CYBR 330: Team Hawk

Inventory Management

Zach Madison, Seiya Genda, Kwanho Kwon

The Benchmark

This projects benchmark is an inventory management system that is developed by Google's Gemini AI. The purpose of this is to highlight the shortcomings of pure AI development, and to show how output can be improved. The shortcomings of the benchmark is as follows. Systems on top of systems; when tasked with adding new features the benchmark, instead of integrating into existing systems, it tends toward not change existing code which results in tech debt. This can even be seen in the menus, instead of intuitively moving the exit button to another number, it simply added an option after exit. Secondly, the AI will follow instructions in implementation but will not always develop in a way that makes sense. This can be seen in category implantation, the program would originally create category nodes, but those nodes did not store any items; instead, the items stored the category node. While this technically works, it does not take advantage of the benefits of a category tree, nor is it intuitive to work with. Third, the implementation always relies on pythons' default modules and never develops original systems unless it has to; this can be seen in use of default python sort. Finally, the benchmark will never try to make optimizations that are not surface level; all programming made by the AI relies on the easiest solution to develop, this is likely because most programmers do not go out of their way to optimize systems.

Changes From Benchmark

Our group has developed focused on four primary improvements from the benchmark.

Binary Search: Written by Kwanho Kwon, Modified by Zach Madison

Our widest reaching change is the implementation of binary search. This was done by including a second dynamic array that stores the names of items and their indexes. This second array is always sorted alphabetically, allowing us to replace item searches with binary searches. This effects the functions add_item, remove_item, and edit_item.

Heap Sort: Written by Seiya Genda

Our second change is heap sort implementation; this is replacing default python sort, which is Timsort. This allows us to save memory using an in-place sort. Additionally, it remains the same time complexity, O(n log(n)).

Recursion: Written by Kwanho Kwon

Previously, the "display_category_tree" function used recursion. Our implementation of this function has replaced recursion with looping. This change should both bypass python's recursion limit and save time by removing function call overhead.

Inventory Search With Category Filter: Written by Zach Madison

The original implementation of category filter in the inventory search function searched through all items and checked the category_path variable in them. This is inefficient because in most situations very few items should be included in the filter. Our implementation of the function altered the category tree to be able to store items; using this, the program now searches the category tree for the correct branch and shows all items under that branch. This retains the same time complexity but is significantly faster in most reasonable scenarios.

Analysis of Improvements

Binary Search

Add Item

The original implementation ran in O(n) the new iteration also runs in O(n), This is because the original has two O(n) calls, one being an item search and the other being a category search. Our implementation saves time overall by turning the first O(n) search into a O(log(n)) binary search, this has also been shown in performance analysis testing.

Remove Item

This function remains O(n) but usually runs slower in the new implementation because it's forced to take on additional O(n) calls to save time elsewhere.

Edit Item

This function originally ran in O(n) time but now runs in O(log(n)). This time save is due to replacing a search with the binary search.

Heap Sort

Our implementation of heap sort remains at the same time complexity, O(nlog(n)). However, heapsort does save on space from being an in-place sort. This changes the sort from O(n) additional space required to O(1) additional space required.

Recursion

Our removal of recursion removes the python recursion limit. Additionally, this should save time by reducing function call overhead. However, this has not shown in testing. During testing our implementation has shown to be slower by a very small margin. This could either be margin of error in testing, issues with timing method, or due to how this change was implemented.

Inventory Search With Category Filter

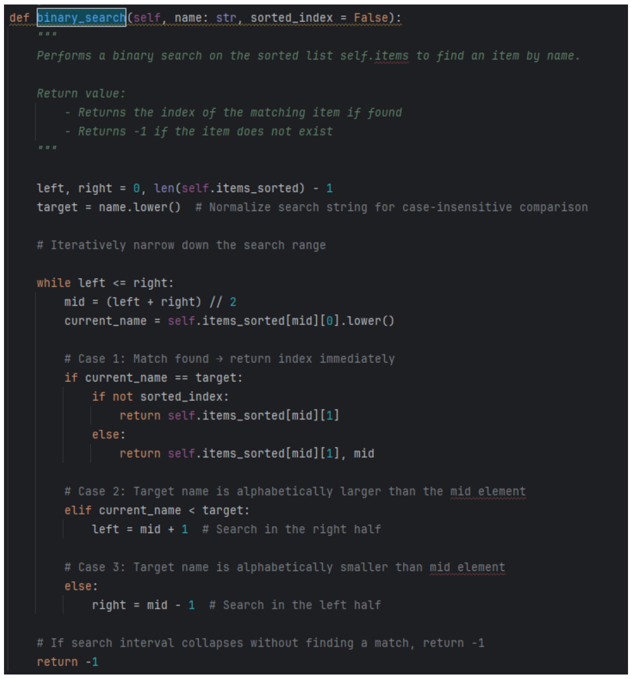
While this change retains the same time complexity, it usually saves time by searching for the correct branch, cutting out many of the items that would normally be searched. The original implementation searched the entire item list for items under the correct branch. This iteration instead looks for the correct branch and prints the items on it.

