CptS 111 — Lab #8

Iterating `for`-loops, `while`-loops, and More

Learning Objectives:

- Write iterating `for`-loops with lists and dictionaries
- Write `while`-loops
- Use the `break` and `continue` commands
- Use the `enumerate()` function

Prerequisites:

- Basics of iterating `for`-loops
- Basics of `while`-loops

Task 1: `while`-loops and `break` and `continue` with loops

In general, it's best to minimize the use of the `break` and `continue` commands because they can make our code more difficult to understand. This is because they can interrupt the flow of a program. However, sometimes they can be used very effectively, e.g., in PA #5, `break` works well with the `while`-loop used in the first part of the assignment. Note that unlike `for`-loops, `while`-loops require a test. If the test is **`True`**, the loop body is executed; if the test is **`False`**, the loop body is skipped. Let's try some examples in an IDLE Shell window. As usual, read the comments and type the text in boldface.

```
>>> # Let's start with while-loops.  It's best to use while-loops only
>>> # when no other loop works as well, e.g., when we want a certain
>>> # set of commands to run, but we don't know how many times.
>>> volunteers = []
>>> entry = '-'
>>> while entry != '':
        entry = input('Enter the name of a volunteer [return to stop]: ')
        volunteers.append(entry)

Enter the name of a volunteer [return to stop]: Sam
Enter the name of a volunteer [return to stop]: Ann
Enter the name of a volunteer [return to stop]: Mohammed
Enter the name of a volunteer [return to stop]: Maria
Enter the name of a volunteer [return to stop]: Yan
Enter the name of a volunteer [return to stop]: Saana
Enter the name of a volunteer [return to stop]: Matt
Enter the name of a volunteer [return to stop]:
>>> volunteers
['Sam', 'Ann', 'Mohammed', 'Maria', 'Yan', 'Saana', 'Matt', '']
>>> len(volunteers)
8
>>>
```

```
>>> # However, as you can see from this example, our list contains an
>>> # empty string (depending on which version of Python3 you're using
>>> # the last command will result in 7 or 8).  We'll see how we can
>>> # fix the list shortly, but first let's try another while-loop
>>> # example.
>>> steps = 0
>>> step_goal = 10000
>>> while steps <= step_goal:
        if steps < step_goal:
            print('Keep on moving!')
        else:
            print('Well done!  Another step goal met.')
        steps += 1000
>>>
>>> # As we see from this example, we can use a conditional in the body
>>> # of our while-loop.  In fact, conditionals are often used in the
>>> # body of a while-loop.  Now let's return to our earlier example.
>>>
>>> # We can use a conditional and the break command to remove the
>>> # empty string in our list.
>>> volunteers = []
>>> entry = '-'
>>> while True:
        entry = input('Enter the name of a volunteer [return to stop]: ')
        if entry == '':
            break
        volunteers.append(entry)

Enter the name of a volunteer [return to stop]: Sam
Enter the name of a volunteer [return to stop]: Ann
Enter the name of a volunteer [return to stop]: Mohammed
Enter the name of a volunteer [return to stop]: Maria
Enter the name of a volunteer [return to stop]: Yan
Enter the name of a volunteer [return to stop]: Saana
Enter the name of a volunteer [return to stop]: Matt
Enter the name of a volunteer [return to stop]:
>>> volunteers
['Sam', 'Ann', 'Mohammed', 'Maria', 'Yan', 'Saana', 'Matt']
>>> len(volunteers)
7
>>>
>>> # The break command tells Python to ignore the remaining commands
>>> # and leave the while-loop.  Thus, the empty string isn't added to
>>> # the list.  Important: Note that we reinitialized the list as
>>> # an empty list.  If we hadn't done this, Python would have added
>>> # the names to the original list of volunteers because lists are
>>> # mutable, i.e., can be changed!  However, we only have to do this,
>>> # i.e., reinitialize the list, volunteers, because we're using an
>>> # IDLE Shell window.
```

```
>>>
>>> # Care must always be used with while-loops because a simple
>>> # mistake can lead to an infinite loop, i.e., the while-loop will
>>> # loop ``forever.''
>>>
>>> # Both the break and continue commands can also be used with for-
>>> # loops.
>>> for i in range(10):
        team = input('Enter team name: ')
        if team != 'Cougs':
            continue
        else:
            print('Go, Cougs!')
            break

Enter team name: Huskies
Enter team name: Beavers
Enter team name: Ducks
Enter team name: Cougs
Go, Cougs!
>>>
>>> # If the condition isn't met in the example above, the continue
>>> # statement tells Python to ignore the remaining commands and
>>> # return to the beginning of the for-loop.  When the condition
>>> # is met, the break command tells Python to stop executing the
>>> # loop, i.e., to break out of it.  However, we don't need to worry
>>> # about infinite loops when using a for-loop.
>>>
>>> # In the example above, the loop will stop without producing any
>>> # result after 10 iterations.  This construct is often used in
>>> # real programming applications, i.e., a limit will be provided
>>> # for the number of iterations, and if a criterion is not met,
>>> # the coder will add a print statement letting the user know.
```

