COE 302
Object Oriented Programming with Python V

Akinola Oluwole

soakinola@abuad.edu.ng

Content

- Flow constructs
- Arrays

Control Flow

- A program’s control flow is the order in which the program’s code executes. The control flow of a Python program is regulated by conditional statements, loops, and function calls.

Conditional

- In a sequence of statements, the Python interpreter normally executes the statements one after another in the order they appear. We say that the **flow of control** passes from one statement to the next in order.
- Sometimes the **flow of control** needs to be different. For example, we often want certain statements to be executed only under certain conditions. A construct that causes this to happen is called a conditional.
- The basic conditional construct in Python is a compound statement that starts with the keyword if. We call such a statement an if-statement for short.

Conditional

The most basic kind of if-statement has this form:

```
if condition :
   Statements
```

Here's an example that

```
Score = 90
If score >= 70:
    print(“Student score is A”)
```

Conditional

The most basic kind of if-statement has this form:

```
#using the keyword 'in' within an 'if statement'
word = "Baseball"
if "b" in word:
  print( "{ } contains the character b".format(word) )
```

Conditional

- The comparison operator that means “does not equal”, like “≠” in mathematics, is written != in Python. The operators < and > mean “is less than” and “is greater than”, as you might expect. For “is less than or equal to”, like “≤” in mathematics, Python uses <=, and similarly >= means “≥”.
- Python has operators whose operands are Booleans, too: and, or, and not. The not operator is unary, and produces False if the value of its operand is True and True if the value of its operand is False, as you might expect.
- The or operator also means what it appears to mean, but Python evaluates it in a particular way. In evaluating an expression of the form A or B, the Python interpreter first evaluates A. If its value is True, the value of the whole expression is True; otherwise, the interpreter evaluates B and uses its value as

Conditional

The condition in an if-statement header is a comparison, as in the example above. The result of a comparison is a value of the Boolean type. This type has only two values, True and False.

These values behave like the integers 1 and 0 in most contexts — in fact, Python considers them numeric values and you can do arithmetic with them.

The main use of Boolean values in Python is to control execution in if-statements

Conditional

Another comparison operator is **in**, which (for example) can be used to test whether a character is in a string.

Here is an example, where c is a variable containing a character:

Array

- It is a collection of elements that can be accessed by means of indexes. Arrays can be linear (i.e., single dimension) or multidimensional (e.g., matrixes).
- Basic arrays are fixed in size, which is the size declared during initialization. They support only a single type (e.g., all integers).
- In Python, array declaration can be performed as

```
from array import *
a = array('i', [1,2,3])
```

Array

- The first parameter ‘i’ specifies the typecode, which in the specific case refers to signed integers of 2 bytes. The second parameter provides initializers for the given array.
- Some languages support dynamic sizing as well as different data types. It is the case, for example, of Python where the following code will be fine:
- `array = [1, "string", 0.2]`
- `type(array[0])`
- `type(array[1])`
- `type(array[2])`

Array Pros and cons

- They are fairly easy to use and they provide direct access by means of *indexes,* and the entire list of elements can be accessed linearly in *O(n)*, where *n* is the number of elements into the list.
- However, inserting and deleting items from the list are more computationally expensive due to shifting operation required to perform these tasks.
- As a worst-case scenario, to better understand why it happens, consider deleting the first element of the list which is at index = 0. This operation creates an empty spot; thus all the items from index = 1 need to be shifted one position backward. This gives us a complexity of *O(n)*. Analogous considerations are for insertion.

List

- List are also known as “arrays” and have similar characteristics.
- A list is a data structure in Python that is a mutable, ordered sequence of elements.
- Mutable means that you can change the items inside, while ordered sequence is in reference to index location.
- The first element in a list will always be located at index 0. Each element or value that is inside of a list is called an item. Just as strings are defined as characters between quotes, lists are defined by having different data types between square brackets [ ]. Also, like strings, each item within a list is assigned an index, or location, for where that item is saved in memory. Lists are also known as a data collection.

List

```
# declaring a list of numbers
nums = [5, 10, 15.2, 20]
print(nums)
```

The difference here is that the value is a set of items declared between square brackets. This is useful for storing similar information, as you can easily pass around one variable name that stores several elements. To separate each item within a list, we simply use commas.