Conclusion

In the end, our group was able to successfully optimize the inventory management benchmark. The greatest lesson our group learned was that the best way to optimize a program is often by considering solutions specific to the program. The best example of this is the category filter. This change does not save time through theoretical analysis, but in practical scenarios this solution saves more time than any other change. Attempting to force an optimization where it does not already fit in can create a very extensive and taxing programming process. For example, our implementation of binary sort needed more infrastructure built around it, which forced tradeoffs. A much simpler solution designed for practical scenarios would have likely shown better results, such as a hash map. To conclude, our group was able to create effective optimizations, and the best optimizations that we were able to find were the ones designed with program use in mind.

Screenshots of Code

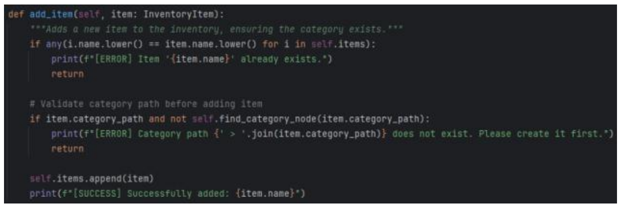
Binary Search



```python
def binary_search(self, name: str, sorted_index = False):
    """
    Performs a binary search on the sorted list self.items to find an item by name.

    Return value:
        - Returns the index of the matching item if found
        - Returns -1 if the item does not exist
    """

    left, right = 0, len(self.items_sorted) - 1
    target = name.lower()  # Normalize search string for case-insensitive comparison

    # Iteratively narrow down the search range

    while left <= right:
        mid = (left + right) // 2
        current_name = self.items_sorted[mid][0].lower()

        # Case 1: Match found → return index immediately
        if current_name == target:
            if not sorted_index:
                return self.items_sorted[mid][1]
            else:
                return self.items_sorted[mid][1], mid

        # Case 2: Target name is alphabetically larger than the mid element
        elif current_name < target:
            left = mid + 1  # Search in the right half

        # Case 3: Target name is alphabetically smaller than mid element
        else:
            right = mid - 1  # Search in the left half

    # If search interval collapses without finding a match, return -1
    return -1
```

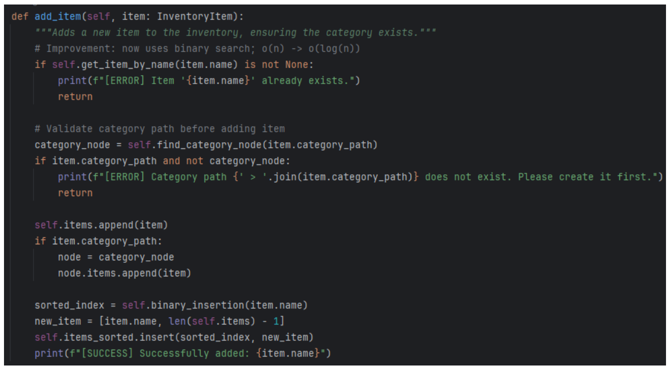
Add Item

Old



```python
def add_item(self, item: InventoryItem):
    """Adds a new item to the inventory, ensuring the category exists."""
    if any(i.name.lower() == item.name.lower() for i in self.items):
        print(f"[ERROR] Item '{item.name}' already exists.")
        return

    # Validate category path before adding item
    if item.category_path and not self.find_category_node(item.category_path):
        print(f"[ERROR] Category path {' > '.join(item.category_path)} does not exist. Please create it first.")
        return

    self.items.append(item)
    print(f"[SUCCESS] Successfully added: {item.name}")
```

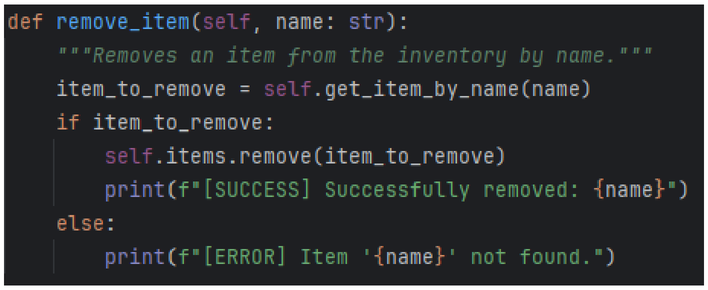
New



```python
def add_item(self, item: InventoryItem):
    """Adds a new item to the inventory, ensuring the category exists."""
    # Improvement: now uses binary search; o(n) -> o(log(n))
    if self.get_item_by_name(item.name) is not None:
        print(f"[ERROR] Item '{item.name}' already exists.")
        return

    # Validate category path before adding item
    category_node = self.find_category_node(item.category_path)
    if item.category_path and not category_node:
        print(f"[ERROR] Category path {' > '.join(item.category_path)} does not exist. Please create it first.")
        return

    self.items.append(item)
    if item.category_path:
        node = category_node
        node.items.append(item)

    sorted_index = self.binary_insertion(item.name)
    new_item = [item.name, len(self.items) - 1]
    self.items_sorted.insert(sorted_index, new_item)
    print(f"[SUCCESS] Successfully added: {item.name}")
```

