Looping Graph

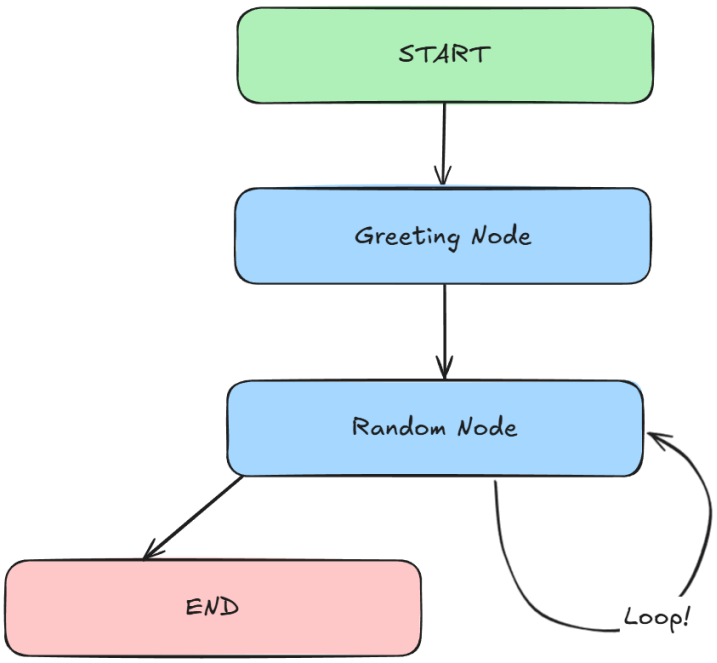
> A **looping graph** is a LangGraph workflow where **one or more nodes call themselves again** (directly or indirectly) until a certain condition is met.

Why Loops are Useful in LangGraph?

Loops are common in AI workflows when:

- **Multi-turn conversations**: Keep asking the user for info until enough is collected.
- **Iterative refinement**: Keep improving a document until quality is good.
- **Retry on failure**: Repeat a step until it works.
- **Agent planning**: Keep reasoning and taking actions until goal is met.

Aim:



- In greeting node, user will state their name
  - Output: Hi, *name*
- Random Node→ Generate 5 random numbers

```
from langgraph.graph import StateGraph, END
import random
from typing import TypedDict, List, Dict
```

```
class AgentState(TypedDict):
    name : str
    number : List[int]
    counter: int
```

```
def greeting_node(state: AgentState) → AgentState:
    """Greeting Node which says hi to the person"""

    state['name'] = f"hi there, {state['name']}"
    state['counter'] = 0
    return state
```

```
def greeting_node(state: AgentState) → AgentState:
    """Greeting Node which says hi to the person"""

    state['name'] = f"hi there, {state['name']}"
    state['counter'] = 0
    return state

def random_node(state: AgentState) → AgentState:
    """Generates a random number from 0 to 10"""
```

```
    state['number'].append(random.randint(0,10))
    state['counter'] +=1
    return state
```

`state['mumber'].append(random.randint(0,10))` :

- Generate random number from 0 to 10 and append to the `number` list

- **We have to decide where to stop**
    - For this, we need a conditional node

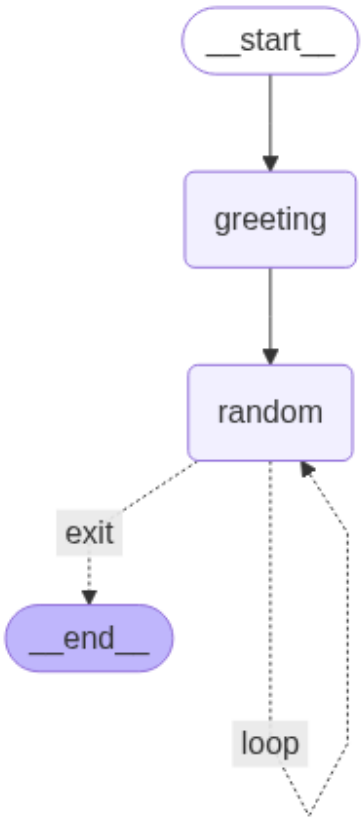
```
def should_continue(state: AgentState)→ AgentState:
    """Function to decide what to do next"""
    if state["counter"]< 5:
            print("ENTERING LOOP", state["counter"])
    return "loop"# Continue loopingelse:
    return "exit"# Exit the loop
```

