the last `for`-loop, i.e., can you do it without explicitly using indexing?² Think about this for a moment. The answer is yes, we can use that same header if we add a bit of code to keep track of the index. Here is an example:

```

1 >>> xlist = ["first", 2, 3.0, 2 + 2, "last"]
2 >>> index = 0
3 >>> for item in xlist:
4     ...     print(index, item)
5     ...     index = index + 1
6     ...
7 0 first
8 1 2
9 2 3.0
10 3 4
11 4 last

```

In line 2, outside the loop, we initialize the variable `index` to 0. This variable will serve as an *accumulator*. Here we are simply accumulating a count: for each pass of the loop we reset `index` to its previous value plus 1.

Alternatively, we can use the `range()` function in the header of the `for`-loop to generate all the values of a given list. Recall that `len()` returns the length of its argument. Thus the following is functionally equivalent to the code above:

²Recall the first item has an index of zero. We can access items of a list by placing the index of the desired element in square brackets following the list itself. So, for example, `xlist[1]` is the second element of the list `xlist`.

```

1 >>> xlist = ["first", 2, 3.0, 2 + 2, "last"]
2 >>> for i in range(len(xlist)):
3     ...     print(i, xlist[i])
4     ...
5 0 first
6 1 2
7 2 3.0
8 3 4
9 4 last

```

As you can see, there are different ways in which we can use a `for`-loop. The previous two loops produce the same result, but the second implementation, in which the indices came directly from the `range()` function in the header, would generally be considered the preferable implementation in that it has fewer lines of code and is consistent with the idioms Python programmers often use.³ All of these `for`-loops are definite loops in that one can say in advance how many times the body of the loop will execute. We can categorize the constructions we have seen as follows:

- Counted loops that use the `range()` function to ensure the body of the loop is executed a given number of times.
- Loops designed to access the elements of a list (or tuple) where the list is given as the iterable in the header. In this case the loop variable corresponds to the values of the elements of the list.
- Loops designed to access the elements of a list (or tuple), but the `range()` function is given as the iterable. In this case the loop variable serves as the index to access the desired elements of the list.

If you understand this, you may move on to the next task. If you have any questions, now is the time to ask the TA!

Task 2: Optional Parameters.

Python provides many built-in functions. The first function we used was the built-in function `print()`. The `print()` function possesses a couple of interesting features that we don't yet know how to incorporate into our own functions: `print()` can take a variable number of parameters and it also accepts optional parameters. The optional parameters are `sep` and `end` which specify the string used to separate arguments and what should appear at the end of the output, respectively. To demonstrate this, you can enter the following statements:

```

1 >>> # Separator defaults to a blank space.
2 >>> print("Hello", "World")

```

³As mentioned in Chap. 1 of the textbook, there are many different ways of coding a particular task. In computer languages, as with natural languages, programmers develop *idioms*, i.e., “characteristic modes of expression.” When you start with a language, it is hard to know what the idioms are. That knowledge comes with experience. Then, sticking with the idioms tends to enhance the readability of the code since idioms are inherently familiar.

```

3 Hello World
4 >>> # Explicitly set separator to string "--*--".
5 >>> print("Hello", "World", sep="--*--")
6 Hello-*--World
7 >>> # Issue two separate print() statements. (Can put multiple
8 >>> # statements on one line if they are separated by semicolon.)
9 >>> # By default, print() terminates its output with a newline.
10 >>> print("Why, Hello"); print("World!")
11 Why, Hello
12 World!
13 >>> # Override the default separator and line terminator with the
14 >>> # optional arguments of sep and end.
15 >>> print("Why,", "Hello", sep="^v^v", end="--*--"); print("World!")
16 Why,^v^vHello-*--World!

```

We create functions in Python using a `def` statement. In the header, enclosed in parentheses, we include the list of formal parameters for the function. Python provides several different constructs for specifying how the parameters of a function are handled. We can, in fact, define functions of our own which accept an arbitrary number of arguments and employ optional arguments. Exploring all the different constructs would take quite a bit of time and the use of multiple arguments isn't currently of interest. However, creating functions with optional arguments is both simple and useful. Thus, let's consider how one obtains a function with optional parameters.

