

Using Notional Machines to reinforce student comprehension of Lists in Introductory Programming

John William Lynch

lynchjohn98@gmail.com

Abstract

Lists are a powerful tool that can be the first obstacle for new programmers. New programmers can have trouble comprehending the full extent of what lists do and the various ways to create and use them. As a TA for introductory programming using python, students struggled using indices to call objects in a List properly, correctly counting index values, and understanding index out of bounds errors. This article will explore how my notional machine was integrated into my lessons, the observed benefits, and potential limitations. My notional machine gives a physical and visual example of lists that can potentially alleviate new programmers struggles with comprehending the List class in Python.

1 Project Description

1.1 Theoretical Framework

As technology advances programming is becoming more widely studied yet remains difficult to learn. To help students learn programming, research has been done into exploring the development of “mental models,” a user’s personal representation of the system they are working on and what occurs during run-time (Cañas, Bajo, & Gonzalvo, 1994). The mental model users develop, while not indicative of their ability, is a manifestation of their learning environment. Canas et al. (1994) found students learning with Trace facilities had mental representations based on the semantic aspects of the coding language, whereas Non-Trace facility students had their mental model based on the syntactic aspects of the coding language (Cañas et al., 1994). The idea that mental models of students are malleable and based on environments they learn in could potentially generate diverse teaching methods, one such method being Notional machines. A Notional machine is an idealized abstraction of a program that is programming language dependent and seeks to give a correct perspective on how the program runs (Sorva, 2013). Studies show that novice programmers sometimes have a hard time comprehending what occurs during their program execution (Du Boulay, O’Shea, & Monk, 1999), and Notional machines can be a useful mechanism to reduce the complexity and ease students into subject matter easily. The simplification of programming via Notional machines could be viewed as harmful as we are not telling the full truth of what is occurring inside the machine, yet it is counterproductive to force novices to the same standards veteran programmers have

(Du Boulay et al., 1999). Furthermore, the advancement of technology has allowed for a myriad of devices to program on and more diverse problems programmers must face (Sorva, 2013), so taking care to not overwhelm novice programmers is crucial. There are various ways to help novice programmers avoid becoming overwhelmed, ranging from programmer tools built for beginners or even methods that use the programmer’s own bodies to reinforce lessons (Sorva, 2013). Building on these concepts from Canas et al., Sorva, and Boulay et al., I created a Notional machine that draws from where students are sitting in their class in an effort to enhance their mental model of how lists work in Python.

2 Notional Machine Description

2.1 Explanation of Problems with Lists

When introducing Lists to students they have to learn a few rules right away; there are differences between an object’s value and an object’s index in a list, we start counting indices at 0, we can dictate how fast we traverse through the list (ie: one at a time, two at a time), different data types can be contained within the same list, and you can iterate through the list in various ways. In an upcoming study, Dr. Craig Miller and Dr. Amber Settle explore list traversal using *by value* and *by index* methods, which could potentially present more struggles as students may have a hard time differentiating between the two (Miller & Settle, 2021).

```
MyList = ["apple", "orange", "banana", "kiwi"]
for i in myList:
    print(i)
```

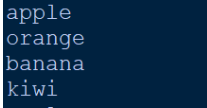


Fig. 1. Simple iteration method using *looping by value* with output.

The method shown in Figure 1 is what is sometimes referred to as *looping by value*, as the *i* in the for loop is the actual string object of apple, orange, banana, kiwi (Miller & Settle, 2021). Besides the immediate issues that can arise here with syntax creations (ie, misplacing the colon after the list, forgetting a bracket, forgetting a parentheses) there are usually not

too many problems. After this introduction we can then introduce indices. From here a secondary form of iteration through a list is introduced.

```
MyList = ["apple", "orange", "banana", "kiwi"]
for i in range(len(MyList)):
    print(i)
```

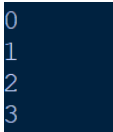


Fig. 2. Simple iteration method using *looping by index* with output.

