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

Problem 1: New Horizons

Step 1: Copy the following code into your IDE.

Step 2: Instantiate an instance of the class `Villager`, which represents characters in Animal Crossing. Store the instance in a variable named `apollo`.

- The `Villager` object created should have the name `"Apollo"`, the species `"Eagle"`, and the catchphrase `"pah"`.

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

# Instantiate your villager here
```

Example Usage:

```
print(apollo.name)
print(apollo.species)
print(apollo.catchphrase)
print(apollo.furniture)
```

Example Output:

```
Apollo
Eagle
pah
[]
```

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

Problem 2: Greet Player

Step 1: Using the `Villager` class from Problem 1, add the following `greet_player()` method to your existing code:

```
def greet_player(self, player_name):
    return f"{self.name}: Hey there, {player_name}! How's it going, {self.catchphras
```

Step 2: Create a second instance of `Villager` in a variable named `bones`.

- The `Villager` object created should have `name` `"Bones"`, `species` `"Dog"`, and `catchphrase` `"yip yip"`.

Step 3: Call the method `greet_player()` with your name and print out `"Bones: Hey there, <your name>! How's it going, yip yip!"`. For example, if your name is Tram, `"Bones: Hey there, Tram! How's it going, yip yip?"` would be printed out to the console.

Example Usage:

```
print(bones.name)
print(bones.species)
print(bones.catchphrase)
print(bones.furniture)
```

Example Output:

```
Bones
Dog
yip yip
[]
```

▶ 💡 **Hint: Class Methods**

Problem 3: Update Catchphrase

In Animal Crossing, as players become friends with villagers, the villagers might ask the player to suggest a new catchphrase.

Adding on to your existing code, update `bones` so that his catchphrase is `"ruff it up"` instead of its current value, `"yip yip"`.

Example Usage:

```
print(bones.greet_player("Samia"))
```

Example Output:

```
Bones: Hey there, Samia! How's it going, ruff it up!
```

▶ 💡 **Hint: Class Attributes**

Problem 4: Set Character

In the previous exercise, we accessed and modified a player's `catchphrase` attribute directly. Instead of allowing users to update their player directly, it is common to create **setter methods** that users can call to update class attributes. This has a few different benefits, including allowing us to validate data before updating our class instance.

Update your `Villager` class with a method `set_catchphrase()` that takes in one parameter `new_catchphrase`.

- If `new_catchphrase` is valid, it should update the villager's `catchphrase` attribute to have value `new_catchphrase` and print `"Catchphrase updated"`.
- Otherwise, it should print out `"Invalid catchphrase"`.

Valid catchphrases are less than 20 characters in length. They must all contain only alphabetic and whitespace characters.

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

        def set_catchphrase(self, new_catchphrase):
                pass
```

Example Usage:

```
alice = Villager("Alice", "Koala", "guvnor")

alice.set_catchphrase("sweet dreams")
print(alice.catchphrase)
alice.set_catchphrase("#?!")
print(alice.catchphrase)
```

Example Output:

```
Example 1:
Catchphrase Updated!
sweet dreams
Invalid catchphrase
sweet dreams
```

Problem 5: 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 player'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

    # New method
    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)
```

Example Output:

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

▶ 💡 **Hint: Writing Methods**

Problem 6: Print Inventory

Update the `Villager` class with a method `print_inventory()` that accepts no parameters except for self.

The method should print the name and quantity of each item in a villager's `furniture` list.

- The name and quantity should be in the format `"item1: quantity, item2: quantity, item3: quantity"` for however many unique items exist in the villager's furniture list
- If the player has no items, the function should print `"Inventory empty"`.

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

    def print_inventory(self):
        # Implement the method here
        pass
```

Example Usage:

```
alice = Villager("Alice", "Koala", "guvnor")

alice.print_inventory()

alice.furniture = ["acoustic guitar", "ironwood kitchenette", "kotatsu", "kotatsu"]
alice.print_inventory()
```

Example Output:

```
Inventory empty
acoustic guitar: 1, ironwood kitchenette: 1, kotatsu: 2
```