First, let's define a function without optional parameters that squares its argument. The following provides the definition and then invokes the function with an argument of 10:

```

1 >>> def square(x):
2     ...     return x * x
3     ...
4 >>> square(10)
5 100

```

As described in the textbook, when we write `square(10)`, the *actual* parameter is 10. The actual parameter is assigned to the formal parameter and then the body of the function is executed. So, in this example, where we have `square(10)`, it is as if we had issued the statement `x=10` and then executed the body of the `square()` function.

If we so choose, Python will let us explicitly establish the connection between the formal and actual parameters when we call a function. Consider the following where the `square()` function is defined exactly as above:

```

1 >>> def square(x):
2     ...     return x * x
3     ...
4 >>> square(x=10)
5 100

```

Note carefully what appears in the argument when the `square()` function is called in line 4. We

say that `x` should be set to 10. This assignment is performed, then the body of the function is executed, and, finally, the return value is 100 which is shown on line 5. In practice, for this simple function, there is really no reason to do. However, some functions have many arguments, some of which are optional and some of which are not. For those functions, explicitly assigning values to the parameters can aid readability.

What happens if we use a different variable name (other than `x`) when we invoke the `square()` function? Let's see:

```
1 >>> def square(x):
2     ...     return x * x
3     ...
4 >>> square(y=10)
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7 TypeError: square() got an unexpected keyword argument 'y'
```

In line 4 we tried to assign the value 10 to the variable `y`. But, the `square()` function doesn't have any formal parameter named `y`, so this produces an error (as shown in lines 5 through 7). We have to use the formal parameter name that was used when the function was defined (which, in this case is `x`).

Now, with that background out of the way: To create a function with one or more optional parameters, we simply assign a default value to the corresponding formal parameter(s) in the header of the function definition. An example helps illustrate this.

Let's create a function called `power()` that raises a given number to some exponent. The user can call this function with either one or two arguments. The second argument is optional and corresponds to the exponent. If the exponent is not given explicitly, it is assumed to be 2, i.e., the function will square the given value. The first argument is required (i.e., not optional) and represents the number that should be raised to the given exponent. The following shows how to implement the `power()` function. Notice that in the header of the function definition (line 1), we simply assign the default value of 2 to the formal parameter `exponent`.

```
1 >>> def power(x, exponent=2):
2     ...     return x ** exponent
3     ...
4 >>> power(10)      # 10 squared, i.e., 10 ** 2.
5 100
6 >>> power(3)      # 3 squared, i.e., 3 ** 2.
7 9
8 >>> power(3, 0.5) # Square root of 3, i.e., 3 ** 0.5.
9 1.7320508075688772
10 >>> power(3, 4)   # 3 ** 4
11 81
12 >>> power(2, exponent=3) # 2 ** 3
13 8
14 >>> power(x=3, exponent=3) # 3 ** 3
15 27
```

```

16 >>> power(exponent=3, x=5) # 5 ** 3
17 125
18 >>> power(exponent=3, 5) # Error!
19 File "<stdin>", line 1
20 SyntaxError: non-keyword arg after keyword arg

```

Lines 4 through 7 show the argument is squared when `power()` is called with a single argument, i.e., the value of the argument is assigned to the formal parameter `x` and the default value of 2 is used for `exponent`. Lines 8 through 11 show what happens when `power()` is called with two arguments. The first argument is assigned to `x` and the second is assigned to `exponent`, thus overriding `exponent`'s default value of 2. In lines 8 and 10, the assignment of actual parameters to formal parameters is based on position—the first actual parameter is assigned to the first formal parameter and the second actual parameter is assigned to the second formal parameter (this is no different than what you have previously observed with multi-parameter functions). So, for example, based on the call in line 8, 3 is assigned to `x` and 0.5 is assigned to `exponent` (which yields $\sqrt[3]{3}$).