After you've completed this task, show your work to your TA to get credit. Keep your IDLE session open for the next task.

Task 2: Iterating for-loops, the `enumerate()` function, and more!

In Lab #7, we covered counting for-loops. In this lab, we'll cover iterating `for`-loops. In fact, all loops are iterating loops because they iterate (repeat) the commands in a loop. However, I've found it helpful to divide `for`-loops into counting and iterating loops. **Counting** `for`-loops always use the **`range()`** function and a counter such as **`i`** as the loop variable in the header while **iterating** `for`-loops use an **iterable**, e.g, a list or a tuple, in the header with a suitable name for the loop variable. They can't always be used interchangeably, but often they can, and it's a matter of a coder's preference which to use. I prefer using iterating `for`-loops when I can because I think they're easier to follow. So let's get started!

```
>>> # The template for an iterating for-loop is:
```

```
      for <loop_variable> in <iterable>:
          <loop_body>

>>> # As you can see, the headers for counting and iterating for-loops
>>> # differ.  In an iterating for-loop, there is no explicit counter.
>>> # Instead each value in the iterable (a string, list, dictionary,
>>> # tuple, or set) is used in the loop itself.  It's good practice
>>> # to give the loop variable a name that makes sense rather than
>>> # using, e.g., just 'i'.  For example, we might use the following:

      for name in names:
          <loop_body>

      for num in nums:
          <loop_body>

      for ch in word:    # Don't use chr!  (Why?)
          <loop_body>

>>> # Now let's look at some examples.  Please note that we'll use
>>> # lists and dictionaries in these, but the same format used for
>>> # lists can be used for strings, tuples, and sets.
>>> names = ['Sam', 'Mohammed', 'Maria', 'Yan']
>>> for name in names:
        print(f'Hello, {name}.')

>>> # In this first example, we used the same list as in Lab #7, but
>>> # rather than using the range() function, we used an iterating for-
>>> # loop with a list iterable.  Python used the first element from
>>> # names in its first iteration, the second element in names in its
>>> # second iteration, and so on.  In Lab #7, we used ``for i in
>>> # range(len(names))'' in the header and ``names[i]'' in the print
>>> # statement which was more complicated.  However, we considered
>>> # four other counting for-loop examples in Lab #7, and none of
>>> # these would work with an iterating for-loop because there is
>>> # no iterable.  Let's look at another list example.
>>> models = []
>>> makes = ['Tesla', 'Ford', 'Hyundai', 'Toyota', 'Honda', 'Jaguar']
>>> for make in makes:
        model = input(f'Enter a model for a {make}: ')
        models.append(model)

Enter a model for a Tesla: Model Y
Enter a model for a Ford: Escape
Enter a model for a Hyundai: Kona
Enter a model for a Toyota: RAV4
Enter a model for a Honda: CR-V
Enter a model for a Jaguar: E-PACE
```

```
>>> models
>>> # In the example above, we've used two lists, one as the iterable
>>> # and the other that we create in the iterating for-loop.  Outside
>>> # the loop, we have to first initialize the list we want to create
>>> # in the loop, and we also define the 'makes' list.  Note the use
>>> # of string formatting in the input() function.  Pretty cool!
>>>
>>> # Next let's turn to dictionaries, but first let me show you
>>> # another reason why I love Python!
>>> cars = dict(zip(makes, models))
>>> cars
>>>
>>> # Wow!  Python makes coding so easy!  We can use the zip()
>>> # function to zip two iterables together to create a zip type.
>>> res = zip(makes, models)
>>> type(res)
>>>
>>> # Then we can use list(), tuple(), set(), or dict() with the zip
>>> # file to create data structures of these types.  However, we have
>>> # to create the zip file each time after it's used or else we'll