List

- `# accessing elements within a list`
- `print( nums[1] ) # will output the value at index 1 = 10`
- `num = nums[2] # saves index value 2 into num`
- `print(num) # prints value assigned to num`

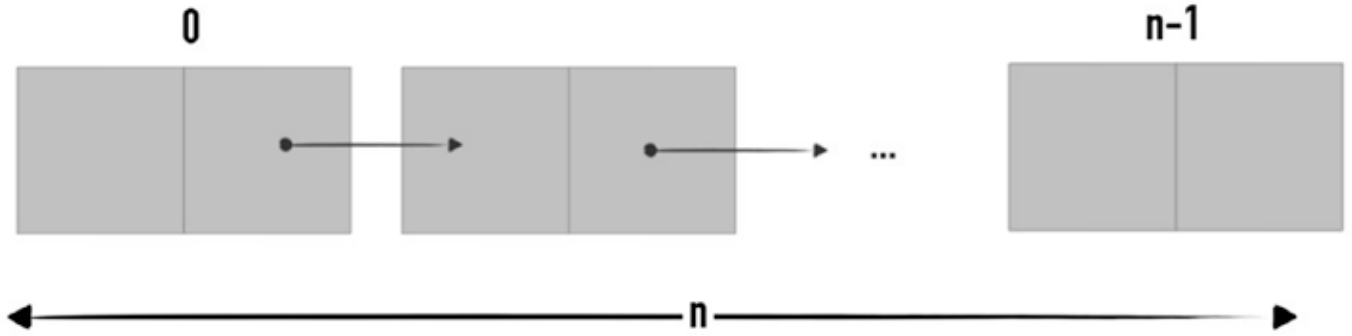
In order to access a specific element within a list, you use an index. When we declare our list variable, each item is given an index. Remember that indexing in Python starts at zero and is used with brackets.

Array Examples

- `a = [None]*5 #a list with five elements`
- `data = [None] * 5`
- `for i in range(0,5):`
  `data[i] = [0] * 5`

Linked List

- A linked list is a collection of nodes, where each node is composed of a value and a pointer to the next node into the list.
- Compared to arrays, add and remove operations are easier because no shifting is required. Removing an item from the list can be performed by changing the pointer of the element prior to the one that needs to be removed.

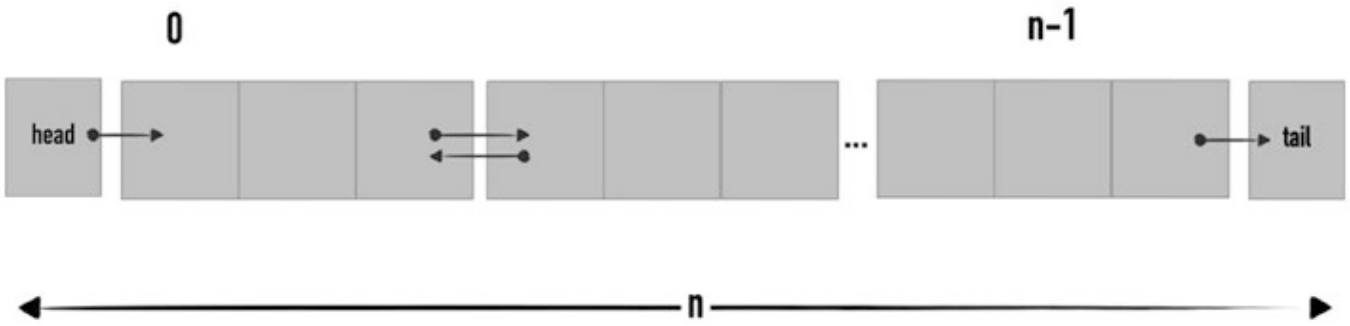


Linked List

- Time complexity is still O(n) in the worst case. Indeed, in order to find the item that needs to be removed, it is required to navigate the entire list up to the searched element.
- The worst case happens when the removal needs to be done on the last element into the list. Often linked lists serve well as underlying implementation of other data structures including stacks and queues.

Doubly Linked List

- A doubly linked list is similar to regular linked lists, but each node stores a pointer to the previous node in the list



- Adding is slightly more complex than the previous type due to more pointers
- to be updated. Add and remove operations still take O(n) due to sequential
- access needed to locate the right node.

Doubly Linked List

- Stack
- Queue
- Hash Map
- Binary Search Trees

Doubly Linked List

- In a figurative way, think about this doubly linked list as the navigation bar into your Google search. You have the current page (the node) and two links (pointers) to the previous and following page.
- Another example is the command history. Suppose you are editing a document. Each action you perform on the text (type text, erase, add image) can be considered a node which is added to the list.
- If you click the undo button, the list would move to the previous version of changes. Clicking redo would push you forward one action into the list.

Reference

- Giuliana Carullo (2020) Implementing Effective Code Reviews How to Build and Maintain Clean Code.