PALs CSC-152

Section 02, Professor Kim, M-W-F

Tony Stone

November 12, 2025

Lesson IV

1 The Loop Structure

1.1 Motivation

Before we dive into the types of loop structures, we should first discuss why we may want to use a loop structure in the first place. Familiarize yourself with the term: **Increment**. Incrementing a given integer i is one of the most fundamental and crucial programming practices there is.

What incrementing mainly allows us to do is keep a count of something. This could be the length of a string, or maybe the number of attempts a user has at a game. Perhaps we would like to index a list in a specific element-wise fashion; incrementing an indexing variable is what we're looking for. This variable counts the positions of the list we've traversed, doesn't it?

We'll be focused on **while** and **for** loops. Perhaps in a bit of a backwards order, but we will: a. seamlessly transition from boolean logic into loop structures, and b. relate back to this section on traversing a list.

1.2 while

The while loop that makes a great use of boolean logic. With the **while** keyword, and then a piece of boolean logic, we can write a loop that starts as long as the condition we define is true, and end the loop when the condition is False. That may be a little abstract, so let's see an example:

```
i = 0
while i < 10:
    i += 1
print(i)
```

Output:

```
10
```

This code will execute since $i = 0 < 10$, and when i finally increments such that $i = 10$ the statement $i < 10$ returns False, therefore breaking the while loop.

WARNING: We have all done it, but please try to avoid:

```
i = 0
while i < 10:
```

This is known as an **infinite loop**. Why? Allow me to pose you a question; will i ever be greater than 10? That's right, after your computer crashes.

Another Example:

```python
    run = True
    i = 0
    while run:
        i += 1
        if i > 9:
            run = False
```

Output:

```
10
```

This is a more explicit means of the above code. Our 'boolean logic' is just the bool 'run' with the value "True" We increment i in the loop checking it's value. When i reaches 10, we change the value of 'run' from "True" to "False" which breaks the loop.

Lets explore a more complex example, a guessing game! We can use this while (*True statement*) to create a paradigm that allows a user any number of guesses. If you are not currently comfortable with function definition, and "import random" looks like wizardry, do not worry about it! One, we will cover functions very soon, so think of this as a soft introduction. Imports will continue to look like wizardry. The Python libraries are so rich and extensive that it *does* feel like magic. So I repeat, don't worry about it! It's just more for your toolbox!

```python
import random
def guessingGame(allowedAttempts):
    numberToGuess = random.randint(1, 10)
    guesses = 1
    guessed = False
    while not guessed:
        guess = input(
            f"attempt {guesses}: What number am I thinking of? ")
        if int(guess) == numberToGuess:
            print("You guessed it!")
            guessed = True
        elif guesses == allowedAttempts:
            print("Better luck next time!")
            break # Exit the loop
        else:
            guesses += 1
```

We can exit a while loop using the **break** keyword. Sometimes we want to 'break' but only for the current iteration, this can be done with the **continue** keyword. We will see more examples of that later on!

Of course, a statement which is *not* false, is true. We can use this to our advantage. We can use this as the true-valued statement in our while loop definition. Most often we do this for **readability**. defining a loop with 'while not (*false statement*)' makes more intuitive sense than 'while (*true statement*' which is meant to represent a false one!

1.3 for

: One of the many things that is great about Python, is how it lends itself to natural language. As a programmer, you may think "I would like to do x for everything in this list." Well

```python
blist = []
for item in alist:
    blist.append(item # With your desired operation x)
```

We see here that we have this variable item, which in each pass of this loop will take on the value of the 0th, 1st,..., nth element of alist. We'll do whatever operation x is, and then append it to our list b.

But do we need a whole new list? No!

```python
for i in range(len(alist)):
    alist[i] = alist[i] # With your desired operation x
```

range is a keyword that returns a list of integers. It is called with range(a, b) and returns a range of $[a, b)$. If range(a, b) is just passed a number, it assumes a is zero. If we pass the length of our list along with it, we'll have a range of integers that covers the indices of our list, i.e., [0, len(alist)-1]. We use this index in order to index the element within the for loop,z to which we can do our operation, and then stick it right back in alist!

Some more concrete examples:

```python
alist = [1, 2, 3]
blist = []
for item in alist:
    blist.append(item ** 2)
print(blist)
```

Output:

```
[1, 4, 9]
```

Or with indexing,

```python
alist = [1, 2, 3]
for i in range(len(alist)):
    alist[i] = alist[i] ** 2
print(alist)
```

Output:

```
[1, 4, 9]
```

It is not inherently clear that there is a better or worse method. This will be entirely problem-dependent! Your job will be to apply the appropriate method for the given task.

We had mentioned before about something known as **list comprehension**. Then, it was relevant to the extent that it is interesting, and relevant to our use with lists. But now it is no mystery! We know what "for" is doing. That's right, list comprehension is an empty list with a for loop stuck inside of it. Lets build a list in two ways, to really see it in action.

```
alist = []
for i in range(1, 4): #Numbers 1-3
    alist.append(i**2)
print(alist)
```

Output:

```
[1, 4, 9]
```

but with list comprehension:

```
alist = [i**2 for i in range(1, 4)]
print(alist)
```

Output:

```
[1, 4, 9]
```

This is such a handy tool to have when you are seeking to create a list! Keeping a mental repository of such Pythonic statements is strongly recommended.

2 Key Terms

- **incrementing**: The act of repetitious addition of an integer.
- **while**: The keyword that begins a loop, and runs for as long as the associated statement is True.
- **break**: The keyword to exit a loop.
- **continue**: The keyword to end the current iteration, and begin at the next one.
- **Infinite loop**: A loop that is set to run for a condition which is unalterable from inside the loop.
- **Readability**: The ability to which human can parse code, and derive meaning.
- **for**: Loop structure that iterates based on a variables inclusion in a collection.
- **list Comprehension**(for loop perspective): A for loop that is nested within list brackets '[]' to generate a list.

3 Availability

- In Lecture:
  - Wednesday: 10:00am - 11:00am
- PAL Sessions:
  - Monday: 2:00pm - 3:00pm
  - Wednesday: 9:00am - 10:00am
  - Wednesday: 2:00pm - 3:00pm

References

[1] Tony Gaddis. *Starting Out with Python.* Pearson, 5th edition, 2021.

[2] GeeksforGeeks. Python Tutorial — Learn Python Programming Language - GeeksforGeeks — geeksforgeeks.org. https://www.geeksforgeeks.org/python/python-programming-language-tutorial/. [Accessed 29-10-2025].

[3] Bradley N. Miller and David L. Ranum. Problem solving with algorithms and data structures using python. https://runestone.academy/ns/books/published/pythonds/index.html, 2014. Interactive online edition — accessed: 2024.

[4] Python Software Foundation. *Python: A dynamic, open source programming language*, 2025. Version 3.x documentation.

[5] W3Schools. W3Schools.com — w3schools.com. https://www.w3schools.com/python/default.asp. [Accessed 29-10-2025].