Introduction to Python Classes and Linked Lists

Part 2 of "Data Structures and Algorithms in Python"

Data Structures and Algorithms in Python is a beginner-friendly introduction to common data structures (linked lists, stacks, queues, graphs) and algorithms (search, sorting, recursion, dynamic programming) in Python, designed to help you prepare for coding interviews and assessments.

How to Run the Code

The best way to learn the material is to execute the code and experiment with it yourself. This tutorial is an executable Jupyter notebook. You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on Google Colab or Kaggle to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up Python, download the notebook and install the required libraries. We recommend using the Conda distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

> **Jupyter Notebooks**: This notebook is made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things - you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top.

Problem

In this notebook, we'll focus our discussion on the following problem:

> **QUESTION**: Write a function to reverse a linked list

Before we answer this question, we need to answer:

- What do we mean by linked list?
- How do we create a linked list in Python?
- How do we store numbers in a linked list?
- How do we retrieve numbers in a linked list

```
!pip install jovian --upgrade --quiet
```

```python
import jovian
```

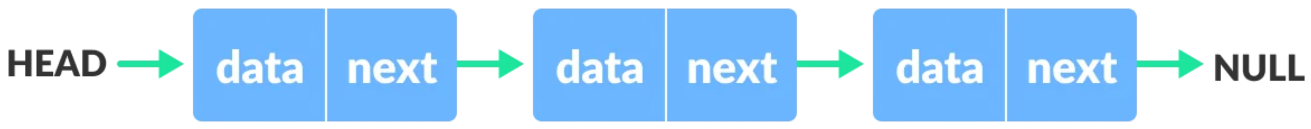
```python
jovian.commit()
```

```
[jovian] Attempting to save notebook..
[jovian] Updating notebook "aakashns/python-classes-and-linked-lists" on
https://jovian.ai/
[jovian] Uploading notebook..
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ai/aakashns/python-classes-and-linked-
lists
```

```
'https://jovian.ai/aakashns/python-classes-and-linked-lists'
```

Linked List

A linked list is a *data structure* used for storing a sequence of elements. It's data with some structure (the sequence).



We'll implement linked lists which support the following operations:

- Create a list with given elements
- Display the elements in a list
- Find the number of elements in a list
- Retrieve the element at a given position
- Add or remove element(s)
- (can you think of any more?)

A Quick Primer on Classes in Python

Let's create a class for it. A class is a blueprint for creating objects.

```python
class Node():
    pass
```

We can create an object with nothing in it.

```python
Node()
```

```
<__main__.Node at 0x7f904a0fdc88>
```

We just created an object of the class `Node`. However, we have to have a way to access the object. We can do so by creating a variable.

```
node1 = Node()
```

The *variable* `node1` holds a reference the object, and can be used to retrieve the object.

```
node1
```

```
<__main__.Node at 0x7f904a0fda58>
```

When we call the `Node()` again, it creates a new object.

```
node2 = Node()
```

```
node2
```

```
<__main__.Node at 0x7f904a0fda90>
```

You can tell that the objects are different because they are at different addresses in the RAM (more on that later).

We can have multiple variables pointing to the same object.

```
node3 = node1
```

```
node3
```

```
<__main__.Node at 0x7f904a0fda58>
```

Our object isn't doing much. Let's give it the ability to store a value. First, we'll store the constant value 0. We can do this using a *constructor*.

```
class Node():
    def __init__(self):
        self.data = 0
```

Two things to note:

- The double underscores
- The self (a replacement for `this`)
- `self.data` creates a property called. We can name a property anything we wish (`val`, `number`, `the_thing_inside` etc.)

```
node4 = Node()
```

So internally what's happening is that Python first creates an empty object, stores the reference to the empty object in an temporary variable called `self`, calls the `__init__` function with `self` as the argument, which then sets the property `data` on the created object with the value 0.

```
node4.data
```

```
0
```

And we can change the value inside the variable.

```
node4.data = 10
```

```
node4.data
```

```
10
```

Let's create nodes with the values 2, 3 and 5

```
node1 = Node()
node1.data = 2
```

```
node2 = Node()
node2.data = 3
```

```
node3 = Node()
node3.data = 5
```

```
node1.data, node2.data, node3.data
```

```
(2, 3, 5)
```

While this is OK, there's an easier way to do it.

```
class Node():
    def __init__(self, a_number):
        self.data = a_number
        self.next = None
```

```
node1 = Node(2)
node2 = Node(3)
node3 = Node(5)
```

```
node1.data, node2.data, node3.data
```

```
(2, 3, 5)
```