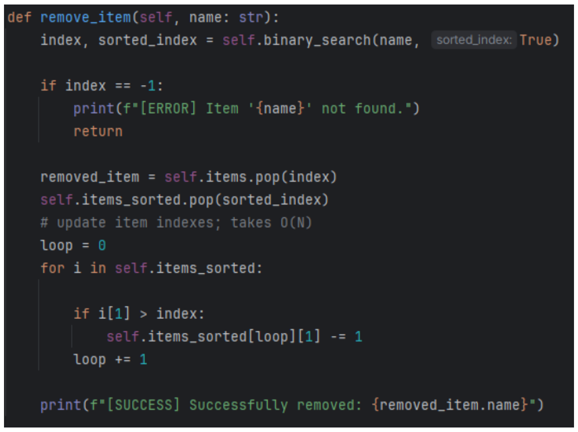
Remove Item

Old



```python
def remove_item(self, name: str):
    """Removes an item from the inventory by name."""
    item_to_remove = self.get_item_by_name(name)
    if item_to_remove:
        self.items.remove(item_to_remove)
        print(f"[SUCCESS] Successfully removed: {name}")
    else:
        print(f"[ERROR] Item '{name}' not found.")
```

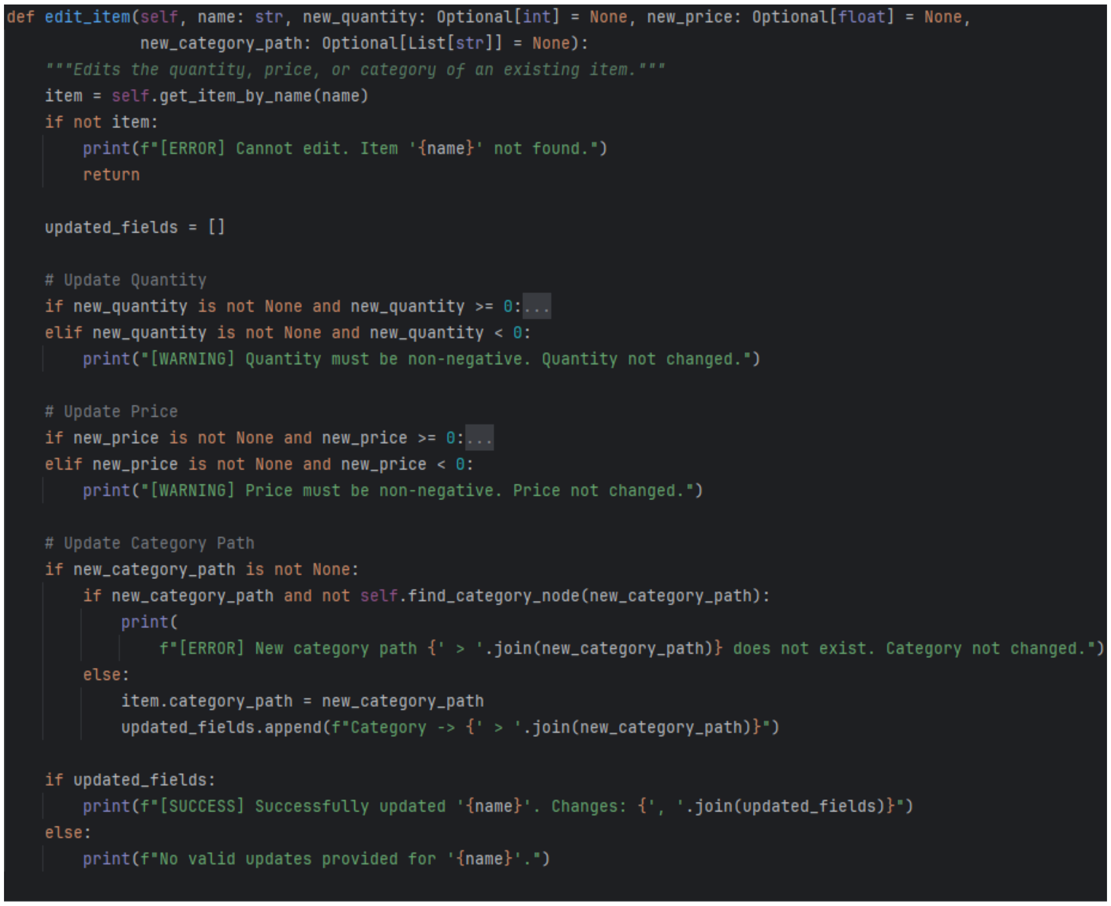
New



```python
def remove_item(self, name: str):
    index, sorted_index = self.binary_search(name,  sorted_index: True)

    if index == -1:
        print(f"[ERROR] Item '{name}' not found.")
        return

    removed_item = self.items.pop(index)
    self.items_sorted.pop(sorted_index)
    # update item indexes; takes O(N)
    loop = 0
    for i in self.items_sorted:

        if i[1] > index:
            self.items_sorted[loop][1] -= 1
        loop += 1

    print(f"[SUCCESS] Successfully removed: {removed_item.name}")
```

