TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Spring 2025 (@ Section 3 | Tuesdays and Thursdays 6PM - 8PM PT)
Personal Member ID#: 117667

Session 1: OOP & Linked Lists

Session Overview

In this session, students will learn to apply Python classes and linked lists through practical exercises. They will begin by creating and manipulating instances of a class and then explore the basics of linked lists, focusing on node creation and linkage. These exercises aim to deepen understanding of object-oriented programming and provide foundational skills in managing custom data structures in Python.

> You can find all resources from today including session slide decks, session recordings, and more on the resources tab

🎢 Part 1 : Instructor Led Session

We'll spend the first portion of the synchronous class time in large groups, where the instructor will lead class instruction for 30-45 minutes.

👨‍💻 Part 2: Breakout Session

In breakout sessions, we will explore and collaboratively solve problem sets in small groups. Here, the **collaboration, conversation, and approach** are just as important as “solving the problem” - please engage warmly, clearly, and plentifully in the process!

In breakout rooms you will:

- Screen-share the problem/s, and verbally review them together
- Screen-share an interactive coding environment, and talk through the steps of a solution approach
    - ProTip: - An Integrated Development Environment (IDE) is a fancy name for a tool you could use for shared writing of code - like Replit.com, Collabed.it, CodePen.io, or other - your staff team will specify which tool to use for this class!
- Screen-share an implementation of your proposed solution
- Independently follow-along, or create an implementation, in your own IDE.

Your program leader/s will indicate which code sharing tool/s to use as a group, and will help break down or provide specific scaffolding with the main concepts above.

▶ **Note on Expectations**

🔎 Problem Solving Approach

To build a long-term organized approach to problem solving, we'll start with three main steps. We'll refer to them as **UPI: Understand, Plan, and Implement**.

We'll apply these three steps to most of the problems we'll see in the first half of the course.

We will learn to:

- **Understand** the problem,
- **Plan** a solution step-by-step, and
- **Implement** the solution

▶ **Comment on UPI**

▶ **UPI Example**

Breakout Problems Session 1

▶ Standard Problem Set Version 1

▶ Standard Problem Set Version 2

▼ Advanced Problem Set Version 1

Problem 1: Villager Class

A class constructor is a special method or function that is used to create and initialize a new object from a class. Define the class constructor `__init__()` for a new class `Villager` that represents characters in the game Animal Crossing. The constructor accepts three required arguments: strings `name`, `species`, and `catchphrase`. The constructor defines four properties for a `Villager`:

- `name`, a string initialized to the argument `name`
- `species`, a string initialized to the argument `species`
- `catchphrase`, a string initialized to the argument `catchphrase`
- `furniture`, a list initialized to an empty list

```
class Villager:
    def __init__(self, name, species, catchphrase):
        self.name = name
        self.species = species
        self.catchphrase = catchphrase
        self.furniture = []
```

Example Usage:

```
apollo = Villager("Apollo", "Eagle", "pah")
print(apollo.name)
print(apollo.species)
print(apollo.catchphrase)
print(apollo.furniture)
```

Output:

```
Apollo
Eagle
pah
[]
```

▶ 💡 **Hint: Intro to Object Oriented Programming**

Problem 2: Add Furniture

Players and villagers in Animal Crossing can add furniture to their inventory to decorate their house.

Update the `Villager` class with a new method `add_item()` that takes in one parameter, `item_name`.

The method should validate the `item_name`.

- If the item is valid, add `item_name` to the villager's `furniture` attribute.
- The method does not need to return any values.

`item_name` is valid if it has one of the following values: `"acoustic guitar"`, `"ironwood kitchenette"`, `"rattan armchair"`, `"kotatsu"`, or `"cacao tree"`.

```
class Villager:
    # ... methods from previous problems

    def add_item(self, item_name):
        pass
```

Example Usage:

```
alice = Villager("Alice", "Koala", "guvnor")
print(alice.furniture)

alice.add_item("acoustic guitar")
print(alice.furniture)

alice.add_item("cacao tree")
print(alice.furniture)

alice.add_item("nintendo switch")
print(alice.furniture)
```

Output:

```
[]
["acoustic guitar"]
["acoustic guitar", "cacao tree"]
["acoustic guitar", "cacao tree"]
```

▶ 💡 **Hint: Class Methods**

Problem 3: Group by Personality

The `Villager` class has been updated below to include the new string attribute `personality` representing the character's personality type.

