Name: ____________________

Week 7: Review and BYOA

Parameters and Arguments

We’ve discussed function parameters before, but it’s worth revisiting a little more formally. To refresh your memory, we use parameters and arguments when we want to pass values into our functions. Let’s look at an example of where this might come in handy. Imagine we are working for YouTube, and we’re working on displaying the details of videos in a playlist, perhaps an episode in a particular season of a TV show. We want to show the name of the video, as well as showing the name of the playlist the video is in. And we want to do this in a function! We could, of course, just put our new code in the middle of the rest of the code, but when there are thousands of lines of code that gets very messy. Instead, we want to write a function where we can put our title printing code, without having to touch the rest of our program.

Now, in the main code, where our function will be called from, other programmers have done the hard work of getting the name of the video for us, and they’ve also made a dictionary where the keys are video names, and the values are the playlists that those videos are in. So, for example, the dictionary could look something like this:

```
{ "Rick Astley - Never Gonna Give You Up (Video)": "Troll Videos",
  "What are Frogs, Mulder?": "important videos (complete)",
  "Everything Wrong With Battleship In 6 Minutes Or Less": "Movie Sins Videos - CinemaSins",
  "Li'l Sebastian": "Parks and Rec Season 3" }
```

Or, formatting it another way:

Key:	Value:
"Rick Astley - Never Gonna Give You Up (Video)"	"Troll Videos"
"What are Frogs, Mulder?"	"important videos (complete)"
"Everything Wrong With Battleship In 6 Minutes Or Less"	"Movie Sins Videos - CinemaSins"
"Li'l Sebastian"	"Parks and Rec Season 3"

This dictionary will help us get the playlist name so that we can include it in our title, but first we have to get access to it! Remember, functions can't see variables that are defined outside of the function, and the video name and the name-to-playlist dictionary have been created in the main code, outside of our function. How do we pass them in to our function, so we can use them there?

Well, you may remember that we use function parameters, but I want to be a little more specific this time. We have to use two different things in order to pass these values in: Function **parameters** and function **arguments**. Let's look at what those are.

A function **argument** is the easier one to explain. That's when we put a value between the parentheses when we're calling a function, which tells Python that we want to send that value to the function. For instance, when we write:

```
int("50")
add(20, 5)
print("Hello, world!")
```

the "50", the 20 and 5, and the "Hello, world!" are all function arguments, since they're in between the parentheses of our calls to the functions int(), add(), and print(), respectively. Putting them in between those parentheses just tells Python that the function should get that value and use it! After all, if we couldn't pass "50" to int(), then int() wouldn't know what value we wanted it to convert, and if we couldn't pass "Hello, world!" to print(), then print() wouldn't know what to print. So this is how we take a value from outside of a function, and tell Python to send it to the function, so that the function can use it.

Function **parameters** are the other side of this coin. If we want to pass a value in with an argument, we need to define our function so there's somewhere for it to go! We do this by defining function parameters, which we just put in the parentheses after the function name on our def line. So we might write this: (The bodies have been omitted for now, so we're only looking at the definition, or the line with the def.)

```
def int(to_convert):
def add(int_a, int_b):
def print(string_to_print):
```

These parameters just hold whatever values are passed in as function arguments when the function is called. So, using our examples above, when int() is called, to_convert will be "50". When add() is called, int_a will be 20, and int_b will be 5, because the first argument goes into the first parameter, the second argument into the second parameter, and so on and so forth. Every argument needs a parameter to go into, and every parameter needs to be filled by an argument, or else we'll get an error! So if we called

```
int("50", "25")
```

or

```
int()
```

we would get an error! In the first example, we have too many arguments, and not enough parameters, and in the second example we have too few arguments, and too many parameters. The number of parameters and the number of arguments always have to match exactly! Let's look at how this might look for our title function:

```
# Our code
print_title(title, title_to_playlist_dict):
    # We'll fill this out later!

# Main code that calls our function
video_title = get_video_name() # Written by someone else
playlist_dict = create_video_to_playlist_dictionary() # Written by someone else
print_title(video_title, playlist_dict) # Our function
```

Let's look at this in a bit of detail. First, notice that we haven't filled out our print_title function yet - that's okay! We're just looking at how we call the function right now, so we'll deal with the function body later. So let's see what's going on here.