In line 12 we see the function can be called with the optional parameter explicitly “named” in an assignment statement (thus optional parameters are sometimes called *named parameters*). Line 14 shows that, in fact, both parameters can be named when the function is called. Keep in mind, however, that `x` is *not* an optional parameter—a value must be provided for `x`. If one names all the parameters, then the order in which the parameters appear is not important. This is illustrated in line 16 where the optional parameter appears first and the required parameter appears second (here the function calculates 5^3). Finally, line 18 shows that we cannot put an optional parameters before an unnamed required parameter.

Let us consider one more example in which a function calculates the y value of a straight line. Recall that the general equation for a line is

$$y = mx + b$$

where x is the independent variable, y is the dependent variable, m is the slope, and b is the intercept (i.e., the value at which the line crosses the y axis). Let's write a function called `line()` that, in general, has three arguments corresponding to x , m , and b . The slope and intercept will be optional parameters where the slope has a default value of 1 and the intercept has a default value of 0. The following illustrates both the construction and use of this function (please read the comments in the code!):

```

1 >>> def line(x, m=1, b=0):
2 ...     return m * x + b
3 ...
4 >>> line(10)           # x=10 and defaults of m=1 and b=0.
5 10
6 >>> line(10, 3)       # x=10, m=3, and default of b=0.
7 30
8 >>> line(10, 3, 4)    # x=10, m=3, b=4.
9 34
10 >>> line(10, b=4)    # x=10, b=4, and default of m=1.

```

```
11 14
12 >>> line(10, m=7) # x=10, m=7, and default of b=0.
13 70
```

You should enter these functions and make sure you understand what is happening. If you understand this, please move on to the next task. If not, ask questions!

Task 3: Joining a list of Strings.

We want to write a function called `join()` which has one required parameter and one optional parameter. The first parameter is a list, each element of which should be a string. This parameter is required. The second parameter is a string. It is optional and has a default value of a single blank space. This optional parameter is named `sep`. The `join()` function returns a single string consisting of the concatenation of all the elements in the list with the elements separated by `sep`.

Here is a demonstration of the correct behavior of this function.

```
1 >>> name = ["Zippy", "the", "Pinhead"]
2 >>> join(name) # Use default separator, i.e., a blank space.
3 'Zippy the Pinhead'
4 >>> join(name, sep="") # Set separator to the empty string.
5 'ZippythePinhead'
6 >>> join(name, sep="-*-") # Use a three-character separator.
7 'Zippy-*-the-*-Pinhead'
8 >>> name = ["Madonna"]
9 >>> join(name) # Ensure function works with single string in list.
10 'Madonna'
11 >>> join(name, sep="-*-") # Still works...
12 'Madonna'
```

In this code the `name` list defined in line 1 contains three strings. When this is passed as an argument to the `join()` function in line 2, the return value is a single string with all the elements of `name` concatenated, with blank spaces between the values of the elements. In line 4 the separator is set to the empty string. The result on line 5 is the pure concatenation of the elements of `name` with no separators. In line 6 the `join()` function is called with a separator of `-*-`. Line 7 shows the resulting string. In line 8 a list is created with a single string. Lines 9 through 12 show that the `join()` function behaves properly for this single-element list (i.e., the separator is not erroneously added at the end).

As an aside, recall that actual concatenation of strings is realized by putting the plus sign between two strings. For example, `"Hi" + "Ho"` creates the string `"HiHo"`.

Now try to implement the `join()` function. Don't be afraid to make mistakes, but be sure to learn from them. If you get to a point where you feel stuck, you can continue reading. If you systematically implement the steps in the following recipe, you should obtain the correct function.

1. The header of the function definition should specify that `join()` has two parameters: a list

and a string. The string parameter is named `sep` and, since it is optional, it is set to the default value of a single blank space (don't forget to enclose the space in quotes).

