TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Fall 2024 (@ Section 1 | Tuesdays and Thursdays 5PM - 7PM PT)
Personal Member ID#: **111572**

Unit 5 Cheatsheet

Overview

Here is a helpful cheatsheet outlining common syntax and concepts that will help you in your problem solving journey! Use this as reference as you solve the breakout problems for Unit 5. This is not an exhaustive list of all data structures, algorithmic techniques, and syntax you may encounter; it only covers the most critical concepts needed to ace Unit 5. In addition to the material below, you will be expected to know any required concepts from previous units. You may also find advanced concepts and bonus concepts included at the bottom of this page helpful for solving problem set questions, but you are not required to know them for the unit assessment.

Standard Concepts

Object Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that uses 'objects' and their interactions to build applications and computer programs. We can think of objects as virtual representations of real life object or data type. Python offers us many built-in objects or data types like integers (Python representation of a whole number), floats (Python representation of a decimal number), and strings (Python representation of a chunk of text), but the real magic of OOP is that it allows us to also create our own data types!

Classes

Classes are the mechanism through which we can design our own custom data types. A class is a blueprint for our new data type/object and defines the characteristics and functionality that every object of that class will have. Classes are comprised of two things:

- **properties** variables that describe *characteristics* of our object.
- **methods** functions that define *behaviors* of our object.

You can think of properties like adjectives that describe our object and methods like verbs that describe actions that our objects can take or that we can perform on our object.

Defining a Class

Let's look at an example of using a class to create a virtual representation of a dog. We can create a `Dog` class by representing a dog as a combination of its name, breed, and owner - these are the `Dog` class' properties.

```
class Dog:
    def __init__(self, name, breed, owner):
        self.name = name
        self.breed = breed
        self.owner = owner
```

Instantiating a Class

The `__init__()` function, also called the *constructor* describes how to build a new object or *instance* of a class. The job of the constructor is to assign values to object properties. Like a normal function, constructors can have parameters. The `Dog` constructor above takes in three arguments: the `name` of the dog we are creating, the `breed` of the dog, and the `owner` of the dog and assigns the argument values to the class properties of the same name.

The constructor (and all class methods) includes a special parameter `self`, which refers to the current instance of the class that is being created. We do not need to pass in a value for self when we call the constructor.

We call constructors differently than we call normal functions. Instead of calling the constructor by writing its function name `__init__()`, we call it by using the name of the class. For example, we can create a new `Dog` object/instance with the following:

```
my_dog = Dog('Fido', 'Cocker Spaniel', 'Ada Lovelace')
```

The above code creates a dog whose name is `'Fido'`, breed is `'Cocker Spaniel'` and owner is `'Ada Lovelace'`.

Class Properties

We can access properties of an object using dot notation: `object_name.property_name`.

```
print(my_dog.name) # Prints Fido
```

In the above example, we stored our Dog instance in the variable `my_dog`, so `my_dog` is the name of our object. The property we're interested in printing is the dog's name so we place `name` after `my_dog` separated with a `.`.

Class Methods

Methods are just functions attached to objects. We can define a method by using the same syntax we use to define a function, but indenting it inside of a class definition. All methods take in the parameter `self`, referring to the current instance of the class.

```
class Dog:
    def __init__(self, name, breed, owner):
        self.name = name
        self.breed = breed
        self.owner = owner

    def call_dog(self):
        print(f"Here {self.name}!")
```

We can call a class method using the same dot notation we do for properties: `object_name.method_name()`.

```
my_dog.call_dog() # Prints 'Here Fido!'
```

Methods can take parameters in addition to self.

```
class Dog:
    def __init__(self, name, breed, owner):
        self.name = name
        self.breed = breed
        self.owner = owner

    def call_dog(self):
        print(f"Here {self.name}!")

    def command_trick(self, trick):
        print(f"{self.name}, {trick}!")
```

These parameters need to be passed arguments as we would with a normal function.

```
my_dog.command_trick("roll over") # Prints 'Fido, roll over!'
```