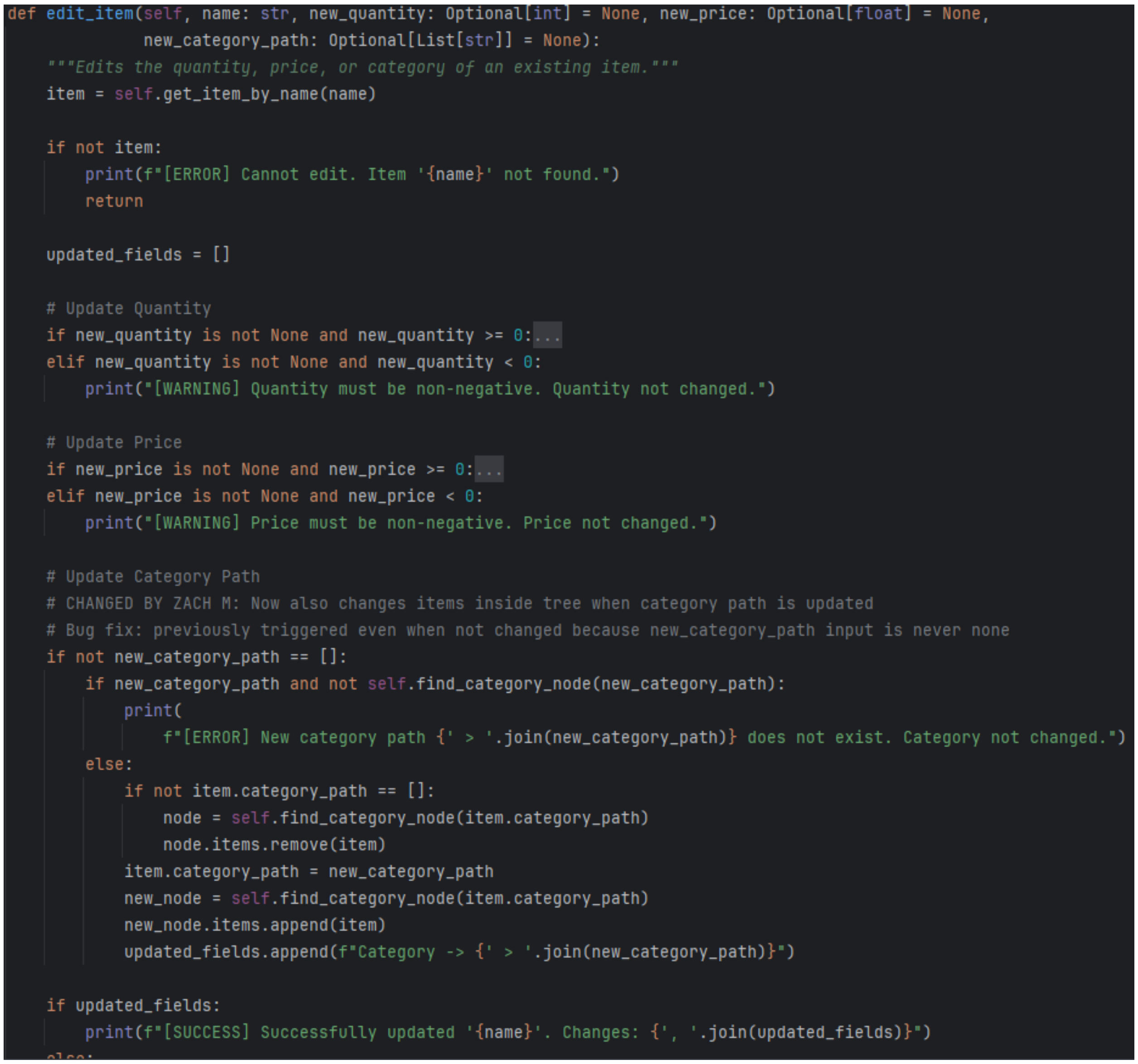
Edit Item

Old



```python
def edit_item(self, name: str, new_quantity: Optional[int] = None, new_price: Optional[float] = None,
              new_category_path: Optional[List[str]] = None):
    """Edits the quantity, price, or category of an existing item."""
    item = self.get_item_by_name(name)
    if not item:
        print(f"[ERROR] Cannot edit. Item '{name}' not found.")
        return

    updated_fields = []

    # Update Quantity
    if new_quantity is not None and new_quantity >= 0:...
    elif new_quantity is not None and new_quantity < 0:
        print("[WARNING] Quantity must be non-negative. Quantity not changed.")

    # Update Price
    if new_price is not None and new_price >= 0:...
    elif new_price is not None and new_price < 0:
        print("[WARNING] Price must be non-negative. Price not changed.")

    # Update Category Path
    if new_category_path is not None:
        if new_category_path and not self.find_category_node(new_category_path):
            print(
                f"[ERROR] New category path {' > '.join(new_category_path)} does not exist. Category not changed.")
        else:
            item.category_path = new_category_path
            updated_fields.append(f"Category -> {' > '.join(new_category_path)}")

    if updated_fields:
        print(f"[SUCCESS] Successfully updated '{name}'. Changes: {', '.join(updated_fields)}")
    else:
        print(f"No valid updates provided for '{name}'.")
```

