ZED SHAW'S HARD WAY SERIES

Learn MORE PYTHON 3 the HARD WAY

The Next Step for
New Python Programmers

ZED A. SHAW

LEARN MORE PYTHON 3
THE HARD WAY

```python
    colors.push("Zinc White")
    colors.push("Nickle Yellow")
    colors.push("Perinone")
    assert colors.remove("Cobalt") == 0
    colors.dump("before perinone")
    assert colors.remove("Perinone") == 2
    colors.dump("after perinone")
    assert colors.remove("Nickle Yellow") == 1
    assert colors.remove("Zinc White") == 0

def test_first():
    colors = SingleLinkedList()
    colors.push("Cadmium Red Light")
    assert colors.first() == "Cadmium Red Light"
    colors.push("Hansa Yellow")
    assert colors.first() == "Cadmium Red Light"
    colors.shift("Pthalo Green")
    assert colors.first() == "Pthalo Green"

def test_last():
    colors = SingleLinkedList()
    colors.push("Cadmium Red Light")
    assert colors.last() == "Cadmium Red Light"
    colors.push("Hansa Yellow")
    assert colors.last() == "Hansa Yellow"
    colors.shift("Pthalo Green")
    assert colors.last() == "Hansa Yellow"

def test_get():
    colors = SingleLinkedList()
    colors.push("Vermillion")
    assert colors.get(0) == "Vermillion"
    colors.push("Sap Green")
    assert colors.get(0) == "Vermillion"
    assert colors.get(1) == "Sap Green"
    colors.push("Cadmium Yellow Light")
    assert colors.get(0) == "Vermillion"
    assert colors.get(1) == "Sap Green"
    assert colors.get(2) == "Cadmium Yellow Light"
    assert colors.pop() == "Cadmium Yellow Light"
    assert colors.get(0) == "Vermillion"
    assert colors.get(1) == "Sap Green"
    assert colors.get(2) == None
    colors.pop()
    assert colors.get(0) == "Vermillion"
    colors.pop()
    assert colors.get(0) == None
```

Study this test carefully so that you have a good idea of how each operation *should* work before trying to implement it. I wouldn't write all of this code into a file at once. Instead it's better to do just one test at a time and make it work in small chunks.

WARNING! At this point if you are unfamiliar with automated testing then you will want to watch the video to watch me do it.

Introductory Auditing

As you make each test run you will conduct an audit of your code to find defects. Eventually you will track the number of defects you find with your audits, but for now you're going to just practice auditing code after you write it. An "audit" is similar to what a tax agent does when the government thinks you are cheating on your taxes. They go through each transaction, each amount of money coming in, all money going out, and why you spent it the way you did. A code audit is similar as you go through each function, and analyze all parameters coming in, and all values going out.

To conduct a basic audit you will do this:

1. Start at the top of your test case. Let's do `test_push` in this example.
2. Look at the first code line and determine what is being called and what is being created. In this case it's `colors = SingleLinkedList()`. That means we are creating a `colors` variable, and calling the `SingleLinkedList.__init__` function.
3. Jump to the top of this `__init__` function, keeping your test case and the target function (`__init__`) side by side. Confirm that you have done so. You then confirm that you called it with the *correct number and type of function arguments*. In this case `__init__` only takes `self`, and it should be the right type.
4. You then go into `__init__` and go line by line, confirming each function call and variable in the same way. Does it have the right number of arguments? The right types?
5. At each branch (`if-statement`, `for-loop`, `while-loop`) you confirm that the logic is correct and that it handles any possible conditions in the logic. Do `if-statements` have `else` clauses with errors? Do `while-loops` end? Then dive into each branch and track the functions the same way, diving in, checking variables, and coming back, checking returns.
6. When you get to the end of a function or any `return`, you jump back to the `test_push` caller to check that what *is* returned matches what you expect when you called it. Remember, though, that you do this for each call in `__ini__` as well.
7. You are done when you reach the end of the `test_push` function and you've done this recursive checking into and out of each function it calls.

