Prerequisite Tutorial: Python Concepts for Depth First Search

This interactive guide covers the fundamental Python data structures and advanced concepts used in the Depth First Search (DFS) algorithm (`dfs_refactor.py`). **For each section, run the provided code snippets to verify your understanding.**

1. Fundamental Python Building Blocks (Required Knowledge)

The DFS code relies heavily on these basic Python types.

Lists, Tuples, and Dictionaries

- **Lists (`[]`):** Ordered, mutable (changeable) collections used for storing the `path` (sequence of actions). Lists allow duplicates.
- **Tuples (`()`):** Ordered, *immutable* (unchangeable) collections. Used to bundle data together safely, like a node's state and its corresponding path: `(state, path)`.
- **Dictionaries (`{}`):** Unordered key-value pairs. Used internally by the search framework to define states and transitions.

Practice Task 1.1: Objects in Lists and Membership

We store complex data in our main structure.

Concept	Explanation	Code Example
Objects in Lists	Lists can hold any Python object. Our Stack holds tuples, and those tuples contain lists.	`stack_item = (10, ['North', 'East'])`
Membership Check	Checking if an element is present in a list or set.	`if state not in expanded_states:`

Code to Run:

```
# A list representing a search queue
search_queue = []

# State 'A' reached via path ['N']
state_A = (1, 1)
path_A = ['N']
search_queue.append((state_A, path_A))

# State 'B' reached via path ['N', 'E']
state_B = (1, 2)
path_B = ['N', 'E']
search_queue.append((state_B, path_B))

print("Is state_A in the queue? (Wrong check):", state_A in search_queue)
print("Correct check (The item itself):", (state_A, path_A) in search_queue)
```

Expected Output:

```
Is state_A in the queue? (Wrong check): False
Correct check (The item itself): True
```

2. Abstract Data Type: The Stack (LIFO)

DFS explores the deepest branch first. This requires a **Stack**, which is a **Last-In, First-Out (LIFO)** structure. Imagine a stack of dinner plates: you can only add to (Push) or take from (Pop) the top.

Concept	Explanation	Code Snippet
Push	Adds a new item to the *top* of the stack.	`dfs_stack.push(item)`
Pop	Removes and returns the item currently at the *top* of the stack.	`item = dfs_stack.pop()`
Check Empty	Checks if the Stack has any nodes left to process.	`while not dfs_stack.isEmpty():`

Practice Task 2.1: LIFO Behavior

Simulate the LIFO behavior using a basic Stack implementation.

Code to Run:

```
import util # Assumed utility library provided by the framework
my_stack = util.Stack()

print("1. Pushing A, B, C")
my_stack.push("A")
my_stack.push("B")
my_stack.push("C")

print("2. Pop 1:", my_stack.pop())
print("3. Pop 2:", my_stack.pop())
print("4. Is stack empty:", my_stack.isEmpty())
print("5. Pop 3:", my_stack.pop())
print("6. Is stack empty:", my_stack.isEmpty())
```

Expected Output:

```
1. Pushing A, B, C
2. Pop 1: C
3. Pop 2: B
4. Is stack empty: False
5. Pop 3: A
6. Is stack empty: True
```

3. Advanced Set Usage for Graph Search

When tracking visited states in a large graph, the speed of checking "Have I been here before?" is critical. We use the Python `set` for (instant) lookup time.

Why not a List for Visited States?

Structure	Check Time
List (`[]`)	**Slow ():** Must check every single item in the list.
Set (`{}`)	**Fast ():** Can instantly determine if an item is present.

Practice Task 3.1: Set Operations

Run this code to see the core set operations used in DFS.

Code to Run:

```
expanded_states = set()
state_X = (5, 5)
state_Y = (1, 2)

# 1. Add a state when it's expanded
expanded_states.add(state_X)
expanded_states.add(state_Y)
print("Set after adding:", expanded_states)

# 2. Check a known state (Fast Lookup)
print("Is state_X already expanded?", state_X in expanded_states)

# 3. Check a new state
state_Z = (0, 0)
print("Is state_Z already expanded?", state_Z in expanded_states)
```

Expected Output:

```
Set after adding: {(5, 5), (1, 2)}
Is state_X already expanded? True
Is state_Z already expanded? False
```

4. Critical Error Avoidance: Path Mutability (The Copying Issue)

This is the most critical source of error. **Lists are mutable** (changeable). If you use `append()`, all references to that list are changed, which breaks backtracking logic.

Bug Example (DO NOT USE `append()` for path creation)

We will use the list slice operator (`[:]`) or concatenation (`+`) to create a **shallow copy**, ensuring each path branch is unique.

Method	Outcome

BUG (append)	Modifies the *original* list object.
FIX (concatenation)	Creates a *new* list object.