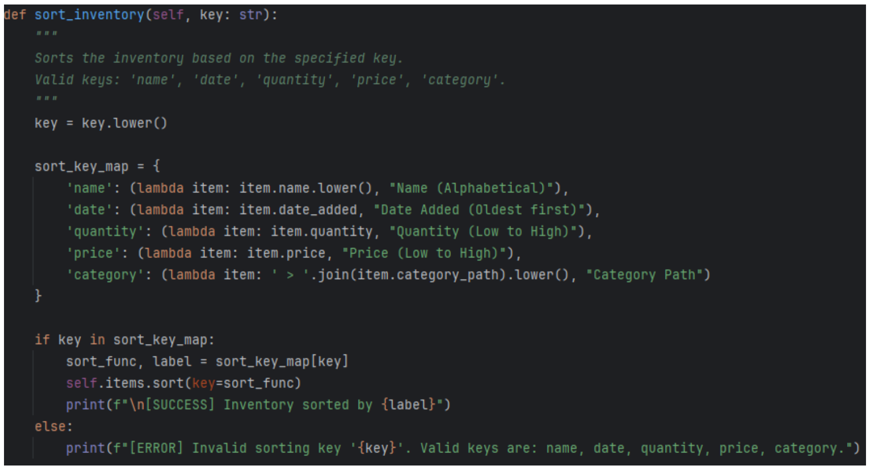
New



```python
def edit_item(self, name: str, new_quantity: Optional[int] = None, new_price: Optional[float] = None,
              new_category_path: Optional[List[str]] = None):
    """Edits the quantity, price, or category of an existing item."""
    item = self.get_item_by_name(name)

    if not item:
        print(f"[ERROR] Cannot edit. Item '{name}' not found.")
        return

    updated_fields = []

    # Update Quantity
    if new_quantity is not None and new_quantity >= 0:...
    elif new_quantity is not None and new_quantity < 0:
        print("[WARNING] Quantity must be non-negative. Quantity not changed.")

    # Update Price
    if new_price is not None and new_price >= 0:...
    elif new_price is not None and new_price < 0:
        print("[WARNING] Price must be non-negative. Price not changed.")

    # Update Category Path
    # CHANGED BY ZACH M: Now also changes items inside tree when category path is updated
    # Bug fix: previously triggered even when not changed because new_category_path input is never none
    if not new_category_path == []:
        if new_category_path and not self.find_category_node(new_category_path):
            print(
                f"[ERROR] New category path {' > '.join(new_category_path)} does not exist. Category not changed.")
        else:
            if not item.category_path == []:
                node = self.find_category_node(item.category_path)
                node.items.remove(item)
            item.category_path = new_category_path
            new_node = self.find_category_node(item.category_path)
            new_node.items.append(item)
            updated_fields.append(f"Category -> {' > '.join(new_category_path)}")

    if updated_fields:
        print(f"[SUCCESS] Successfully updated '{name}'. Changes: {', '.join(updated_fields)}")
```