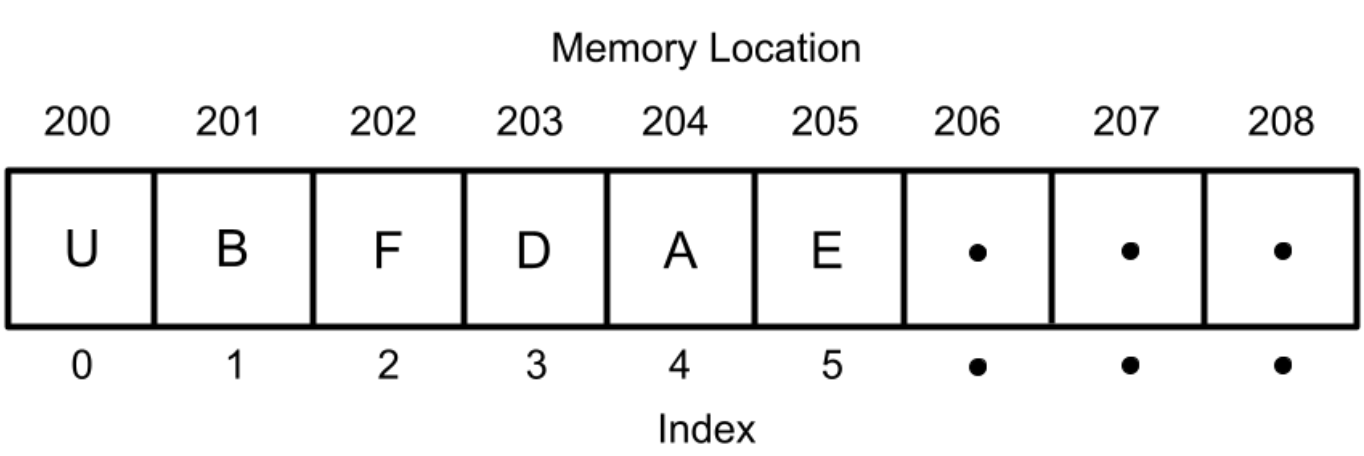
Linked Lists

Linked Lists vs Arrays

At a surface level, linked lists and Python lists are equivalent. They both allow us to store multiple items of any data type together as an ordered collection of data.

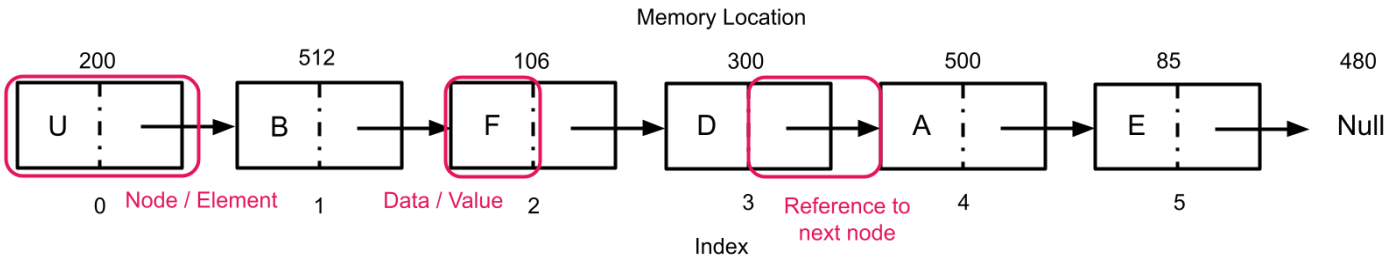
Linked lists and normal lists differ in how they store each item they contain within memory.

Normal Python lists store each consecutive element in an array. For example, say we have the list `['U', 'B', 'F', 'D', 'A', 'E']` with element `'U'` stored at memory location 200. Elements `'B'` through `'E'` would be stored at memory locations 201 through 205 respectively.



In contrast, each element of a linked list can be stored in non-related memory locations. The first item in a linked list might be stored at memory location 200, and the second item might be stored at memory location 512.

So that we can find each element of the list, each item in the list not only stores the data it wants to hold like `'U'` or `'B'`, but also a reference to where the next item in the list is stored in memory.