First, there's the variables video_title and playlist_dict, which contain the title of our video, and the dictionary of video names to playlist titles. The variable names, as well as their contents, are arbitrary. We could have renamed video_title to title_of_video, vid_name, or even fizz_glarbnorp. Python does not care what our variable names are! Python will simply take the value stored in the variable, and pass it in into the function.

We're also using functions to get the values of video_title and playlist_dict. This is one of the nice things about using functions - we didn't have to write that code! The get_video_name() function probably does fancy things to get the name from the user's web request, and the create_video_to_playlist_dictionary() might do something like getting the dictionary from a database somewhere. But since someone else wrote that code, we don't have to! We can focus on just our function, without worrying about that other code.

So, where does our function get called? Well, on that last line, where is says `print_title(video_title, playlist_dict)`, that's calling our function. And it's calling it with two arguments: `video_title` and `playlist_dict`. These values of these two arguments will be passed in to our function and put into the function parameters, `title` and `title_to_playlist_dict`. The names are different, because we only pass in the values stored in our variables, not the variables themselves! So whatever the title of the video stored in `video_title` is, it will be stored into `title` in our function. And similarly, the value in `playlist_dict` will be stored in `title_to_playlist_dict` when our function is running.

Since the values will be stored in our function parameters, we can use those parameters when we write our function! Just like we use variables, and can treat them as if they are the values stored in them, we can use parameters, and treat them as if they are the values stored in them.

In the main code, where we have the value of our video stored in the variable `video_title`, we can use that variable anywhere we want the video title. Similarly, once that value is passed in to our function via the `title` function parameter, we can use that parameter anywhere we want the video title! Let's look at what the final code might look like:

```python
# Our code
print_title(title, title_to_playlist_dict):
    playlist_name = title_to_playlist_dict[title]
    full_title = title + ": " + playlist_name
    print(full_title)

# Main code that calls our function
video_title = get_video_name() # Written by someone else
playlist_dict = create_video_to_playlist_dictionary() # Written by someone else
print_title(video_title, playlist_dict) # Our function
```

The only new thing here is the function body - we have our video-name-to-playlist dictionary stored in `title_to_playlist_dict`, and our video title stored in `title`, so we just use them! We get the value from the dictionary associated with our video's title, and store that into a variable. Then we use string concatenation to get our data into the "Title: Playlist" format, and then print it out! Notice how we were able to use our parameters just like variables, and the values stored in the parameters when our function runs are the values of the arguments when we called the function. So the value in the parameter `title` is the same as the value in the argument `video_title`, and the value in the parameter `title_to_playlist_dict` is the same as the value in the argument `playlist_dict`.

Return Values

Now, in our last example, we just printed out the title that we came up with. This probably isn't going to work! Most likely there will be some other function that displays something in the app, rather than just printing it. Let's assume someone has written a function for us, called display_in_app(), which takes one parameter, which is the string to display. So we might call display_in_app("Hello, world!"), and then the app would display "Hello, world!". We could call this from inside of our function, but let's assume we want to call this from the main code instead. This might seem silly, but in the real world you do things like this frequently, to make it more clear what the code is doing when someone comes back to read it later.

So, if we want to call the display_in_app() function from our main code, we'll need a way to get the complete title out of our function! We've already learned we do this via the return statement, but, again, I want to be a little more precise about what's going on. When Python runs a file, it will start with the first line, and go through line by line, executing each line one after another, unless it finds something that tells it to change the order the lines are executed in, such as an if, while, function call, etc. I want to focus on specifically what happens when Python's execution order is interrupted by function calls. First though, I want to give a simpler example without functions, to show how Python handles multiple things happening on one line. Let's imagine we have the following code:

```
result = (5 + 4) * 3 - 2
```

This is pretty familiar, we're just storing the result of some math operations into a variable. We could calculate what the result will be, but instead, let's look at how Python calculates it. We know Python does PEMDAS, but it's also important to realize that Python is basically doing each step one at a time, and replacing each calculation with its result. So while the above is what we have in our code, what Python will execute would be something like this:

```
result = (5 + 4) * 3 - 2
result = (9) * 3 - 2
result = 9 * 3 - 2
result = 27 - 2
result = 25
```

Python can only do one operation at a time, so in order to calculate the result of this multi-step problem, it has to do each step on its own. And once it's done a step, it takes the result and replaces the calculation with the result.