2. Following the header, initialize an accumulator to be the empty string.
3. Implement a `for`-loop that will loop over all the elements of the list *except* the last one. Use the `range()` function in the header to obtain the desired indices for elements of the list (again, exclude the last element of the list).
4. In the body of the loop, concatenate an element of the list to the accumulator and concatenate the separator to the accumulator. (You can do this in either one statement or two.)
5. Outside the loop, concatenate the last element of the list to the accumulator.
6. Return the accumulator.

Once you think your function is working properly, please demonstrate your code to the TA. If he agrees it is correct, please move on to the next task.

Task 4: Summation Notation and Accumulators.

Before turning to the next task, we want to review summation notation (which hopefully you have seen in your math courses) and the use of an “accumulator” in Python.

First, let us assume we want to calculate the following sum

$$1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \dots + \frac{1}{100} \quad (1)$$

After you stare at this for a bit you should realize that the individual terms in the sum are given by $1/k^2$ where k takes on values from 1 to 10. We can write this using summation notation as follows

$$\sum_{k=1}^{k=10} \frac{1}{k^2} \quad (2)$$

The symbol Σ is the uppercase Greek letter Sigma (this is the Greek equivalent of the letter S which you can think of as standing for “sum”). In the example above, the symbol Σ tells us we should sum the expression to the right side of the symbol after we plug in values of the index k starting at 1 and stopping at 10. We call the index k the *summation variable*. In general, the starting value of the index appears below the Σ while the ending value appears above Σ . Sometimes the summation is supposed to go on forever and in this case we often write the symbol for infinity (∞) above Σ .

Consider the following summation

$$\sum_{k=0}^{\infty} \frac{k+1}{k^3 + 2k^2 + 3k + 4} = \frac{1}{4} + \frac{2}{5} + \frac{3}{26} + \frac{4}{58} + \frac{5}{112} + \dots \quad (3)$$

In this case the index k starts at zero and goes to infinity. Since there are an infinite number of terms in this summation, we obviously can't sum all the terms. However, since the terms get smaller and

smaller (and since the summation does converge to a finite value), we can approximate the sum by summing a finite number of terms.

Let's assume we want to sum the first 15 terms in this series (i.e., the terms associated with k from 0 to 14). The following code shows one way in which we might calculate this summation using Python.

```
1 >>> # Define the function that yields the individual terms in
2 >>> # the summation.
3 >>> def f(k):
4     ...     return (k+1)/(k**3 + 2 * k**2 + 3 * k + 4)
5     ...
6 >>> # Initialize an accumulator called 'sum' to zero. Add
7 >>> # successive terms to this.
8 >>> sum = 0
9 >>> for k in range(15):
10    ...     sum = sum + f(k)
11    ...     print(k, f(k), sum)
12    ...
13 0 0.25 0.25
14 1 0.2 0.45
15 2 0.115384615385 0.565384615385
16 3 0.0689655172414 0.634350132626
17 4 0.0446428571429 0.678992989769
18 5 0.0309278350515 0.70992082482
19 6 0.0225806451613 0.732501469982
20 7 0.0171673819742 0.749668851956
21 8 0.0134730538922 0.763141905848
22 9 0.0108459869848 0.773987892833
23 10 0.00891410048622 0.782901993319
24 11 0.00745341614907 0.790355409468
25 12 0.00632295719844 0.796678366667
26 13 0.00543056633049 0.802108932997
27 14 0.00471401634192 0.806822949339
```

This code starts by defining a function $f()$ which computes the individual terms in the summation for a given value of the index k . Following the definition of this function, in line 8 we initialize a variable called `sum` to zero. This variable will be used as an *accumulator*. As the name implies, `sum` is what we will use to build up the sum. The `for`-loop, whose header is in line 9, is such that the loop variable `k` takes on the values from 0 to 14. The loop variable `k` corresponds to the summation variable k in the equations above. For each value of `k` we take the current value of `sum` and add to that $f(k)$ (and store the result back in `sum`). The `print()` statement in line 11 is in the `for`-loop and shows the value of `k`, the term corresponding to the value of `k` (i.e., $f(k)$), and the current sum up to the value of `k`.