Practice Task 4.1: Path Mutability vs. Copying

Run the code below to see how `append()` corrupts shared lists, while concatenation creates safe, new lists.

Code to Run:

```
# Path for Node A
path_A = ['North', 'East']

# --- BUGGY WAY (MUTATION) ---
path_B_bug = path_A # path_B_bug now points to the SAME list object as path_A
path_B_bug.append('South')
print(f"MUTATION: path_A is now {path_A}")


# --- CORRECT WAY (CONCATENATION) ---
path_C = ['North', 'East']
path_D_fix = path_C + ['South'] # Creates a BRAND NEW list object
print(f"COPY FIX: path_C remains {path_C}")
print(f"COPY FIX: path_D is new {path_D_fix}")
```

Expected Output:

```
MUTATION: path_A is now ['North', 'East', 'South']
COPY FIX: path_C remains ['North', 'East']
COPY FIX: path_D is new ['North', 'East', 'South']
```

Conclusion: We must use `new_path = path_so_far + [direction]` to safely extend the path.

5. Tuple Unpacking

This is an efficient Python feature that allows us to instantly assign the components of a tuple to variables.

Practice Task 5.1: Unpacking

Simulate unpacking the item popped from the Stack, and the successor tuple returned by the search problem.

Code to Run:

```
# 1. Unpacking the Stack Item (State and Path)
stack_item = ((10, 20), ['North', 'North'])
current_state, path_so_far = stack_item

print("State unpacked:", current_state)
print("Path unpacked:", path_so_far)
```

```
# 2. Unpacking a Successor Item (State, Direction, Cost)
successor_data = ((10, 21), 'East', 1.0)
successor_state, direction, _ = successor_data

print("\nSuccessor State:", successor_state)
print("Direction:", direction)
print("Cost (Ignored by _):", _)
```

Expected Output:

```
State unpacked: (10, 20)
Path unpacked: ['North', 'North']

Successor State: (10, 21)
Direction: East
Cost (Ignored by _): 1.0
```

6. Putting It All Together: The Final DFS Algorithm

This task combines the Stack (Section 2), the Set (Section 3), and Path Copying (Section 4) to implement the complete Depth First Search Graph Algorithm.

Final DFS Code (`dfs_refactor.py`)

```
def depthFirstSearch(problem):
    """Search the deepest nodes in the search tree first."""

    # 1. Stack Initialization (Section 2)
    dfs_stack = util.Stack()
    start_state = problem.getStartState()
    dfs_stack.push((start_state, [])) # Pushing the start state and an empty
path

    # 2. Set Initialization (Section 3)
    # Tracks states we have already processed (expanded) to avoid loops
    expanded_states = set()

    # Begin the LIFO loop
    while not dfs_stack.isEmpty():

        # 3. Pop the deepest node (LIFO) (Section 2, 5)
        current_state, path_so_far = dfs_stack.pop()

        # 4. Check if the goal is found
        if problem.isGoalState(current_state):
            return path_so_far

        # 5. Check if already expanded (Section 3)
        # If we have already fully explored this state, skip it
        if current_state in expanded_states:
            continue

        # Mark state as expanded *after* pulling it from the stack
        expanded_states.add(current_state)

        # 6. Explore successors
        # Successors are (successor_state, direction, cost) (Section 5)
```

```
        for successor_state, direction, _ in
problem.getSuccessors(current_state):

            # Only consider states we haven't expanded yet
            if successor_state not in expanded_states:

                # 7. CRITICAL: Create new path using concatenation (Section 4)
                # This prevents mutation bugs.
                new_path = path_so_far + [direction]

                # 8. Push the new node onto the stack (LIFO behavior)
                dfs_stack.push((successor_state, new_path))

    return [] # Should only run if the goal is unreachable
```

Explanation by Section

Code Lines	Concept	Purpose in DFS
`dfs_stack = util.Stack()`	**The Stack (Section 2)**	Holds nodes pending exploration, ensuring LIFO (deepest-first) behavior.
`expanded_states = set()`	**The Set (Section 3)**	Provides fast lookups to check if a state has been processed, making it a "Graph Search" algorithm.
`current_state, path_so_far = dfs_stack.pop()`	**Tuple Unpacking (Section 5)**	Extracts the state and its accumulated path from the popped tuple.
`if current_state in expanded_states:`	**Set Lookup (Section 3)**	Checks if we've already done the work for this state. If so, we `continue` to the next node on the stack.
`new_path = path_so_far + [direction]`	**Path Concatenation (Section 4)**	**Fixes the mutation bug!** Creates a new, unique path object for the successor node to guarantee correct backtracking.
`dfs_stack.push((successor_state, new_path))`	**The Stack (Section 2)**	Adds the newly found path to the top of the stack, ready to be explored immediately (Depth First).

This comprehensive review, complete with the final algorithm and concept references, should fully prepare your students for the lesson!