The same thing happens with functions! Whenever Python sees a function, it's just like it if saw a math operation - it knows that it needs to go and do some calculations. And just like with the math above, Python will replace the function call with its result when it has finished calculating. So if we do a simple print statement with some concatenation, to Python the execution might be like this:

```
print("Hello, " + "world!")
print("Hello, world!")

```

Notice that empty line on the bottom! Python sees a function, so it knows it needs to calculate it, but it also sees that there's a concatenation operation as a function argument. It needs to know the value of the arguments before it can execute the function, so first it calculates "Hello, " + "world!" and replaces it with "Hello, world!". Then Python can evaluate the print() function - and here's the tricky part. When it finishes calculating print(), print() is replaced with the value it returns, just like how "Hello, " + "world!" was replaced with the value it returned. But print() doesn't return any value! So to Python, it ends up looking like a blank line. Let's take another look at this with some array functions:

```
my_list = [1, 2, 3]
new_list = my_list.append(4)
```

What's going to happen here? Well, it's reasonable to think that new_list will have [1, 2, 3, 4] in it, but what Python actually executes on that line is this:

```
new_list = my_list.append(4)
new_list =
```

.append() also doesn’t return anything! But Python always replaces functions with the value they return. So .append() is replaced with nothing, because it returns nothing. This means that new_list is going to be empty! The same thing applies to functions we create. If we write

```
def add(int_a, int_b):
    print(int_a + int_b)

answer = add(5, 3)
```

Python will print the result of adding 5 and 3 when it runs our function, but answer will end up empty, just like new_list! This is because we didn’t return a value. If we want to assign the result of our function into a variable, we have to return the value we want to assign, like this:

```
def add(int_a, int_b):
    print(int_a + int_b)
    return int_a + int_b

answer = add(5, 3)
```

Now when Python executes the line answer = add(5, 3), it will run the lines in add, printing out 8, and also returning 8. So when it replaces add(5, 3) with the result returned, it will be replaced with 8, and Python will see this:

```
answer = 8
```

Now we’ve successfully stored the result of our addition into a variable outside of our function!

Note this also means that if we write

```
add(5, 3)
```

Python will execute it like this:

```
add(5, 3)
8
```

And what does Python do when it sees a value on a line by itself? Well, since we didn’t tell it to do anything with that 8, it will just ignore it and move on! If you want Python to do something with a value you return, you’ll need to store it into a variable, pass it into another function, or otherwise make use of it.

Going back to our YouTube example, we can modify our function to make use of this! If we want to use the title we create in the main function, we just have to return it, like this:

```
print_title(title, title_to_playlist_dict):
    playlist_name = title_to_playlist_dict[title]
    full_title = title + ": " + playlist_name
    return full_title
```

And the main code will have to be changed to use the returned value somewhere, or it will just forget it immediately! So our end code would look like this:

```python
# Our code
print_title(title, title_to_playlist_dict):
    playlist_name = title_to_playlist_dict[title]
    full_title = title + ": " + playlist_name
    return full_title

# Main code that calls our function
video_title = get_video_name() # Written by someone else
playlist_dict = create_video_to_playlist_dictionary() # Written by someone else
full_title = print_title(video_title, playlist_dict) # Our function
display_in_app(full_title)
```

Building Our Own Array

In Python, we're lucky because our arrays are classes, and so they have a lot of nice methods, such as .append(), .pop(), .remove(), .count(), and so on. But in many languages, we will not always have access to methods like this by default. In C++, for instance, you can only do two things with arrays: Get an element at an index, and put a value at an index. Using just this then, can we build our own array class, and set up all of the methods we're used to having access to? The answer is yes, of course, or I wouldn't be asking the question. However, I want to show you that these methods are not magic, and that we know enough to be able to build them. Let's walk through how we would build an Array class in Python, using only array indexing, and recreate the other array methods.