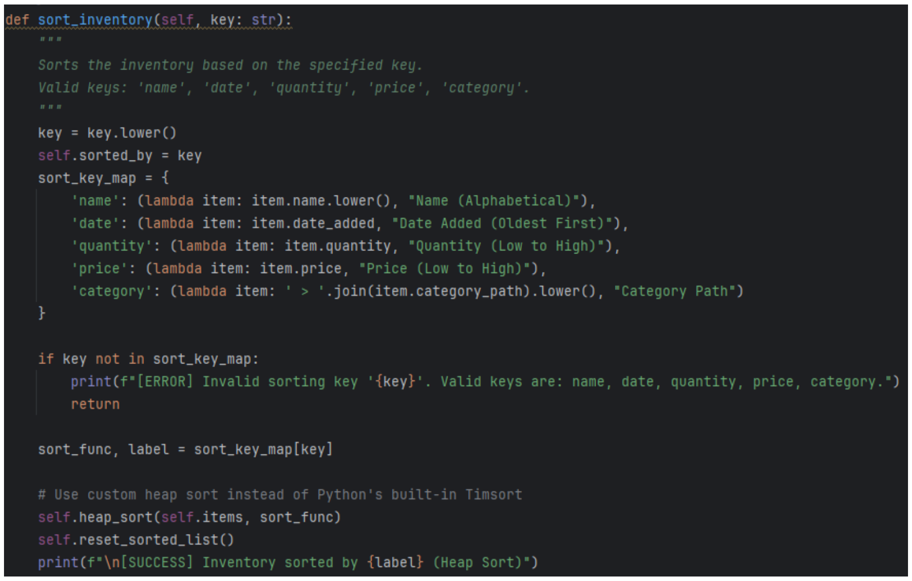
Sort

Old



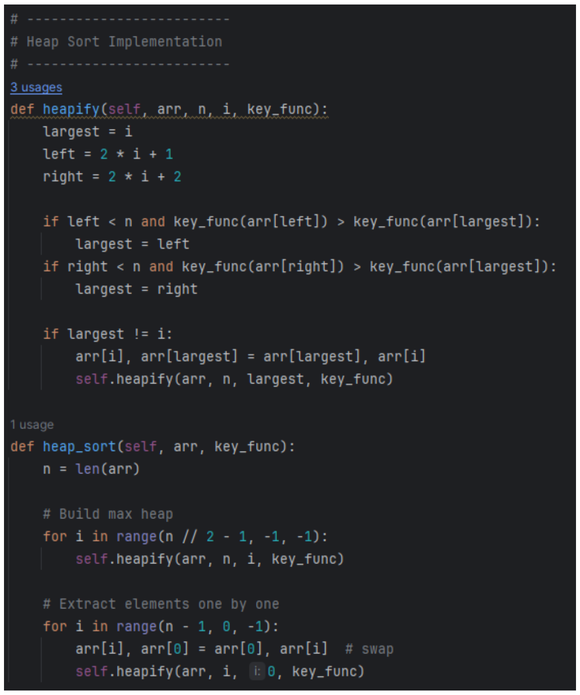
```python
def sort_inventory(self, key: str):
    """
    Sorts the inventory based on the specified key.
    Valid keys: 'name', 'date', 'quantity', 'price', 'category'.
    """
    key = key.lower()

    sort_key_map = {
        'name': (lambda item: item.name.lower(), "Name (Alphabetical)"),
        'date': (lambda item: item.date_added, "Date Added (Oldest first)"),
        'quantity': (lambda item: item.quantity, "Quantity (Low to High)"),
        'price': (lambda item: item.price, "Price (Low to High)"),
        'category': (lambda item: ' > '.join(item.category_path).lower(), "Category Path")
    }

    if key in sort_key_map:
        sort_func, label = sort_key_map[key]
        self.items.sort(key=sort_func)
        print(f"\n[SUCCESS] Inventory sorted by {label}")
    else:
        print(f"[ERROR] Invalid sorting key '{key}'. Valid keys are: name, date, quantity, price, category.")
```

New



```python
def sort_inventory(self, key: str):
    """
    Sorts the inventory based on the specified key.
    Valid keys: 'name', 'date', 'quantity', 'price', 'category'.
    """
    key = key.lower()
    self.sorted_by = key
    sort_key_map = {
        'name': (lambda item: item.name.lower(), "Name (Alphabetical)"),
        'date': (lambda item: item.date_added, "Date Added (Oldest First)"),
        'quantity': (lambda item: item.quantity, "Quantity (Low to High)"),
        'price': (lambda item: item.price, "Price (Low to High)"),
        'category': (lambda item: ' > '.join(item.category_path).lower(), "Category Path")
    }

    if key not in sort_key_map:
        print(f"[ERROR] Invalid sorting key '{key}'. Valid keys are: name, date, quantity, price, category.")
        return

    sort_func, label = sort_key_map[key]

    # Use custom heap sort instead of Python's built-in Timsort
    self.heap_sort(self.items, sort_func)
    self.reset_sorted_list()
    print(f"\n[SUCCESS] Inventory sorted by {label} (Heap Sort)")
```

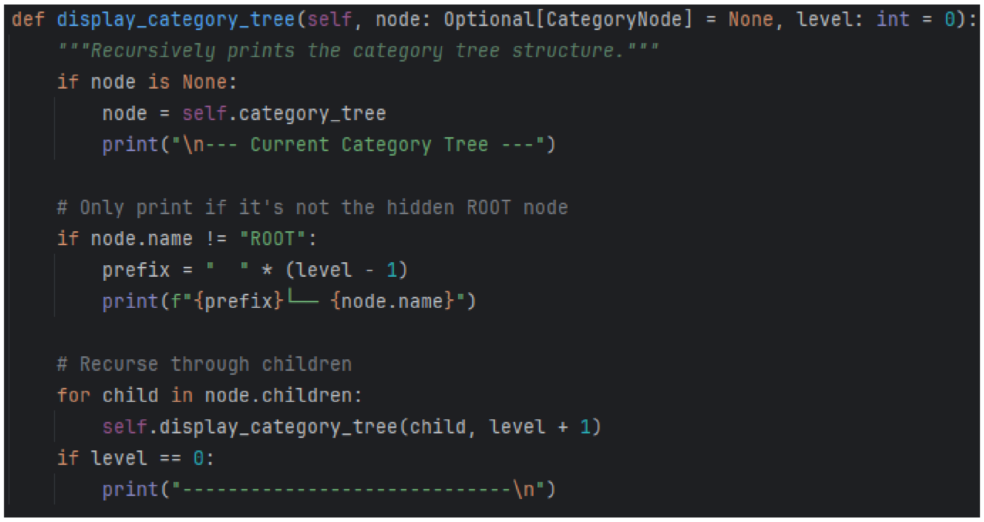
Heap Sort



```
# ------------------------
# Heap Sort Implementation
# ------------------------
3 usages
def heapify(self, arr, n, i, key_func):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and key_func(arr[left]) > key_func(arr[largest]):
        largest = left
    if right < n and key_func(arr[right]) > key_func(arr[largest]):
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        self.heapify(arr, n, largest, key_func)

1 usage
def heap_sort(self, arr, key_func):
    n = len(arr)

    # Build max heap
    for i in range(n // 2 - 1, -1, -1):
        self.heapify(arr, n, i, key_func)

    # Extract elements one by one
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # swap
        self.heapify(arr, i, i: 0, key_func)
```