Initialize graph and add nodes:

```
graph = StateGraph(AgentState)

graph.set_entry_point("greeting")

graph.add_node("greeting", greeting_node)
graph.add_node("random", random_node)


graph.add_edge("greeting", "random")

graph.add_conditional_edges(
    "random",
```

```
    should_continue,
    {"loop":"random",
     "exit": END}
)
app = graph.compile()
```

```
from IPython.display import display, Image
display(Image(app.get_graph().draw_mermaid_png()))
```



Invoke:

```
app.invoke({'name': 'john', 'number': [], 'counter': -1})
```

```
ENTERING LOOP 1
ENTERING LOOP 2
ENTERING LOOP 3
ENTERING LOOP 4

{'name': 'hi there, john', 'number': [0, 3, 5, 10, 5], 'counter': 5}
```

Here, counter is optional. No matter what value you set, it will generate 5 numbers.

!!If you only want to pass the name (& not number), write `state['number'] = []` in the `greeting_node`

Exercise

Your task:

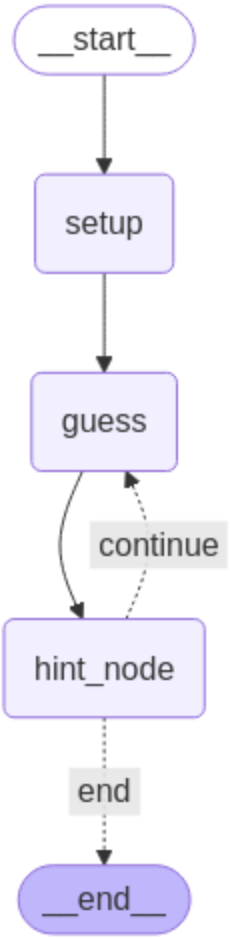
Make the graph on the right! You need to implement an Automatic Higher or Lower Game.

Set the bounds to between 1 to 20. The Graph has to keep guessing (max number of guesses is 7) where if the guess is correct, then it stops, but if not we keep looping until we hit the max limit of 7.

Each time a number is guessed, the hint node should say higher or lower and the graph should account for this information and guess the next guess accordingly.

Input: {"player_name": "Student", "guesses": [], "attempts": 0, "lower_bound": 1, "upper_bound": 20}

Hint: It will need to adjust its bounds after every guess based on the hint provided by the hint node.




```
from langgraph.graph import StateGraph, END
import random
from typing import TypedDict, List, Dict

class GameState(TypedDict):
    player_name: str
    target_number: int
    guesses: List[int]
    attempts: int
    hint: str
    lower_bound: int
    upper_bound: int
```

```
def setup_node(state: GameState) -> GameState:
    """Initialize the game with a random target number"""

    state["player_name"] = f"Welcome, {state['player_name']}!"
    state["target_number"] = random.randint(1, 20)
    state["guesses"] = []
    state["attempts"] = 0
    state["hint"] = "Game started! Try to guess the number."
    state["lower_bound"] = 1
    state["upper_bound"] = 20
    print(f"{state['player_name']} The game has begun. I'm thinking of a numbe
r between 1 and 20.")
    return state
```

`possible_guesses = [...]`

```
[i for i in range(state["lower_bound"], state["upper_bound"] + 1) if i not in stat
e["guesses"]]
```

- This creates a list of integers **from lower_bound to upper_bound** (inclusive).

```
for i in range(1, 21):
    if i not in state["guesses"]:
        return i
```

1. `if i not in state["guesses"]`

- This **filters out** any number already guessed.
- `state["guesses"]` is expected to be a list (e.g., `[1, 3]`).
- This creates a list of numbers within the **current search range** (`lower_bound` to `upper_bound`, inclusive).
- It filters out **already guessed numbers**, using `state["guesses"]`.

> ✅ Purpose: Don’t guess the same number again.