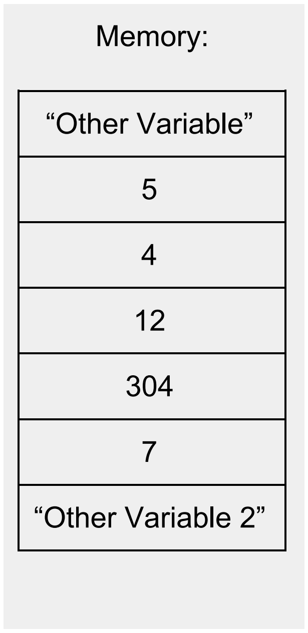
First off, we'll need a class. Since we're making an array, let's just call our class Array.

```python
class Array():
```

Next, we'll need somewhere to store the values we want to put in our array. Obviously we're going to have to use Python's built-in arrays for this, but this brings up another catch: Python is also cheating when it creates arrays. You might remember how we draw arrays in memory, as a bunch of spots one after another, like this:

5	4	12	304	7

As it turns out, that's how the computer actually stores the array as well, simply by taking adjacent spaces in memory and reserving them for the array. So, for instance, if we stored this array into memory along with some other variables, it might end up looking like the table below.



But this gives us a new problem: How does the computer know how many spaces in memory to reserve for the array? After all, can't we just call .append() whenever we want and make our array bigger? This is somewhere Python cheats again. If we were to call .append() on the array on the left, there's no more space in memory for another value to be put there. So Python would have to find a new place in memory that had room to hold 6 values in a row, and move everything into that new location before adding our new item to the array.

We're going to assume something a little simpler, and closer to what C++ does for arrays. Instead of having to worry about moving things around in memory, we're simply going to take in a parameter when our array is created, that tells us the maximum number of elements that can be stored in it. So if a user said Array(7), we would make an array with room for 7 elements.

How are we going to do this? Well, Python has a somewhat odd feature of arrays, which is that you can use multiplication to repeat arrays, just like you can with strings. So if we write [0] * 3, we'll get out [0, 0, 0], and if we write ["hey"] * 5, we'll get ["hey", "hey", "hey", "hey", "hey"]. What we can do then, is use Python's special `None` keyword, which stands for an empty variable, and use the multiplication syntax to make an array with a number of `None`s equal to the maximum number of values for our array. So, for example, if the maximum length was 3, we would say [None] * 3, and get out [None, None, None], which is an empty array of size 3.

In our class, we will need to know how large to make the array when we're creating our class, this will have to go in our __init__ method. We'll take in self, as always, and then another parameter telling us how big to make our array, then use that variable to create an empty array of the desired size, then store it into a class variable so that we can use it later, like so:

```python
class Array():
    def __init__(self, int_max_size):
        self.array = [None] * int_max_size
```

This will allow us to pass an argument to Array() when we're creating it, telling it how large it should be. So if we were to say `Array(10)`, self.array would be set to an array of size 10, where every entry in that array was `None`, or empty.

Append

Now that we can create our array, let's take a look at the first of the functions we'll want to make: .append(). This will store a new element into our array, putting it after all of the elements that have already been stored. And remember, the only tool we're letting ourselves use on self.array is our square bracket index notation - all of the other methods are the things we're trying to build! For .append(), we know it's going to need to have a parameter, so that the user can pass the value to append as an argument. So we know what the function **signature** (the part on the def line consisting of the name and the number of parameters) will look like:

```
    def append(self, value):
```

What will the body be though? If we want to put the value into the first empty slot, then we can just use a loop to check each element in the array one at a time until we get to one that is empty, or `None`:

```
    def append(self, value):
        int_i = 0
        ith_value = self.array[int_i]
        while ith_value != None:
            int_i += 1
            ith_value = self.array[int_i]
        self.array[int_i] = value
```

We create a counter, and start it off at zero, and use that to get the first element of our array. Then, we do our loop, incrementing our counter and checking if we've reached an empty position yet. Once we have, we exit the loop, and store our value in that empty position. One of the things we should always worry about with while loops like this is what happens in what we call the **edge cases**. These are the scenarios that are special, or tricky, such as when we're appending the first thing to an empty list, or when we're trying to append to a full list. We're not going to worry about the case when the list is full right now, but we'll come back to it later. You should take a moment to make sure you understand why this code works for the first insertion though.

Pop

The next method we want to create is .pop(). There are two versions of .pop(), actually, one that takes no arguments and removes the last thing in the list, and one that takes an index as an argument, and removes the thing at that index. But, if you remember, we made a point that the number of arguments we pass when we call a function always has to exactly match the number of parameters we put in the function signature (remember, that's just the bit on the def line with the name and parameters). How can a function only sometimes take an argument then? The answer is, as usual, more Python magic. For now, let's just write separate pop_last() and a pop_index() methods.