Now we are ready to define a class for our Linked list.

```
class LinkedList():
    def __init__(self):
        self.head = None
```

```
list1 = LinkedList()
```

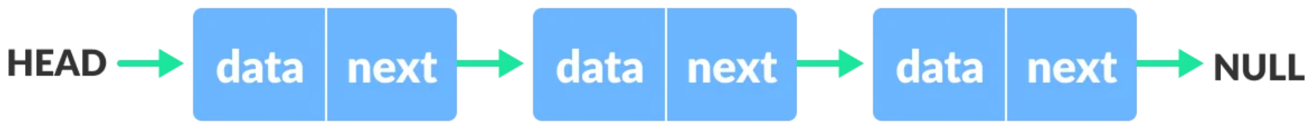
```
list1.head = Node(2)
```

```
list1.head.next = Node(3)
```

```
list1.head.next.next = Node(4)
```

```
list1.head.data, list1.head.next.data, list1.head.next.next.data
```

```
(2, 3, 4)
```



```
list1.head, list1.head.next, list1.head.next.next, list1.head.next.next.next
```

```
(<__main__.Node at 0x7f904a177390>,
 <__main__.Node at 0x7f904a177438>,
 <__main__.Node at 0x7f904a1774e0>,
 None)
```

While it's OK to set value like this, we can add a couple of arguments.

```
class LinkedList():
    def __init__(self):
        self.head = None

    def append(self, value):
        if self.head is None:
            self.head = Node(value)
        else:
            current_node = self.head
            while current_node.next is not None:
                current_node = current_node.next
            current_node.next = Node(value)
```

```
list2 = LinkedList()
list2.append(2)
```

```
list2.append(3)
list2.append(5)
```

```
list2.head.data, list2.head.next.data, list2.head.next.next.data
```

```
(2, 3, 5)
```

Next, let's add a method to print the value in a list.

```
class LinkedList():
    def __init__(self):
        self.head = None

    def append(self, value):
        if self.head is None:
            self.head = Node(value)
        else:
            current_node = self.head
            while current_node.next is not None:
                current_node = current_node.next
            current_node.next = Node(value)

    def show_elements(self):
        current = self.head
        while current is not None:
            print(current.data)
            current = current.next
```

```
list2 = LinkedList()
list2.append(2)
list2.append(3)
list2.append(5)
```

```
list2.show_elements()
```

```
2
3
5
```

Let's add a couple of more functions: `length` and `get_element` to get an element at a specific position.

```
class LinkedList():
    def __init__(self):
        self.head = None

    def append(self, value):
        if self.head is None:
            self.head = Node(value)
        else:
            current_node = self.head
            while current_node.next is not None:
                current_node = current_node.next
            current_node.next = Node(value)

    def show_elements(self):
        current = self.head
        while current is not None:
            print(current.data)
            current = current.next

    def length(self):
        result = 0
        current = self.head
        while current is not None:
            result += 1
            current = current.next
        return result

    def get_element(self, position):
        i = 0
        current = self.head
        while current is not None:
            if i == position:
                return current.data
            current = current.next
            i += 1
        return None
```

```
list2 = LinkedList()
list2.append(2)
list2.append(3)
list2.append(5)
list2.append(9)
```

```
list2.length()
```

```
4
```

```
list2.get_element(0)
```

```
2
```

```
list2.get_element(1)
```

```
3
```

```
list2.get_element(2)
```

```
5
```

```
list2.get_element(3)
```

```
9
```

Given a list of size `N`, the the number of statements executed for each of the steps:

- `append`: N steps
- `length`: N steps
- `get_element`: N steps
- `show_element`: N steps

Reversing a Linked List - Solution

Here's a simple program to reverse a linked list.

```
def reverse(l):
    if l.head is None:
        return

    current_node = l.head
    prev_node = None

    while current_node is not None:
        # Track the next node
        next_node = current_node.next

        # Modify the current node
        current_node.next = prev_node

        # Update prev and current
        prev_node = current_node
        current_node = next_node

    l.head = prev_node
```

```
list2 = LinkedList()
list2.append(2)
list2.append(3)
list2.append(5)
list2.append(9)
```

```
reverse(list2)
```

```
list2.show_elements()
```

```
9
5
```

```
3
2
```

That's how you reverse a linked list!

```python
import jovian
```

```python
jovian.commit()
```

```
[jovian] Attempting to save notebook..
```