>>> # end up with an empty container.
>>> res = zip(makes, models)
>>> l_cars = list(res)    # That's an 'el', not a one!
>>> l_cars
>>> res = zip(makes, models)
>>> t_cars = tuple(res)
>>> t_cars
>>> res = zip(makes, models)
>>> s_cars = set(res)
>>> s_cars
>>>
>>> # Note that we get a list of tuples, a tuple of tuples, or a set
>>> # of tuples with a zipped object.  Only dict() doesn't result in
>>> # tuples.  Now let's use an iterating for-loop with our dictionary
>>> # 'cars'.  We can use three different approaches.

      for key in dictionary.keys():
      for key in dictionary.values():
      for key, value in dictionary.items():

>>> # Let's try each of these.
>>> for make in cars.keys():
        print(f'What do you think of {make}s?')
>>>
>>> for model in cars.values():
        print(f'Are you familiar with the {model}?')
>>>
>>> for make, model in cars.items():
        print(f``I'm thinking of buying a {make} {model}.'')
```

```
>>> # So using dictionaries as iterables is a little more complicated
>>> # than using lists, but it's still not difficult.  However, you
>>> # have to expend some effort to memorize these concepts or at
>>> # least enough effort to know what you can do even if you can't
>>> # remember how to do it (then you can look it up!).
>>>
>>> # We'll end this task with a quick look at the enumerate()
>>> # function which can be used in PA #7.  As with the range()
>>> # function, enumerate() is used as a header in a for-loop.  It
>>> # can be used with any iterable in Python.  It can be used to
>>> # create itemized lists of any iterable.  Let's start with our
>>> # volunteers list.
>>> for i, name in enumerate(volunteers):
        print(f'{i+1}. {name}')
>>>
>>> # Notice that we had to add a 1 to i.  Otherwise, the list would
>>> # have started with a 0.  We can use the same header for tuples
>>> # and sets, but for dictionaries, the header is more complicated.
>>> for i, (make, model) in enumerate(cars.items()):
        print(f'{i+1}. {make:7}:  {model}')
>>>
>>> # Again, we have to use cars.items() in order to retrieve key-
>>> # value pairs, but we also have to group them as a tuple in the
>>> # header.  Otherwise, Python thinks there are three objects, but
>>> # enumerate() can only use two objects.  Also, notice that we used
>>> # 7 in the f-string format specifier in order to align all the
>>> # colons that follow the keys.  This 7 was chosen by trial and
>>> # error.  There's nothing magical about it!
```

After you've completed this task, show your work to your TA to get credit.

Task 3: CptS 111 Fitness App

This task uses a lot of the commands and constructs demonstrated in Task 2. It requires coding four functions as described below in the order in which they should appear. The program prompts the user for the number of steps they've walked each day of the week, creates a dictionary from two lists, uses the dictionary to determine the maximum number of steps walked and the day they were walked, finds the average number of steps walked, and prints all the results to stdout (= standard output), your monitor in this case, as shown in the examples below. Save your program as `lab8_t3.py` in your Lab8 folder under CS111. Before beginning your program in an IDLE Editor window, do the example below in your IDLE Shell window, and figure out how it works. It should help you with one of the functions.

```
>>> nums = [5, 3, 2, 17, 11, 7, 13] # Create a list of positive integers
>>> max_num = 0                     # Initialize the max number
>>> for num in nums:
        if num > max_num:           # If current number larger than
            max_num = num           # max, make it the new max value
```

```
>>> print(f'The largest number is {max_num}.')
```

Now we can begin! Read the `main()` function first and consider using the function stub `pass` to help you develop your program.