Before we get started on that though, let's notice something: pop_last() is just a special case of pop_index()! So if we can just find the index of the last element, we can call pop_index() from pop_last(), and reduce the amount of code we have to write. Let's do pop_index() first then, and then write pop_last() using it.

So how do we pop an item at an index? Well, we know we want it removed from the list, so that, e.g., [3, 5, 7].pop(1) would become [3, 7]. But what does this look like in memory? Our initial array will look something like the “Before pop” table below. Our initial instinct might be to just write over the 5 with an empty value, and put a `None` in there, but that doesn't quite work. If you look at the “After simple deletion” table, we see that if we just replace the 5 with a `None`, we're going to have a problem where the values in our array are no longer continuous, which isn't what we want. To get around this, we're going to have to first delete the element we're popping, and then move all of the elements after it back by one, so that we don't have any gaps.

Before pop:

3	5	7	None

After simple deletion:

3	None	7	None

After deletion with shift:

3	7	None	None

Since we have the index at which the deletion is to occur, we can just set the element at that index to None, then shift all the elements following that back with a loop, until we reach a `None` value, which will tell us we're at the end of the elements in the array.

Initial array:

3	5	7	11

Delete 5:

3	None	7	11

Move 7 back:

3	7	7	11

Move 11 back:

3	7	11	11

Delete duplicate last element:

3	7	11	None

There is one catch though: Computers can't actually move numbers. When we store a number into a spot in our array in order to move it back, the previous number will still exist. This is because what we're really doing is making a copy of the value we're moving, and then overriding the value of the space we're moving into. Fortunately, the extra copy of most of the numbers will be itself overridden by the next number we move. The exception to this is the last number, which, unless we watch out, we'll end up with two copies of, since there's nothing left to copy to override its value in the old position. The good news is that the fix is really easy: We just delete that one leftover number when we're done shifting everything back!

The table on the left above shows what this process will look like, and illustrates two more things to note. First, since we're going to be moving the 7 into the space occupied by the 5, we didn't really need to bother to replace the 5 by a `None`. We could have simply moved the 7 into that space, and it would have overridden the 5 just fine. Second, what do we do if the array is full? There won't be a `None` value at the end to tell us when we've reached the last element. The easiest way to solve this is to simply store in a class variable the size of the array, so that we know how many total slots there are, and then stop when we see a None or when we reach that number.

Notice we haven't written any code at all yet! We've just been thinking about what we're trying to do, what we want to happen in memory, and how things should look before and after our code runs. After all, we can't write code if we don't know what we want it to do! But now that we've spent time figuring that out, let's take a stab at turning this into code. We know we're going to have to change our __init__ function to store the total size of our array, for one. We also know that we're going to move values back until we either see a None or reach the end of our array, which sounds like a good place to use a while loop, and also like we've got an idea of what the condition will be. How will we move values back? Well, we'll just copy them into the previous slot in the array with a simple variable assignment. And once we're done, we have to remember to delete that leftover copy of the last element! This is enough for us to take a stab at coding this up:

```python
class Array():
    def __init__(self, int_max_size):
        self.array = [None] * int_max_size
        self.size = int_max_size

    # .append() has been omitted for simplicity's sake.

    def pop_index(self, index):
        current_index = index
        next_index = index + 1
        while next_index < self.size and self.array[next_index] != None:
            self.array[current_index] = self.array[next_index]
            current_index += 1
            next_index += 1
        self.array[current_index] = None
```

Let's look at what we did, and how it solved our problem. First, we store the index we want to pop the value at into current_index, as well as the next index. This is because if the next index is ever equal to the size of our array, or if the value at that index is ever None, we know that we should stop, so we need to keep track of that next value for our while loop condition. If the next_index is still less than our total size, and self.array[next_index] isn't None, then we can go ahead and safely store the value at next_index into the slot at current_index, overriding whatever is currently there, just like the example above shows. We then increment things by 1, check our condition again, and repeat.

Once the condition is false, we know that we've reached the end, either because we're at the end of the whole array, or because the next element is None. We're basically done at this point, but we have to remember to get rid of that one extra copy of our last element. And since the last element is whatever was in next_index last time, and we've since incremented our counters, that extra copy will be in current_index now, and we can just set the item at current_index to `None`. You should make sure you understand why this code works!