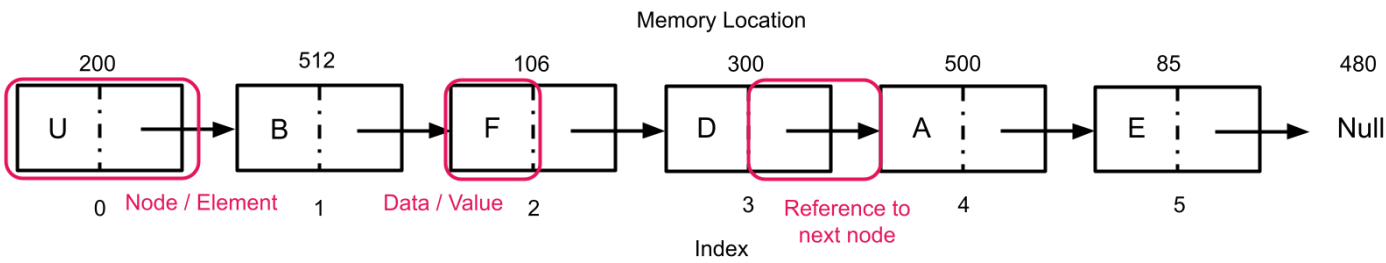


The reason both lists and linked lists exist, is because linked lists have a lower time and space complexity for certain operations. Follow your curiosity and do some independent research if you'd like to explore this more!

Defining a Linked List

Linked lists are not a built-in data type in Python, therefore we must define a class to build a linked list.

Each element of a linked list is called a *node*. A node consists of a *value* equivalent to an element's value in a normal list, as well as a *pointer* or reference variable to the next node in the list.



A `Node` class might commonly look like the following.

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

The `next` property is the pointer to the next node in the list. `next` is either another `Node` instance or `None` meaning there is no node after the current node. By default, `next` is `None`.

In technical interviewing scenarios, it is likely that you will only be provided with a `Node` class, and a variable containing the first node in the linked list (also called the head of the list).

In some scenarios, you may also be given a linked list class that includes a reference to the `head` of the list as an attribute and operations commonly performed on a list, such as appending a new node, as methods.

```
class LinkedList:
    # Constructor.
    def __init__(self):
        # The first node in the linked list.
        # The head is either a Node object or None if the list is empty.
        self.head = None

    # Method. Adds a new node with the specific data value to the beginning of the linked li
    def add_first(self, value):
        pass

    # Method. Adds a node with specified value to the end of the list.
    def append(self, value):
        pass

    # Method. Returns the length of the list.
    def length(self):
        pass

    # Method. Returns the value at a given index in the linked list.
    # Index count starts at 0.
    # Returns None if there are fewer nodes in the linked list than the index value.
    def get_at_index(self, index):
        pass
```

Linked List Traversal

Most interviewing problems involving linked list, will require you to traverse a linked list. Traversing a linked list means sequentially accessing each element of the list, starting with the first (head) node.

To perform a traversal, take the following steps:

1. Create a pointer `current` and initialize it to the first node in the list.
2. Create a while loop that continues iterating as long as the `current` node is not `None` (aka as long as there are still nodes in the list we haven't traversed).

3. In the body of the while loop:
    - perform whatever actions you would like to the current node (e.g. printing the current node's value)
    - update `current` to point at the next node in the linked list

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


    def traverse(head):
        current = head
        while current:
            # Perform operations on the current node (e.g., print the value)
            current = current.next # Move to the next node
```

Advanced Concepts

Multiple Pass Technique

Sometimes a linked list must be traversed multiple times in order to solve a problem. This is common when we need to extract some information about the list before we begin manipulating it. For example, there are many problems that require us to know the length of a linked list in order to solve the overall problem.

Example Usage:

The following function returns the values of a linked list as pairs in a list of tuples. If the linked list has an odd length, the values cannot be evenly paired and the function returns `None`.

```
Example Input List: 1->2->3->4
Example Input: head = 1
Expected Return Value: [(1, 2), (3, 4)]

Example Input List: 1->2->3
Example Input: head = 1
Expected Return Value: None
```

```
class Node:
    def __init__(self, val, next=None):
        self.value = val
        self.next = next

def pair_nodes(head):
    # Check if the list is empty
    if not head:
        return None

    # Pass 1: Get the length of the list
    current = head
    length = 0
    while current:
        length += 1
        current = current.next

    # If the length is odd, return None
    if length % 2 != 0:
        return None

    # Pass 2: If the length is even, create tuples of pairs
    pairs = []
    current = head
    while current:
        # Since we know the list length is even, current.next should always be valid here
        pairs.append((current.value, current.next.value))
        current = current.next.next  # Move to the next pair

    return pairs
```

Because we perform a constant number of traversals and each traversal is done in series, the time complexity of a function is not affected by conducting multiple consecutive passes over the list. If the length of our input list is `n`, the above function `pair_nodes()` performs `O(n + n)` operations which reduces to `O(2n)` which can be further reduced to `O(n)` operations.

Variations of this technique may also nest loops or extract certain traversals into helper functions. For example, we can rewrite our `pair_nodes()` function so that getting the length of the list is encapsulated in a helper function.