- **`get_steps()`**: This non-void function has a single parameter, the list of days in the week. It initializes an empty list and then uses an iterating `for`-loop to prompt and obtain the number of steps a user walked as shown in the examples below. It appends each number to the list that was initialized and returns this list to `main()`. Hint: Recall that you can use string formatting in the argument of the `input()` function. Be sure you know how to create this prompt! Also, choose a good name for your list.
- **`max_steps_day()`**: This non-void function has one parameter, the dictionary of days (keys) and number of steps (values). It initializes values for the maximum number of steps (see example before on previous page!) and the day of the maximum number of steps. It then finds the actual maximum number of steps and the day of the maximum number of steps using an iterating `for`-loop with the dictionary as the iterable and returns both values to `main()`.
- **`print_results()`**: This void function has five parameters: the dictionary of days (keys) and number of steps (values), the total number of steps, the average number of steps, the maximum number of steps, and the day the maximum number of steps occurred, and prints an itemized list of the days of the week and the number of steps walked that day using the `enumerate()` function, the total number of steps, the average number of steps, the maximum number of steps walked in a day, and the day which this occurred as shown in the examples below. Recall that commas in numbers can be obtained using a comma in the replacement field:

  `{:,}`

  This will work nicely for your output sentences as shown in the examples below, but it's a little bit trickier to obtain the commas with the formatting for each day. Try

  `{:>n,}`

  where I'll let you figure out the value of `n`. You'll also need to provide the number of spaces occurring before the colons in the output. Hint: Which day has the most letters?
- **`main()`**: This void function has no parameters. It creates a list of the days of the week (Sunday through Saturday) and uses this list as the argument to the non-void function `get_steps()` which returns a list of the steps walked in the week. It finds the total number of steps by summing this list and the average number of steps walked using this total and then creates a dictionary by zipping together the lists of the days of the week (keys) and the number of steps walked each day (values). The non-void function `max_steps_day()` is called with the dictionary as its argument, and it returns the maximum number of steps and the day which this occurred. Finally, the void function `print_results()` is called with the dictionary, total number of steps, average number of steps, maximum number of steps, and day which maximum number of steps occurred as the arguments.

```
Enter the number of steps you walked on Sunday: 14426
Enter the number of steps you walked on Monday: 2011
Enter the number of steps you walked on Tuesday: 14544
Enter the number of steps you walked on Wednesday: 1437
Enter the number of steps you walked on Thursday: 13937
```

```
Enter the number of steps you walked on Friday: 2286
Enter the number of steps you walked on Saturday: 14276

Your weekly steps:
1. Sunday   :  14,426
2. Monday   :   2,011
3. Tuesday  :  14,544
4. Wednesday:   1,437
5. Thursday :  13,937
6. Friday   :   2,286
7. Saturday :  14,276

You walked a total of 62,917 steps.
The average number of steps you walked was 8,988.
The maximum number of steps you walked (14,544) occurred on Tuesday.

Enter the number of steps you walked on Sunday: 16126
Enter the number of steps you walked on Monday: 2647
Enter the number of steps you walked on Tuesday: 13940
Enter the number of steps you walked on Wednesday: 947
Enter the number of steps you walked on Thursday: 13944
Enter the number of steps you walked on Friday: 1519
Enter the number of steps you walked on Saturday: 18966

Your weekly steps:
1. Sunday   :  16,126
2. Monday   :   2,647
3. Tuesday  :  13,940
4. Wednesday:     947
5. Thursday :  13,944
6. Friday   :   1,519
7. Saturday :  18,966

You walked a total of 68,089 steps.
The average number of steps you walked was 9,727.
The maximum number of steps you walked (18,966) occurred on Saturday.
```

When your program is working properly, demonstrate it to your TA to get credit.

There are only three tasks in this week's lab, because the last task was pretty challenging. You should feel really good about yourself and your coding, especially if you finished everything! You may even want to use the program you wrote in the last task, weather permitting!