Now, we should always ask ourselves about edge cases. We've worried about the case where the array is full, but what about when it's empty, or if we have an array of size 0? What will happen then? In both of these cases our code will throw an error, but this is actually probably what we want it to do - after all, it doesn't make sense to remove something from an empty list. But what about the case where the size is 1? We should definitely be able to remove the last item from a list. In that case, the next value will be `None` immediately, which means we will skip the while loop, and our cleanup line will run, deleting the value! So our code will work if there is only 1 element. You should always ask yourself these sort of questions when you're writing code! It's much easier to fix bugs if you find them by thinking about edge cases when writing your code than trying to figure out what's going wrong when you find them due to the errors they cause later on.

We also want to make sure that we're never going to be trying to get an invalid element out of our array, such as an element less than 0, or greater than our largest index. We could have a problem with this if the user passed in a negative number, but assuming they don't do that, we'll be okay. This is because we use next_index to make sure current_index never leaves the range of valid indexes. It is possible for next_index to get too large, and become equal to self.size, which is one more than our max index, but if it does, then we will skip the parts of the code involving self.array[next_index], which would throw an IndexError and tell us our index was out of range.

This is a little tricky in the while loop itself - notice that if next_index is equal to self.size, then it looks like we're going to throw an error when we evaluate the second half of the `and` condition, the `self.array[next_index] != None` bit. However, Python will not evaluate the second half of an `and` condition if the first part is false, and `next_index < self.size` will be false. This is known as "short circuiting" the condition, and is common practice in most languages.

Now let's look at pop_last(), our no-argument version of pop(). This one is a lot easier now that we have pop_index()! All we do is loop over the array until we either find that the next element is None, or we are at the end of the array, then use that index to call pop_index(). Our code then, would look like this:

```
    def pop_last(self):
        current_index = 0
        while current_index < self.size and self.array[current_index] != None:
            current_index += 1
        self.pop_index(current_index - 1)
```

The only tricky bit here is that we have to subtract 1 from our current index before passing it - this is because it will only stop incrementing once it's either found a `None`, or reached the size of the array, and in both cases it will have gone one index too far. If you're not sure why this is, try drawing out two arrays, one with some `None`s at the end, and one where the values fill the entire array, drawing out memory for this function, and walking through it, line by line, to see what it will do.

Remove

Next, let's do remove(). The remove() function will look through the array and remove the first element whose **value** matches the argument passed to the function. Notice that this is different from .pop(), which will remove the element whose **index** matches the argument passed in. Let's think about how we can approach this one.

We're obviously going to need to loop over our values again, from index 0 to the last index, and when we find a value stored that matches the value passed in, we'll need to remove it from our array. But wait - didn't we just write code to remove elements from our array? Can we make use of that? Yes we can! As long as we know the index of the element we want to remove, we can just call .pop(index), and we've already written the code to handle removing the element and shifting everything down. This is why functions are so useful, and why it's useful to be able to call class functions from inside other class functions. This boils down to a pretty simple loop:

```
    def remove(self, value):
        bool_found = False
        int_i = 0
        while not bool_found:
            if self.array[int_i] == value:
                self.pop_index(int_i)
                bool_found = True
            int_i += 1
```

The only real noteworthy thing here is our use of a "flag value" bool_found to know how long to loop for. There are other ways we could have done this, but this is a fairly common pattern.

We should still think about our edge cases: What happens if there are more than one instance of the value in the array, or if there aren't any? Well, if there are multiple, our flag value will make sure we exit the loop after the first one, which is what we want. However, there is a problem if the value isn't in the array at all! See if you can figure out how to fix the problem!

Insert

Sometimes we want to put an element into our array at a position other than the end. This is when we use insert()! We just pass it two arguments, the position to insert at, and the item to insert. But wait - if we're inserting in the middle, we can't just put the element in that position, or else we'll end up overwriting whatever is already there! To avoid overwriting anything, we'll have to move everything else in our array forward by 1, so that we open up a space, basically doing the opposite of what we did in .pop() when we moved everything backwards by 1 to purposefully remove an element. You can see what this will look like on the left here.

Initial array:

5	3	89	4	None

Inserting at index 1:

5	3	89	4	4

5	3	89	89	4

5	3	3	89	4

5	-100	3	89	4

Just like with remove, we start copying elements, but this time we start with the last element, and copy it into an empty space. Then we go backwards, copying each element one space forward, handily taking care of the fact that we'd made duplicates. When we get to the index we want, we will now have two copies of it, so we can just overwrite the one in the index we want with our new value, without losing any of the elements of our array.