```
class Node:
    def __init__(self, val, next=None):
        self.value = val
        self.next = next

# Helper function: Returns the length of the list
def get_length(head):

    # Pass 1: Get the length of the list
    current = head
    length = 0
    while current:
        length += 1
        current = current.next
    return length

def pair_nodes(head):
    # Check if the list is empty
    if not head:
        return None

    # If the length is odd, return None
    if get_length(head) % 2 != 0: # Call helper function
        return None

    # Pass 2: If the length is even, create tuples of pairs
    pairs = []
    current = head
    while current:
        # Since we know the list length is even, current.next should always be valid here
        pairs.append((current.value, current.next.value))
        current = current.next.next  # Move to the next pair

    return pairs
```

Temporary Head Technique

When we solve linked list problems, we often encounter edge cases that involve the head of a linked list. Common edge cases involving the head of a linked list include:

- Deleting the current head of a linked list
- Adding a new element to the front of a linked list
- Reducing a linked list containing only one node to an empty list
- Adding a new node to an empty list

A common technique for handling these edge cases is creating a temporary head node. To apply this technique, an extra node object is created to serve as the temporary head of the list. We can use the temporary head as a placeholder that allows us to easily add items to an empty linked list or manipulate the actual head of a list as if it were any other node.

Example Usage:

The following function accepts the head of a linked list and a value and deletes the first node in the list with the specified value. It returns the head of the modified list.

```
Example Input List: 1->2->4
Example Input: head = 1, val = 2
Expected Return Value: 1
Expected Return List: 1->4
```

```
class Node:
    def __init__(self, val, next=None):
        self.value = val
        self.next = next

def delete_node(head, val):
    temp_head = Node("temp") # Initialize a temporary head node
    temp_head.next = head    # Point the temporary head at the input list

    # Traverse the list
    previous = temp_head
    current = head
    while current:
        if current.value == val:          # If the current node is the node to delete
            previous.next = current.next  # Delete the node
            break                         # Break out of the loop

        previous = current
        current = current.next
    return temp_head.next # Return the head of the input list
```

Because we created a temporary head, deleting the head of the list can be handled in the same way that deleting any other element in the list is handled. We can also proceed directly to traversing the list without checking if `head` is `None`, because the existence of the temporary head means there is at least one node int he list.

▶ **Deleting a node without a temporary head**

▶ ⚠️ **Temp Node Terminology**

Slow-Fast Pointer Technique

The slow-fast pointer technique, also called the tortoise and hare algorithm, is a common technique used to solve linked list and other data structure problems that involve cyclic structures. It involves using two pointers that move at different speeds to solve problems such as cycle detection, finding the middle of a list, or determining the meeting point within a cycle.

To apply the technique, create two pointers:

1. **Slow Pointer**: Increment one step at a time.
2. **Fast Pointer**: Increment two steps at a time.

Example Usage:

The following function returns `True` if a linked list has a cycle and `False` otherwise.


```
class Node:
    def __init__(self, val, next=None):
        self.value = val
        self.next = next

def has_cycle(head):
    if not head:
        return False

    slow = head  # Starts at the head
    fast = head  # Also starts at the head

    while fast and fast.next:
        slow = slow.next          # Move slow pointer by one
        fast = fast.next.next     # Move fast pointer by two

        if slow == fast:          # If they meet, there's a cycle
            return True

    return False  # If fast reaches the end, no cycle
```

Think about this algorithm as if you had a walker and a runner moving around a track. If the walker and runner start at the same time and the runner moves twice as fast as the walker, eventually the runner will lap the walker. If the walker and runner were instead moving along a straight path, the walker would never see the runner again!

Source: via GIPHY

If you think of the slow pointer as the walker and the fast pointer as the runner, we can apply the same logic to say that if the slow pointer and fast pointer have the same value, the fast pointer is "lapping" the slow pointer and there must be a cycle in the list.

Bonus Syntax & Concepts

The following concepts are nice to know and may improve your code readability or help you solve certain problems more easily and efficiently. However, they are not *required* to solve any of the problems in this unit. These concepts are **not in scope for either the Standard or Advanced Unit**

5 assessments, and you do not need to memorize them! Click on each concept to read more about how to use it.

- `lst.extend(x)` Modifies the given list by appending all elements in iterable `x`.
- `lst.reverse()` Reverses the given list.