The method shown in Figure 2 is referred to as *looping by index*, as the i in this scenario is not the actual object, rather the index of the object (Miller & Settle, 2021). The `range()` function takes in a specific number and begins at 0 and increments by 1 until it reaches 1 less than the specified number. This is combined with the `len()` function, which returns the given length of a specified object. Combining these means that `len(MyList)` returns a 4 because `MyList` has four objects, and `range()` iterates from 0 up until 1 less than 4, ending at 3. This looks inherently more complicated yet is a tool that can be helpful for more of the complex coding problems. If you wanted to print out the items like the looping by value method, you would need to call the list in the print statement at that specific index, shown in Figure 3.

```
for i in range(len(MyList)):
    print(MyList[i])
```

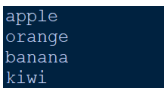


Fig. 3. Simple iteration method using *looping by index* with object output.

2.2 Novice Programmer Exercises using Lists

A sample problem using the *looping by index* method shown in Figure 2 and Figure 3 would be attempting to find if a string occurs sequentially in a list without using any built-in methods (ie: `count()`, `set()`). The reason for not using built in methods is to show that one can iterate through more than one item in a list during a for loop iteration by modifying the value of their index. Take the list in Figure 4 for example.

```
MyList = ["apple", "pear", "pear", "kiwi"]
```

Fig. 4. List of strings with sequential string *pear* occurring.

We see that *pear* occurs one after the other, and our statement should return `True` if asked if two words occur sequentially. How does one access the next object in the list to do this comparison? There are multiple ways to approach this problem, but for arguments sake we will use the *looping by index*

approach. One can access other objects in a list by manipulating their current index location via arithmetic. For instance, adding 1 to your current index location will move you one to the right in the list, with the inverse occurring if subtracting 1. An example of this is shown in Figure 5

```
>>> MyList[0]
'apple'
>>> MyList[1+1]
'pear'
>>> MyList[3-1]
'pear'
>>> MyList[2+1]
'kiwi'
```

Fig. 5. Using arithmetic to edit index value to access objects.

We need to make our program state that if our current index is a certain value that is equal to the next index's value, we have a match. Rewriting the code to add in an if statement will fix this, yet an unexpected error occurs as Figure 6 displays.

```
for i in range(len(MyList)):
    if MyList[i] == MyList[i+1]:
        print("The same word occurs back to back")
```

the same word occurs back to back
Traceback (most recent call last):
File "C:/Users/John Lynch/Desktop/list.py", line 4, in <module>
 if MyList[i] == MyList[i+1]:
IndexError: list index out of range

Fig. 6. Code that accesses two indices but throws an `IndexError`.

When running the program, we get our successful print message but an unexpected side result. This `IndexError` was a consistent theme throughout my lab assistant experience, and students struggled with finding a solution. The solution is that we reduce how far our `range(len)` function goes so that it does not go out of bounds. This is done by modifying the code from `range(len(MyList))` into `range(len(MyList)-1)`. As a veteran programmer, my mental model of lists was cemented and just stating that they needed to reduce the range by one because they are adding one would not be sufficient.

2.3 Notional Machine Usage

During lab sessions students would be sitting in five rows of six students. I would then single out a row and begin with the student closest to my desk. This student would be the start of the list and would be given the index of 0. Following this the student would be tasked with saying their index number and then would point to the student next to them and say, "plus 1". This pointing is indicative of the idea we can add to our index value to increase where we currently are in a list. This next student would then say their index number and point down to the other student. This would continue until the end of the row where the last student would say their index number and I would ask them to point to the student next in line and say "plus 1". Since they are the last student they would point to the wall/edge of the class, and this would reflect how Python

goes out of bounds. Making Python use a for loop to access an index that doesn't exist is like the last student pointing at the non-existent next student. At this point the student should say their index number and how many students (objects) are in that row (the list). The inverse would then occur where the last student points back to the previous student and says "minus 1" until it returned towards the beginning of the list again. This idea illustrates the process of using $[i]$, $[i+1]$, and $[i-1]$ within their code. Stating the number of users in the row during each iteration can help them understand that they would need to subtract from the length of the list if they were adding to their indices so they can avoid index out of bounds errors. Illustrations of this process is given in Figure 7.