A lot of this code is going to look very similar to the code we wrote for .reverse()! Note that it's important we go backwards here instead of forwards though - if we went forwards, starting at the index we wanted to free up and copying the elements into the next slot, we would have copied our 3 on top of our 89, overwriting the 89. Then, when we tried to copy the 89 into the 4's slot, we would have copied the 3 instead, and overwritten the 4 as well. So we have to start at the back!

Here's what the actual code might look like:

```python
def insert(self, int_index, value):
    int_i = 0
    while self.array[int_i] != None:
        int_i += 1
    while int_i > int_index:
        self.array[int_i] = self.array[int_i - 1]
        int_i -= 1
    self.array[int_i] = value
```

First we do a while loop to find the first empty slot, then, starting there, we go backwards and copy the value in the prior to int_i into the slot at int_i. When we've reached int_index, we know we've already copied whatever was in that slot, so we can now safely overwrite it with our value.

The edge case where we don't have an empty slot in our array isn't taken care of here, but when we come back and look at how we would fix the similar problem in .append(), we'll take care of this one too. We also don't have a way to make sure that int_index isn't less than 0, or greater than the largest element in our array. Unfortunately, while we could certainly add if statements to check for these things, the proper way of handling them and telling the code calling this function that there was an error requires a topic we haven't quite gotten to yet, so we'll have to put this off for now.

Count

You may remember this function from the Poker lab - count will tell you the number of times a value is present in the array. How can we implement this one? Well, we need to check all of our values, so we're going to need a loop, and we're going to need some sort of counter to keep track of how many match the value passed in. Let's see how that looks:

```
    def count(self, value):
        counter = 0
        int_i = 0
        while int_i < self.size and self.array[int_i] != None:
            if self.array[int_i] == value:
                counter += 1
            int_i += 1
        return counter
```

You may notice that this is the first function we've written that returns a value! All of our other functions have simply **modified** the array, but without returning anything. Other than that, this function is very straightforward, and doesn't really introduce anything we haven't seen in the previous functions.

Just like with .pop_index(), we have to be careful not to access anything beyond the end of our array, and we make use of Python "short-circuiting" the `and` condition to make sure that when int_i reaches self.size, we don't try to get the element at that (non-existent) index from our array.

Reverse

Now we're getting into some of the functions we haven't talked as much about yet, though you may have seen them anyway. The reverse() function will take a list, and do exactly what is says - reverse it. So the first element will become the last, the last will become the first, and so on. This one sounds complicated, but I'm hoping you'll see that if we think about the tools we have available, we can often find ways to make complicated things very simple.

At first glance, it seems like we're going to have to do something like swapping the first and the last element, then swapping the second and the second-to-last element, and so on and so forth, with one counter going forwards, and another going backwards. And this would work, but it would also be rather complicated. If we didn't stop the counters in the right place, they would pass each other and keep going, swapping values that had already been swapped, and un-reversing the list. We'd have to figure out where the middle is, and stop them there. And that would require us to figure out if there were an odd or even number of elements, since the "middle" of an even number of things is actually two of them. Don't get me wrong, we could totally do this! But it's finicky enough that we should think for a moment and see if we can find another way.

What if, rather than swapping each pair of elements, we just started from the back, and appended each item into a new array? Then that new array would have all of our elements in reverse order, and we could just replace our array with the new array. So we would do something like this:

Initial Array:

"cat"	"dog"	"fish"	None	None

New Array:

None	None	None	None	None

Initial Array:
i = 2

"cat"	"dog"	**"fish"**	None	None

New Array:

"fish"	None	None	None	None

Initial Array:
i = 1

"cat"	**"dog"**	"fish"	None	None

New Array:

"fish"	"dog"	None	None	None

Initial Array:
i = 0

"cat"	"dog"	"fish"	None	None

New Array:

"fish"	"dog"	"cat"	None	None

First, we make a new empty array, the same size as our current array. Then we figure out where the last element in our list is, and point at it with a counter. Once we've done that, we append the element at that position into the new array, decrease the counter by 1, and repeat. Once we're done, the new array will have our reversed list, and we can just use an assignment to override our initial array with the new array! Let's see it in code:

```
    def reverse(self):
        int_i = 0
        new_array = Array(self.size)
        while int_i < self.size and self.array[int_i] != None:
            int_i += 1
        int_i -= 1
        while int_i >= 0:
            new_array.append(self.array[int_i])
            int_i -= 1
        self.array = new_array.array
```