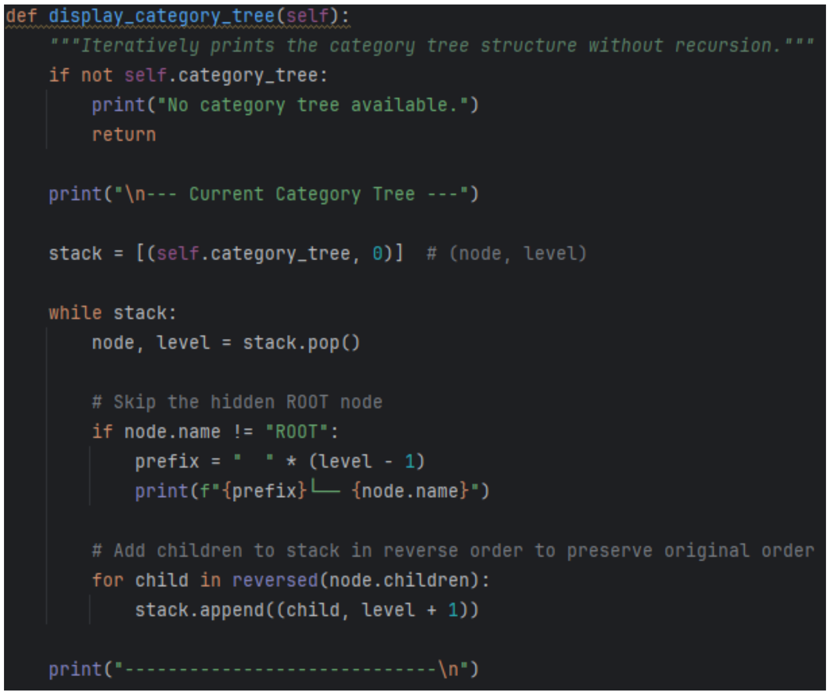
Category Display

Old



```
def display_category_tree(self, node: Optional[CategoryNode] = None, level: int = 0):
    """Recursively prints the category tree structure."""
    if node is None:
        node = self.category_tree
        print("\n--- Current Category Tree ---")

    # Only print if it's not the hidden ROOT node
    if node.name != "ROOT":
        prefix = "  " * (level - 1)
        print(f"{prefix}└── {node.name}")

    # Recurse through children
    for child in node.children:
        self.display_category_tree(child, level + 1)
    if level == 0:
        print("-----------------------------\n")
```

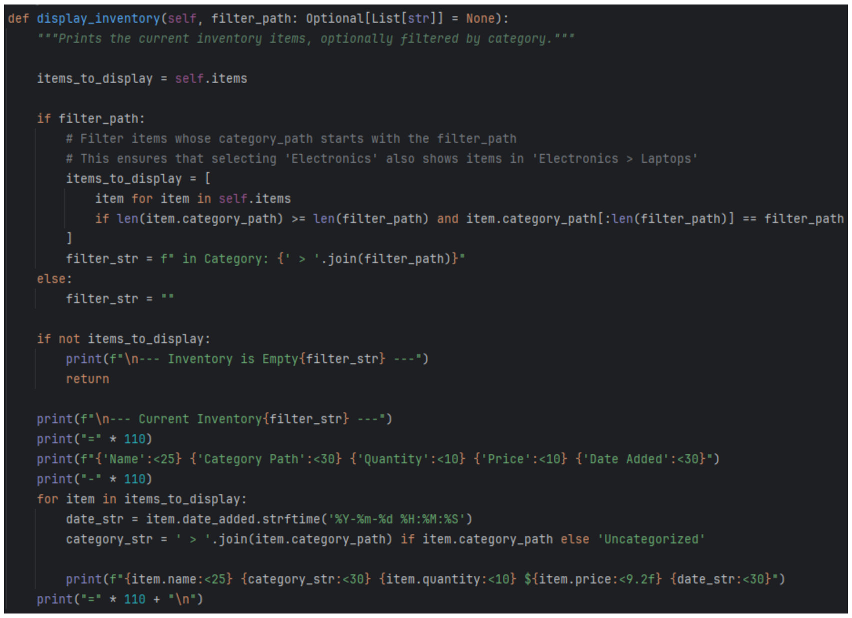
New



```python
def display_category_tree(self):
    """Iteratively prints the category tree structure without recursion."""
    if not self.category_tree:
        print("No category tree available.")
        return

    print("\n--- Current Category Tree ---")

    stack = [(self.category_tree, 0)]  # (node, level)

    while stack:
        node, level = stack.pop()

        # Skip the hidden ROOT node
        if node.name != "ROOT":
            prefix = "  " * (level - 1)
            print(f"{prefix}└── {node.name}")

        # Add children to stack in reverse order to preserve original order
        for child in reversed(node.children):
            stack.append((child, level + 1))

    print("-----------------------------\n")
```