3 Potential Conceptual Advantages of Usage

No surveys or research was done with this Notional machine, it was merely a strategy employed to help comprehend the index out of bounds error. A potential advantage of this exercise is that it could make students think about how list traversal occurs when using the loop by index method. Another potential advantage is displaying how indices increase, and the differences between the number of objects in a list against the index locations of those objects. This Notional machine could be expanded upon by having students say a color of their shirt or shoes they are wearing to attribute the idea that they are an object instead of the index. This would tie into the differences between looping by value and looping by index. Aiding their mental model to differentiate between a List's object's value and object's index would be beneficial. A small gain could be the interactivity between students it poses. This can present an opportunity to be a activity for icebreaking and getting students away from purely coding during lab time.

4 Conceptual Disadvantages of Usage

Mental models are malleable and indicative of a students learning environment (Cañas et al., 1994), notional machines are language dependent and do not always apply to every situation (Sorva, 2013). Because of this the notional machine could place barriers or detract from a student's mental model in future coding practices. Learning Arrays in Java pose large differences from Lists in Python. Arrays are type specific and cannot combine different types of objects (ie: a string and int in the same array), so having the students use a notional machine that implies the container can have mixed objects could confuse them in future lessons. This notional machine is reliant on the looping by index method and does not contain an abstract idea for how one might traverse a list using the looping by value method. This method only contains a $+1$, -1 idea for looping by index, and does not display the full power that index manipulation can have on lists such as multiplication and using negative index values to acquire certain objects (ie: `MyList[-1]` will always return the last item in the list). Another drawback is more broad, but just as important. Michael Berry and Michael Kolling address the issue of a lack of a shared abstract model that students can base their mental models on, and that ad-hoc