Make sure you understand this notation and this code. Once you are comfortable with it, feel free to move on. However, if you have questions, think about them a bit and then ask the TA (or one of your classmates) for help!

Task 5: Another Accumulator Exercise.

The previous task provides code that sums terms in a series that only depends on the summation variable. The following is another example of a series where the terms in the series only depend on the summation variable (or summation index, which is k in the following)

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{100} = \sum_{k=1}^{k=100} \frac{1}{k} \quad (4)$$

Often “ $k =$ ” is omitted from the term above Σ . So, in the expression on the right, one can just write 100 above Σ to indicate the ending value of the index. In the following we want to generalize this further and introduce an additional variable.

Functions that involve the sum of several terms often also employ Σ -notation. Here are three examples:

$$f(x) = 1 + x + x^2 + x^3 + x^4 + x^5 + x^6 = \sum_{k=0}^6 x^k \quad (5)$$

$$g(x) = x + \frac{1}{2}x^3 + \frac{1}{3}x^5 + \frac{1}{4}x^7 + \frac{1}{5}x^9 = \sum_{k=1}^5 \frac{1}{k} x^{2k-1} \quad (6)$$

$$h(x) = \frac{x-1}{x^2} + \frac{x-2}{x^3} + \frac{x-3}{x^4} + \dots + \frac{x-k_{\max}}{x^{k_{\max}+1}} = \sum_{k=1}^{k_{\max}} \frac{x-k}{x^{k+1}} \quad (7)$$

In the last function $h(x)$ we haven’t said explicitly what the ending value is for the index k . We have just labeled it k_{\max} and perhaps we want to consider what happens when it takes on different values. So, in a sense h is really a function of two things: x and k_{\max} . Thus, it would be more descriptive to write this function as $h(x, k_{\max})$.

Now, let’s return to the expression shown in Eq. (4). Let’s calculate this sum in Python. We certainly do not want to enter all the terms by hand. Instead, we will use a `for`-loop to do most of the work for us. We start by initializing a variable that is an “accumulator” which represents the sum. Since, when we start, nothing has been added to the sum, we initialize it to zero. Let’s call this variable/accumulator `sum`. Then, we write a `for`-loop that adds one term to the sum for each iteration of the loop. Thus, Eq. (4) can be calculated using:

```
1 sum = 0.0
2 for i in range(1, 101):
3     sum = sum + 1 / i
```

Now, consider the following function (which is a slight variation of the f function given in Eq. (5)):

$$f(x, k_{\max}) = 1 + x + x^2 + x^3 + \dots + x^{k_{\max}} = \sum_{k=0}^{k_{\max}} x^k \quad (8)$$

Note that f has two arguments, x and k_{\max} .

For the final task you must write a Python function that calculates $f(x, k_{\max})$. This function should take two arguments and return the result of the summation. It does not print anything nor does it prompt for any input—all the input it needs is provided by the two arguments. The following demonstrates the proper behavior of this function:

```
1 >>> f(1.0, 0)           # x = 1.0; k_max = 0
2 1.0
3 >>> f(1.0, 1)           # x = 1.0; k_max = 1
4 2.0
5 >>> f(1.0, 10)          # x = 1.0; k_max = 10
6 11.0
7 >>> f(2.2, 10)          # x = 2.2; k_max = 10
8 4868.485845094404
9 >>> f(-2.2, 10)         # x = -2.2; k_max = 10
10 1826.3071919104018
```

To implement this function you should use an accumulator and a `for`-loop. Since you are building up a sum, you will need an accumulator that must be initialized before the `for`-loop. For this particular task the terms of the series are simple enough that you do not need to define an auxiliary function as was discussed in the previous task. Don't forget that exponentiation is realized with `**`. (You can use exponentiation in your solution. However, there is a solution that can be implemented without exponentiation! If you have spare time, you should see if you can implement this.)

Once you are satisfied that your function is behaving properly, show it to the TA and demonstrate that it does indeed work. When the TA agrees that it works, you are done!