Outside of the `Villager` class, write a *function* `of_personality_type()`. Given a list of `Villager` instances `townies` and a string `personality_type` as parameters, return a list containing the *names* of all villagers in `townies` with `personality` `personality_type`. Return the names in any order.

```
class Villager:
    def __init__(self, name, species, personality, catchphrase):
        self.name = name
        self.species = species
        self.personality = personality
        self.catchphrase = catchphrase
        self.furniture = []
    # ... methods from previous problems

def of_personality_type(townies, personality_type):
    pass
```

Example Usage:

```
isabelle = Villager("Isabelle", "Dog", "Normal", "what's up?")
bob = Villager("Bob", "Cat", "Lazy", "pthhhpth")
stitches = Villager("Stitches", "Cub", "Lazy", "stuffin'")

print(of_personality_type([isabelle, bob, stitches], "Lazy"))
print(of_personality_type([isabelle, bob, stitches], "Cranky"))
```

Example Output:

```
['Bob', 'Stitches']
[]
```

Problem 4: Telephone

The `Villager` constructor has been updated to include an additional attribute `neighbor`. A villager's `neighbor` is another `Villager` instance and represents their closest neighbor. By default, a `Villager`'s neighbor is set to `None`.

Given two `Villager` instances `start_villager` and `target_villager`, write a function `message_received()` that returns `True` if you can pass a message from the `start_villager` to the `target_villager` through a series of neighbors and `False` otherwise.

```
class Villager:
    def __init__(self, name, species, personality, catchphrase, neighbor=None):
        self.name = name
        self.species = species
        self.personality = personality
        self.catchphrase = catchphrase
        self.furniture = []
        self.neighbor = neighbor
    # ... methods from previous problems

def message_received(start_villager, target_villager):
    pass
```

Example Usage:

```
isabelle = Villager("Isabelle", "Dog", "Normal", "what's up?")
tom_nook = Villager("Tom Nook", "Raccoon", "Cranky", "yes, yes")
kk_slider = Villager("K.K. Slider", "Dog", "Lazy", "dig it")
isabelle.neighbor = tom_nook
tom_nook.neighbor = kk_slider

print(message_received(isabelle, kk_slider))
print(message_received(kk_slider, isabelle))
```

Example Output:

```
True
Example 1 Explanation: Isabelle can pass a message to her neighbor, Tom Nook. Tom No
message to his neighbor, KK Slider. KK Slider is the target, therefore the function

False
Example 2 Explanation: KK Slider doesn't have a neighbor, so you cannot pass a messa
KK Slider.
```

Problem 5: Linked Up

A **linked list** is a new data type that, similar to a normal list or array, allows us to store pieces of data sequentially. The difference between a linked list and a normal list lies in how each element is stored in a computer's memory.

In a normal list, individual elements of the list are stored in adjacent memory locations according to the order they appear in the list. If we know where the first element of the list is stored, it's really easy to find any other element in the list.

In a linked list, the individual elements called **nodes** are not stored in sequential memory locations. Each node may be stored in an unrelated memory location. To connect nodes together into a sequential list, each node stores a reference or pointer to the next node in the list.

Connect the provided node instances below to create the linked list `kk_slider -> harriet -> saharah -> isabelle`.

A function `print_linked_list()` which accepts the **head**, or first element, of a linked list and prints the values of the list has also been provided for testing purposes.

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

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

kk_slider = Node("K.K. Slider")
harriet = Node("Harriet")
saharah = Node("Saharah")
isabelle = Node("Isabelle")

# Add code here to link the above nodes
```

Example Usage:

```
print_linked_list(kk_slider)
```

Example Output:

```
K.K. Slider -> Harriet -> Saharah -> Isabelle
```

▶ 💡 **Hint: Intro to Linked Lists**

Problem 6: Got One!

Imagine that behind the scenes, Animal Crossing uses a linked list to represent the order fish will appear to a player who is fishing in the river. The `head` of the list represents the next fish that a player will catch if they keep fishing.

Write a function `catch_fish()` that accepts the `head` of a list. The function should:

1. Print the name of the fish in the `head` node using the format `"I caught a <fish name>!"`.
2. Remove the first node in the list.

The function should return the new head of the list. If the list is empty, print `"Aw! Better luck next time!"` and return `None`.

A function `print_linked_list()` which accepts the **head**, or first element, of a linked list and prints the list data has also been provided for testing purposes.

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

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.fish_name, end=" -> " if current.next else "\n")
        current = current.next

def catch_fish(head):
    pass
```

Example Usage:

```
fish_list = Node("Carp", Node("Dace", Node("Cherry Salmon")))
empty_list = None

print_linked_list(fish_list)
print_linked_list(catch_fish(fish_list))
print(catch_fish(empty_list))
```

Example Output:

```
Carp -> Dace -> Cherry Salmon
I caught a Carp!
Dace -> Cherry Salmon
Aw! Better luck next time!
None
```

Problem 7: Fishing Probability

Imagine that Animal Crossing is still using a linked list to represent the order fish will appear to a player who is fishing in the river! The `head` of the list represents the next fish that a player will catch if they keep fishing.