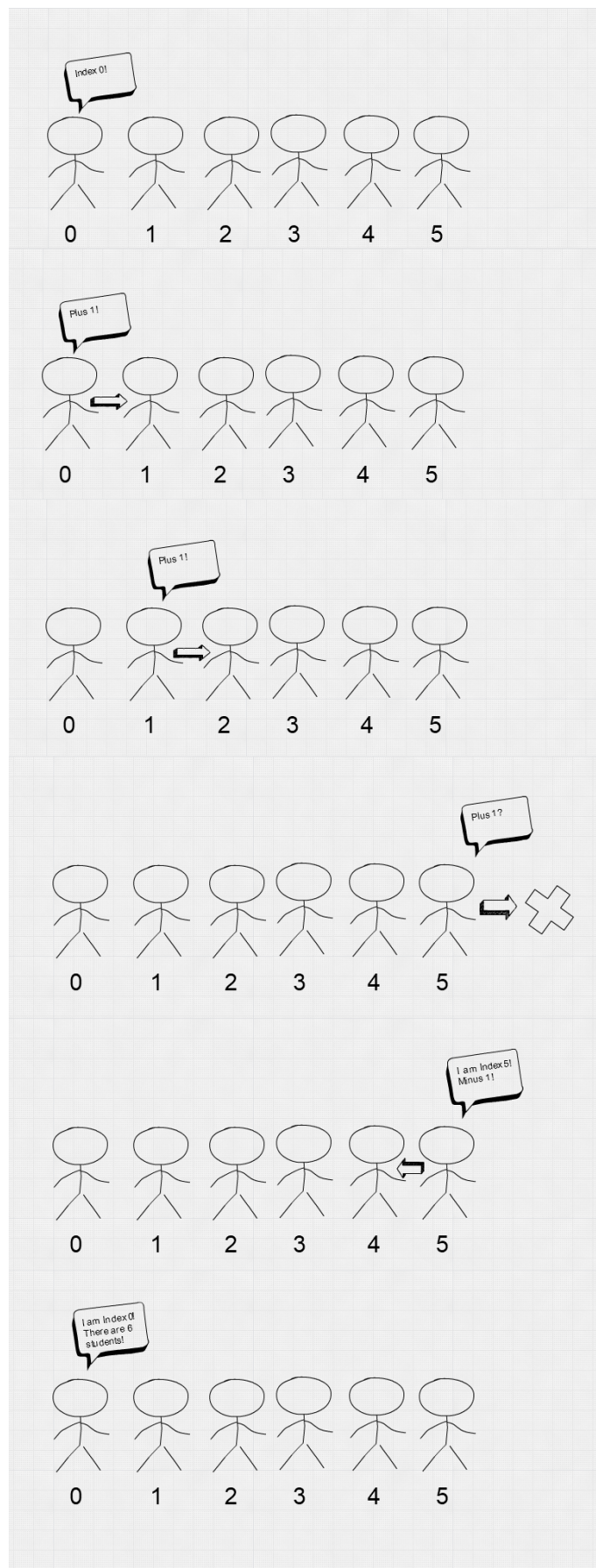


Fig. 7. Example illustration of the Notional machine process.

models are usually formed by instructors but can range in effectiveness (Berry & Kölling, 2014). My notional machine was something I created based on my own mental model of Lists. Having taken the same Python course two years prior, I understood students confusion and frustration at times and wanted a more in-depth movement oriented solution as opposed to a worded explanation. While the intentions can be good, creating a model like this could cause more confusion and not be effective for each student.

5 Conclusion

In their paper surveying over 500 students and teachers, Essi Lahthinen, Kirsti Ala-Mutka, and Hannu-Matti Jarvinen found that “the biggest problem of novice programmers does not seem to be the understanding of basic concepts but rather learning to apply them.” (Lahtinen, Ala-Mutka, & Järvinen, 2005). The students’ experiences with index out of bounds errors reflect this, as while students can grasp the movement of going up and down their row with adding and subtracting, they still had trouble creating for loops or knowing when to use a certain for loop iteration method. Practice is always a great benefit for programmers but should not be the sole answer when more research and advancements in the realm of mental models and notional machines can be taken. An aggregation of various notional machines can be what helps push the trend towards a more systematic and widely-accepted model and is what inspired me to create this paper. I believe any effort put into aiding students with interactive models and new machines will be greatly beneficial but should be taken with caution. In our pursuit to help students create proper mental models for their immediate problems we should be wary not to create ones that cripple their long-term understanding. Despite this, any effort to push the medium forward should be taken, and I would like see other notional machines and how they are used brought forth and discussed.

References

- Berry, M., & Kölling, M. (2014, 06). The state of play: A notional machine for learning programming. *ITICSE 2014 - Proceedings of the 2014 Innovation and Technology in Computer Science Education Conference*. doi: 10.1145/2591708.2591721
- Cañas, J., Bajo, M., & Gonzalvo, P. (1994, 05). Mental models and computer programming. *International Journal of Human-Computer Studies*, 40, 795-811. doi: 10.1006/ijhc.1994.1038
- Du Boulay, B., O’Shea, T., & Monk, J. (1999). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Human-Computer Studies*, 51(2), 265 - 277. doi: <https://doi.org/10.1006/ijhc.1981.0309>
- Lahtinen, E., Ala-Mutka, K., & Järvinen, H.-M. (2005, 09). A study of the difficulties of novice programmers. In (Vol. 37, p. 14-18). doi: 10.1145/1067445.1067453
- Miller, C. S., & Settle, A. (2021). Mixing and matching loop strategies: By value or by index? New York, NY, USA: Association for Computing Machinery. Retrieved from <https://doi.org/10.1145/3408877.3432368> doi: 10.1145/3408877.3432368
- Sorva, J. (2013, 06). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, 13, 8:1-8:31. doi: 10.1145/2483710.2483713