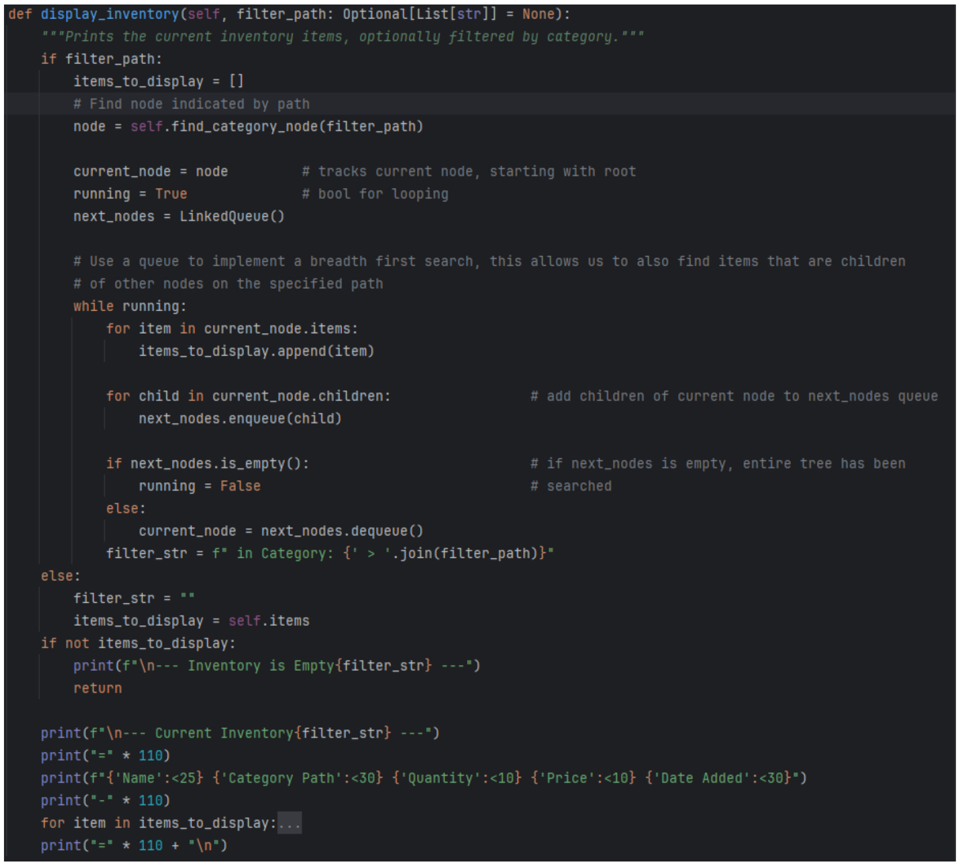
Item Display

Old



```python
def display_inventory(self, filter_path: Optional[List[str]] = None):
    """Prints the current inventory items, optionally filtered by category."""

    items_to_display = self.items

    if filter_path:
        # Filter items whose category_path starts with the filter_path
        # This ensures that selecting 'Electronics' also shows items in 'Electronics > Laptops'
        items_to_display = [
            item for item in self.items
            if len(item.category_path) >= len(filter_path) and item.category_path[:len(filter_path)] == filter_path
        ]
        filter_str = f" in Category: {' > '.join(filter_path)}"
    else:
        filter_str = ""

    if not items_to_display:
        print(f"\n--- Inventory is Empty{filter_str} ---")
        return

    print(f"\n--- Current Inventory{filter_str} ---")
    print("=" * 110)
    print(f"{'Name':<25} {'Category Path':<30} {'Quantity':<10} {'Price':<10} {'Date Added':<30}")
    print("-" * 110)
    for item in items_to_display:
        date_str = item.date_added.strftime('%Y-%m-%d %H:%M:%S')
        category_str = ' > '.join(item.category_path) if item.category_path else 'Uncategorized'

        print(f"{item.name:<25} {category_str:<30} {item.quantity:<10} ${item.price:<9.2f} {date_str:<30}")
    print("=" * 110 + "\n")
```

New



```python
def display_inventory(self, filter_path: Optional[List[str]] = None):
    """Prints the current inventory items, optionally filtered by category."""
    if filter_path:
        items_to_display = []
        # Find node indicated by path
        node = self.find_category_node(filter_path)

        current_node = node         # tracks current node, starting with root
        running = True              # bool for looping
        next_nodes = LinkedQueue()

        # Use a queue to implement a breadth first search, this allows us to also find items that are children
        # of other nodes on the specified path
        while running:
            for item in current_node.items:
                items_to_display.append(item)

            for child in current_node.children:                 # add children of current node to next_nodes queue
                next_nodes.enqueue(child)

            if next_nodes.is_empty():                           # if next_nodes is empty, entire tree has been
                running = False                                 # searched
            else:
                current_node = next_nodes.dequeue()
            filter_str = f" in Category: {' > '.join(filter_path)}"
    else:
        filter_str = ""
        items_to_display = self.items
    if not items_to_display:
        print(f"\n--- Inventory is Empty{filter_str} ---")
        return

    print(f"\n--- Current Inventory{filter_str} ---")
    print("=" * 110)
    print(f"{'Name':<25} {'Category Path':<30} {'Quantity':<10} {'Price':<10} {'Date Added':<30}")
    print("-" * 110)
    for item in items_to_display:...
    print("=" * 110 + "\n")
```