Problem 7: 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 8: 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
```

```
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
```

Problem 9: Nook's Cranny

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.

Using the provided `Node` class below, create a linked list `Tom Nook -> Tommy` where the instance `tom_nook` has value `"Tom Nook"` which points to the instance `tommy` that has value `"Tommy"`.

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

Example Usage:

```
tom_nook = Node("Tom Nook")
tommy = Node("Tommy")
tom_nook.next = tommy
print(tom_nook.value)
print(tom_nook.next.value)
print(tommy.value)
print(tommy.next)
```

Example Output:

```
Tom Nook
Tommy
Tommy
None
```

▶ 💡 **Hint: Intro to Linked Lists**

Problem 10: Timmy and Tommy

In a linked list, pointers can be redirected to any place in the list.

Using the linked list from Problem 9, create a new Node `timmy` with value `"Timmy"` and place it between `tom_nook` and `tommy` so the new linked list is `tom_nook -> timmy -> tommy`.

Example Usage:

```
print(tom_nook.value)
print(tom_nook.next.value)
print(timmy.value)
print(timmy.next.value)
print(tommy.value)
print(tommy.next)
```

Example Output:

```
Tom Nook
Timmy
Timmy
Tommy
Tommy
None
```

Problem 11: Saharah

Using the linked list from Problem 10, remove the `tom_nook` node and add in a node `saharah` with value `"Saharah"` to the end of the list so that the resulting list is `timmy -> tommy -> saharah`.

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

Example Usage

```
print(tom_nook.next)
print(timmy.value)
print(timmy.next.value)
print(tommy.value)
print(tommy.next.value)
print(saharah.value)
print(saharah.next)
```

Example Output:

```
None
Timmy
Tommy
Tommy
Saharah
Saharah
None
```

Problem 12: Print List

Write a function `print_list()` that takes in the head of a linked list and returns a string linking together the **values** of the list with the separator `"->"`.

Note: The "head" of a linked list is the first element in the linked list. Equivalent to `lst[0]` of a normal list.

Example Usage:

```
isabelle = Node("Isabelle")
saharah = Node("Saharah")
cj = Node("C.J.")

isabelle.next = saharah
saharah.next = cj

print(print_list(isabelle))
```

Example Output:

```
Isabelle -> Saharah -> C.J.
```

▶ 💡 **Hint: Linked List Traversal**

Close Section

▼ Standard Problem Set Version 2

Problem 1: Player Class

Step 1: Copy the following code into your IDE.

Step 2: Instantiate an instance of the class `Player` and store it in a variable named `player_one`.

- The `Player` object should have the `character` `"Yoshi"` and the `kart` `"Super Blooper"`.

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

Example Usage:

```
player_one.character
player_one.kart
player_one.items
```

Example Output:

```
Yoshi
Super Blooper
[]
```

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

Problem 2: Get Player

Step 1: Using the `Player` class from Problem 1, add the following `get_player()` method to your existing code:

```
def get_player(self):
    return f"{self.character} driving the {self.kart}"
```

Step 2: Create a second instance of `Player` in a variable named `player_two`.

- The `Player` object created should have `character` `"Bowser"` and `kart` `"Pirahna Prowler"`.

Step 3: Use the method `get_player()` below to print out `"Match: Yoshi driving the Super Blooper vs Bowser driving the Pirahna Prowler"`.

Example Usage:

```
player_two.character
player_two.kart
player_two.items
```

Example Output:

```
Bowser
Pirahna Prowler
[]
```

▶ 💡 **Hint: Class Methods**

Problem 3: Update Kart

Players might want to update their choice of kart for their next race.

Update `player_one` so that their kart is `"Dolphin Dasher"` instead of its current value, `"Super Blooper"`.

Example Usage:

```
print(player_one.get_player())
```

Example Output:

```
Yoshi driving the Dolphin Dasher
```

▶ 💡 **Hint: Class Attributes**

Problem 4: Set Character

In the previous exercise, we accessed and modified a player's `kart` attribute directly. Instead of allowing users to update their player directly, it is common to create **setter methods** that users can call to update class attributes. This has a few different benefits, including allowing us to validate data before updating our class instance.

Update your `Player` class with a method `set_character()` that takes in one parameter `name`.

- If `name` is valid, it should update the player's `character` attribute to have value `name` and print `"Character updated"`.
- Otherwise, it should print out `"Invalid character"`.

Valid character names are `"Mario"`, `"Luigi"`, `"Peach"`, `"Yoshi"`, `"Toad"`, `"Wario"`, `"Donkey Kong"`, and `"Bowser"`.

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

    def set_character(self, name):
        pass
```

Example Usage:

```
player_three = Player("Donkey Kong", "Standard Kart")

player_three.set_character("Peach")
print(player_three.character)
player_three.set_character("Baby Peach")
print(player_three.character)
```

Example Output:

```
Character Updated
Peach
Invalid Character
Peach
```

Problem 5: 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', 'super smash']
```

▶ 💡 **Hint: Writing Methods**

Problem 6: Print Inventory

Update the `Player` class with a method `print_inventory()` that accepts no parameters except for self.

The method should print the name and quantity of each item in a player's items list.

- If the player has no items, the function should print `"Inventory empty"`.

```
class Player():
    # ... methods from previous problems

    def print_inventory(self):
        pass
```

Example Usage:

```
player_one = Player("Yoshi", "Super Blooper")
player_one.items = ["banana", "bob-omb", "banana", "super star"]
player_two = Player("Peach", "Dolphin Dasher")

player_one.print_inventory()
player_two.print_inventory()
```

Example Output:

```
Inventory: banana: 2, bob-omb: 1, super star: 1
Inventory empty
```

Problem 7: Race Results

Given a list `race_results` of `Player` objects where the first player in the list came first in the race, 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 8: 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_rank()` 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_rank(my_player):
    pass
```

Example Usage:

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

print(get_rank(luigi))
print(get_rank(peach))
print(get_rank(mario))
```

Example Output:

```
3
1
2
```

Problem 9: Tom and Jerry

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.

Using the provided `Node` class below, create a linked list `cat -> mouse` where the instance `cat` has value `"Tom"` which points to the instance `mouse` that has value `"Jerry"`.

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

Example Usage:

```
print(cat.value)
print(cat.next)
print(cat.next.value)
print(mouse.value)
print(mouse.next)
```

Example Output:

```
Tom
mouse
Jerry
Jerry
None
```

▶ 💡 **Hint: Intro to Linked Lists**

Problem 10: Chase List

In a linked list, pointers can be redirected at any place in the list.

Using the linked list from Problem 9, create a new Node `dog` with value `"Spike"` and point it to the `cat` node so that the full list now looks like `dog -> cat -> mouse`.

Example Usage:

```
print(dog.value)
print(dog.next)
print(dog.next.value)
print(cat.next)
print(cat.next.value)
print(mouse.next.value)
```

Example Output:

```
Spike
cat
Tom
mouse
Jerry
None
```

Problem 11: Update Chase

Using the linked list from Problem 10, remove the `dog` node and add in a node `cheese` with value `"Gouda"` to the end of the list so that the resulting list is `cat -> mouse -> cheese`.

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

Problem 12: Chase String

Write a function `chase_list()` that takes in the head of a linked list and returns a string linking together the **values** of the list with the separator `"chases"`.

Note: The "head" of a linked list is the first element in the linked list, equivalent to `lst[0]` of a normal list.

Example Usage:

```
dog = Node("Spike")
cat = Node("Tom")
mouse = Node("Jerry")
cheese = Node("Gouda")

dog.next = cat
cat.next = mouse
mouse.next = cheese

print(chase_list(dog))
```

Example Output: `"Spike chases Tom chases Jerry chases Gouda"`

▶ 💡 **Hint: Linked List Traversal**

Close Section

- **Advanced Problem Set Version 1**
- **Advanced Problem Set Version 2**