Write a function `fish_chances()` that accepts the `head` of a list and a string `fish_name`. Return the probability rounded down to the nearest hundredth that the player will catch a fish of type `fish_name`.

A function `print_linked_list()` which accepts the **head**, or first element, of a linked list and prints the list data has also been provided for testing purposes.

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

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.fish_name, end=" -> " if current.next else "\n")
        current = current.next

def fish_chances(head, fish_name):
    pass
```

Example Usage:

```
fish_list = Node("Carp", Node("Dace", Node("Cherry Salmon")))
print(fish_chances(fish_list, "Dace"))
print(fish_chances(fish_list, "Rainbow Trout"))
```

Example Output:

```
0.33
0.00
```

▶ 💡 **Hint: Linked List Traversal**

Problem 8: Restocking the Lake

Imagine that Animal Crossing is still using a linked list to represent the order fish will appear to a player who is fishing! The `head` of the list represents the next fish that a player will catch if they keep fishing.

Write a function `restock()` that accepts the `head` of a linked list and a string `new_fish`, and adds a Node with the `fish_name` `new_fish` to the end of the list. Return the `head` of the modified list.

A function `print_linked_list()` which accepts the **head**, or first element, of a linked list and prints the list data has also been provided for testing purposes.

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

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.fish_name, end=" -> " if current.next else "\n")
        current = current.next

def restock(head, new_fish):
    pass
```

Example Usage:

```
fish_list = Node("Carp", Node("Dace", Node("Cherry Salmon")))
print_linked_list(restock(fish_list, "Rainbow Trout"))
```

Example Output:

```
Carp -> Dace -> Cherry Salmon -> Rainbow Trout
```

Close Section

▾ Advanced Problem Set Version 2

Problem 1: Player Class II

A class constructor is a special method or function that is used to create and initialize a new object from a class. Define the class constructor `__init__()` for a new class `Player` that represents Mario Kart players. The constructor accepts two required arguments: strings `character` and `kart`. The constructor should define three properties for a `Player`:

- `character`, a string initialized to the argument `character`

- `kart`, a string initialized to the argument `kart`
- `items`, a list initialized to an empty list

```
class Player:
    def __init__(self, character, kart):
        pass
```

Example Usage:

```
player_one = Player("Yoshi", "Super Blooper")
print(player_one.character)
print(player_one.kart)
print(player_one.items)
```

Example Output:

```
Yoshi
Super Blooper
[]
```

▶ 💡 **Hint: Intro to Object Oriented Programming**

Problem 2: Add Special Item

Players can pick up special items as they race.

Update the `Player` class with a new method `add_item()` that takes in one parameter, `item_name`.

The method should validate the `item_name`.

- If the item is valid, add `item_name` to the player's `items` attribute.
- The method does not need to return any values.

`item_name` is valid if it has one of the following values: `"banana"`, `"green shell"`, `"red shell"`, `"bob-omb"`, `"super star"`, `"lightning"`, `"bullet bill"`.

```
class Player:
    def __init__(self, character, kart):
        self.character = character
        self.kart = kart
        self.items = []

    def add_item(self, item_name):
        pass
```

Example Usage:

```
player_one = Player("Yoshi", "Dolphin Dasher")
print(player_one.items)

player_one.add_item("red shell")
print(player_one.items)

player_one.add_item("super star")
print(player_one.items)

player_one.add_item("super smash")
print(player_one.items)
```

Example Output:

```
[]
['red shell']
['red shell', 'super star']
['red shell', 'super star']
```

▶ 💡 **Hint: Class Methods**

Problem 3: Race Results

Given a list `race_results` of `Player` objects where the first player in the list came first in the race, the second player in the list came second, etc., write a function `print_results()` that prints the players in place.

```
class Player:
    def __init__(self, character, kart):
        self.character = character
        self.kart = kart
        self.items = []
    # ... methods from previous problems

def print_results(race_results):
    pass
```

Example Usage:

```
peach = Player("Peach", "Daytripper")
mario = Player("Mario", "Standard Kart M")
luigi = Player("Luigi", "Super Blooper")
race_one = [peach, mario, luigi]

print_results(race_one)
```

Example Output:

```
1. Peach
2. Mario
3. Luigi
```

Problem 4: Get Rank

The `Player` class has been updated below with a new attribute `ahead` to represent the player currently directly ahead of them in the race.