This process seems tedious at first, and yes it is, but you'll get quicker at it the more often you do it. In the video you'll see me do this *before* I run each test (or at least I try really hard to do it). I follow the following process:

1. Write some test code.
2. Write the code to make the test work.
3. Audit both.
4. Run the test to see if I was right.

Exercise Challenge

We now reach the part where you're ready to try this. First, read through the test and study what it does, and study the code in `sllist.py` to figure out what you need to do. I suggest that when you attempt to implement a function in `SingleLinkedList` you first write the comments describing what it does, then fill in the Python code to make those comments work. You'll see me do this in the video.

When you have spent one or two 45-minute sessions hacking on this and trying to make it work, it's time to watch the video. You'll want to attempt it first so that you have a better idea of what I'm trying to do, which makes the video easier to understand. The video will be me just coding and not talking, but I'll do a voice-over to discuss what's going on. The video will also be faster to save time, and I'll edit out any boring mistakes or wasted time.

Once you see how I do it, and you've taken notes (right?), then go and attempt your more serious tries and perform the code auditing process as carefully as you can.

Auditing

Once you have written your code, make sure you do the auditing process I describe in the introduction to Part III. I will also be doing it in the video for this exercise if you aren't quite sure how it's done.

Study Drill

Your study drill for this exercise is to try to implement this algorithm *again*, completely from memory the way I describe in the introduction to Part III. You should also try to think through what operations in this data structure are most likely painfully slow. When you are done, perform your auditing process on what you created.

This page intentionally left blank

Double Linked Lists

The previous exercise may have taken you quite a while to complete since you have to figure out how to make a single linked list work. Hopefully, the video gave you enough information to complete the exercise and also showed you exactly how to do an audit of your code. In this exercise you're going to implement the better version of linked lists called `DoubleLinkedList`.

In the `SingleLinkedList` you should have realized that any operation involving the end of the list has to travel through each node until it gets to the end. A `SingleLinkedList` is only efficient from the front of the list where you can easily change the `next` pointer. The `shift` and `unshift` operations are fast, but `pop` and `push` cost you as the list gets bigger. You *might* be able to speed this up by keeping a reference to the next-to-last element, but then what if you want to replace that element? Again, you'll have to go through all of the elements to find this one. You can get some speed improvements with a few minor changes like this, but a better solution is to simply change the structure to make it easier to work from any position.

The `DoubleLinkedList` is nearly the same as the `SingleLinkedList` except it also has a `prev` (for previous) link to the `DoubleLinkedListNode` that comes before it. One extra pointer per node and suddenly many of the operations become much easier. You can also easily add a pointer to the end in the `DoubleLinkedList` so you have direct access to both the beginning and the end. This makes `push` and `pop` efficient again since you can access the end directly and use the `node.prev` pointer to get the previous node.

With these changes in mind our node class looks like this:

dllist.py

```python
class DoubleLinkedListNode(object):

    def __init__(self, value, nxt, prev):
        self.value = value
        self.next = nxt
        self.prev = prev

    def __repr__(self):
        nval = self.next and self.next.value or None
        pval = self.prev and self.prev.value or None
        return f"[{self.value}, {repr(nval)}, {repr(pval)}]"
```

All that's added is the `self.prev = prev` line and then handling that in the `__repr__` function. Implementing the `DoubleLinkedList` class uses the same operations as the `SingleLinkedList` class, except you add one more variable for the end of the list:

dllist.py

```python
class DoubleLinkedList(object):

    def __init__(self):
        self.begin = None
        self.end = None
```

Introducing Invariant Conditions

All of the operations to implement are the same, but *now* we have some additional considerations:

dllist.py

```python
    def push(self, obj):
        """Appends a new value on the end of the list."""

    def pop(self):
        """Removes the last item and returns it."""

    def shift(self, obj):
        """Actually just another name for push."""

    def unshift(self):
        """Removes the first item (from begin) and returns it."""

    def detach_node(self, node):
        """You'll need to use this operation sometimes, but mostly
        inside remove().  It should take a node, and detach it from the
        list, whether the node is at the front, end, or in the middle."""

    def remove(self, obj):
        """Finds a matching item and removes it from the list."""

    def first(self):
        """Returns a *reference* to the first item, does not remove."""

    def last(self):
        """Returns a reference to the last item, does not remove."""

    def count(self):
        """Counts the number of elements in the list."""

    def get(self, index):
        """Get the value at index."""

    def dump(self, mark):
        """Debugging function that dumps the contents of the list."""
```