Some of this is stuff we've seen before, such as the loop, and checking for both the size and the `None` condition. However, a few things are worth pointing out. First, we can make a new instance of our class inside a function that's being called on an existing instance of our class. Second, since arrays are variables, we can just set our array equal to the array object inside of our new, reversed, Array class. Lastly, since we incremented our int_i counter until it reached self.size or found a `None`, we had to decrement it by 1 after that loop, in order to back it up to the last actual element in the array.

Note we do not return the reversed array! We just modified the existing array in our class.

Sort

This is the last method we're going to look at. Just like .reverse() will reverse an array, .sort() will sort an array into ascending order, meaning that the small elements come first, and the large elements come later. Let's take a look at how we could write this one.

Sorting is actually a very complicated topic in computer science, since there are a lot of ways to sort a list, and there are different pros and cons to each of them. Some of the sorting algorithms out there can get very complicated, but we're only going to worry about finding one that works, not about finding one that is the fastest or the most memory efficient. To do this, we're going to look at a sorting algorithm called **insertion sort**. Insertion sort is very simple, and it uses a concept kinda like what we did with .reverse() earlier. We're going to take advantage of two facts in order to get our sorted array: First, an empty array is sorted. It might seem silly, but technically all 0 of the elements there are in the right order. Second, if we insert an item into a sorted list either at the current position of the first item larger than it, or at the end if it is the largest, then the list will remain sorted.

Initial Array:

5	3	4	7	None

Sorted Array:

None	None	None	None	None

5	None	None	None	None

3	5	None	None	None

3	4	5	None	None

3	4	5	7	None

Using these two facts, we can build up a sorted array in a fairly simple manner. First, we make an empty array the same size as our initial array. We'll call this array "Sorted", which, by our first fact, is true when we create it.

Next, we take each element in our initial array one at a time, and insert it into the sorted array either at the position of the first element larger than it, or at the end, if it is the biggest. By our second fact, the array will remain sorted when we do this.

Once we've inserted every element in our initial array into the sorted array, the sorted array will still be sorted, and will now contain everything we wanted put in order! So we can simply replace our array with the sorted array, and then our array will be in sorted order. The only tricky part here is making sure we insert each element into the right spot in the sorted array - fortunately, since it is sorted, we can just start at the beginning and loop over it until we either find an element larger than the one we are inserting, or we reach the end of the array, in which case we can simply .append() our element. The code then will look something like this:

```
def sort(self):
    sorted = Array(self.size)
    int_i = 0
    value = self.array[int_i]
    while int_i < self.size and value != None:
        int_j = 0
        sorted_value = sorted.array[int_j]
        while int_j < sorted.size and sorted_value < value and sorted_value != None:
            int_j += 1
            sorted_value = sorted.array[int_j]
        if sorted_value == None:
            sorted.append(value)
        else:
            sorted.insert(int_j, value)
        int_i += 1
        value = self.array[int_i]
    self.array = sorted.array
```

This is a fair bit of code, and there's a lot going on, but it's just doing what we discussed earlier! We make our new array, then we get the first element in our initial array. We start looping, and for every element in our array, we start a second, nested loop, where we loop over the elements in our sorted array until we find one that is larger than our value, or we reach the end of the element present. If we're at the end, then the value we have at the end of the inner loop will be None, so we check for that, and if it is None, we append our element, since getting None back means it was bigger than everything currently in the sorted array, and otherwise we insert it at the position that our inner loop calculates. Once we've done that for every element in our array, we just overwrite our array with the sorted array, and we're done!

Our edge conditions here should be pretty familiar. Does this work when there are no elements in the array? Does it work when the array is full? Did we make sure to double check that we never try to get an element at a non-existent index? (There is in fact an error in this code! See if you can find it. It's been fixed in the full code on GitHub, if you want to see if you're correct. Try to understand why the change was necessary, and not just what was changed!)

Code

The full code is available at www.github.com/sethborder/CS100ArrayExample.

We'll see in a later lesson how we can handle the errors that result from things like negative array indices and so on.