Write a function `get_place()` that accepts a `Player` object `my_player` and returns their current place number in the race.

```
class Player:
    def __init__(self, character, kart, opponent=None):
        self.character = character
        self.kart = kart
        self.items = []
        self.ahead = opponent

def get_place(my_player):
    pass
```

Example Usage:

```
peach = Player("Peach", "Daytripper")
mario = Player("Mario", "Standard Kart M", peach)
luigi = Player("Luigi", "Super Blooper", mario)

player1_rank = get_place(luigi)
player2_rank = get_place(peach)
player3_rank = get_place(mario)

print(player1_rank)
print(player2_rank)
print(player3_rank)
```

Example Output:

```
3
1
2
```

Problem 5: Daisy Chain

A **linked list** is a new data type that, similar to a normal list or array, allows us to store pieces of data sequentially. The difference between a linked list and a normal list lies in how each element is stored in a computer's memory.

In a normal list, individual elements of the list are stored in adjacent memory locations according to the order they appear in the list. If we know where the first element of the list is stored, it's really easy to find any other element in the list.

In a linked list, the individual elements called **nodes** are not stored in sequential memory locations. Each node may be stored in an unrelated memory location. To connect nodes together into a sequential list, each node stores a reference or pointer to the next node in the list.

Connect the provided node instances below to create the linked list `daisy -> peach -> luigi -> mario`.

A function `print_linked_list()` which accepts the **head**, or first element, of a linked list has also been provided for testing purposes.

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

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

daisy = Node("Daisy")
peach = Node("Peach")
luigi = Node("Luigi")
mario = Node("Mario")

# Add code here to link the above nodes
```

Example Usage:

```
print_linked_list(daisy)
```

Example Output:

```
Daisy -> Peach -> Luigi -> Mario
```

▶ 💡 **Hint: Intro to Linked Lists**

Problem 6: Count Racers

Imagine a linked list used to track the order in which Mario Kart players finished in a race. The `head` of the list represents the first place finisher, and the tail or last node in the list represents the last place finisher.

Write a function `count_racers()` that accepts the `head` of the list and returns the number of players who participated in the race.

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

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.player_name, end=" -> " if current.next else "\n")
        current = current.next

def count_racers(head):
    pass
```

Example Usage:

```
racers1 = Node("Mario", Node("Peach", Node("Luigi", Node("Daisy"))))
racers2 = Node("Mario")

print(count_racers(racers1))
print(count_racers(racers2))
print(count_racers(None))
```

Example Output:

```
4
1
0
```

▶ 💡 **Hint: Linked List Traversal**

Problem 7: Last Place

Imagine a linked list used to track the order in which Mario Kart players finished in a race. The `head` of the list represents the first place finisher, and the tail or last node in the list represents the last place finisher.

Given the `head` of the list, write a function `last_place()` that returns the `player_name` of the player that finished last in the race. If the list is empty, return `None`.

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

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.player_name, end=" -> " if current.next else "\n")
        current = current.next

def last_place(head):
        pass
```

Example Usage:

```
racers1 = Node("Mario", Node("Peach", Node("Luigi", Node("Daisy"))))
racers2 = Node("Mario")

print(last_place(racers1))
print(last_place(racers2))
print(last_place(None))
```

Example Output:

```
Daisy
Mario
None
```

Problem 8: Update Rankings

A 1-indexed linked list is used to track the overall standings of players in a Mario Kart tournament. Write a function `increment_rank()` that accepts the `head` of the list and an index `target`. The function should swap the order of the nodes at index `target` and index `target - 1`. If `target` is the first node in the list, return the original list. Otherwise, return the `head` of the modified list.

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

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.player_name, end=" -> " if current.next else "\n")
        current = current.next

def increment_rank(head, target):
    pass
```

Example Usage:

```
Example 1:
racers1 = Node("Mario", Node("Peach", Node("Luigi", Node("Daisy"))))
racers2 = Node("Mario", Node("Luigi"))

print_linked_list(increment_rank(racers1, 3))
print_linked_list(increment_rank(racers2, 1))
print_linked_list(increment_rank(None, 1))
```

Example Output:

```
Mario -> Luigi -> Peach -> Daisy
Mario -> Luigi
None
```

Close Section