2. `if possible_guesses:`

```
guess = random.choice(possible_guesses)
```

- If there are still unguessed numbers in the range, pick one **randomly**.

> ✅ Makes the guess valid and unique.

3. `else:`

```
guess = random.randint(state["lower_bound"], state["upper_bound"])
```

- This is a **fallback**: if **all numbers have been guessed**, just pick **any** number in range, even if it's a duplicate.

> 🔒 Defensive logic to avoid errors if guesses exhausted all options.

4. `state["guesses"].append(guess)`

- Add the new guess to the list of past guesses.

5. `state["attempts"] += 1`

- Increment the number of attempts.

6. Print Statement

```
print(f"Attempt {state['attempts']}: Guessing {guess} (Current range: {state['lower_bound']}-{state['upper_bound']})")
```

- Logs the attempt number, the guessed number, and the current guessing range.
- Helps in debugging or following the game visually.

a) setup_node

- Picks a random target number (`random.randint(1, 20)`).
- Resets guesses and attempts.
- Sets the initial bounds (`1` to `20`).
- Prints a welcome message.

Key thing: This is the starting point — it initializes the game.

```
def hint_node(state: GameState)→ GameState:
    """Here we provide a hint based on the last guess and update the bound
s"""
    latest_guess= state["guesses"][-1]
    target= state["target_number"]

    if latest_guess< target:
        state["hint"]= f"The number {latest_guess} is too low. Try higher!"

        state["lower_bound"]= max(state["lower_bound"], latest_guess+ 1)
        print(f"Hint: {state['hint']}")

    elif latest_guess> target:
        state["hint"]= f"The number {latest_guess} is too high. Try lower!"

        state["upper_bound"]= min(state["upper_bound"], latest_guess- 1)
        print(f"Hint: {state['hint']}")
    else:
        state["hint"]= f"Correct! You found the number {target} in {state['attem
pts']} attempts."
        print(f"Success! {state['hint']}")

    return state
```

```
def guess_node(state: GameState) → GameState:
    """Generate a smarter guess based on previous hints"""

    possible_guesses = [i for i in range(state["lower_bound"], state["upper_bou
nd"] + 1) if i not in state["guesses"]]

    if possible_guesses:
        guess = random.choice(possible_guesses)
    else:

        guess = random.randint(state["lower_bound"], state["upper_bound"])

    state["guesses"].append(guess)
    state["attempts"] += 1
    print(f"Attempt {state['attempts']}: Guessing {guess} (Current range: {state
['lower_bound']}-{state['upper_bound']})")
    return state
```

b) guess_node

- Picks a **random guess** from numbers within the current range (`lower_bound` – `upper_bound`) **that have not been guessed yet**.
- If no new numbers are left, it guesses randomly within bounds (fallback).
- Adds the guess to the `guesses` list.
- Increments `attempts`.

Purpose: Simulates the AI guessing intelligently based on hints.

```
def hint_node(state: GameState) → GameState:
    """Here we provide a hint based on the last guess and update the bound
s"""
    latest_guess = state["guesses"][-1]
    target = state["target_number"]
```

```
    if latest_guess < target:
        state["hint"] = f"The number {latest_guess} is too low. Try higher!"

        state["lower_bound"] = max(state["lower_bound"], latest_guess + 1)
        print(f"Hint: {state['hint']}")

    elif latest_guess > target:
        state["hint"] = f"The number {latest_guess} is too high. Try lower!"

        state["upper_bound"] = min(state["upper_bound"], latest_guess - 1)
        print(f"Hint: {state['hint']}")
    else:
        state["hint"] = f"Correct! You found the number {target} in {state['attempt
s']} attempts."
        print(f"Success! {state['hint']}")

    return state
```

c) hint_node

- Looks at the latest guess and compares it to the target number.
- If guess is **too low** → updates `lower_bound` so future guesses are higher.
- If guess is **too high** → updates `upper_bound` so future guesses are lower.
- If guess is **correct** → sets success message.

Purpose: Gives feedback and narrows the guessing range.

```
def should_continue(state: GameState) → str:
    """Determine if we should continue guessing or end the game"""

    # There are 2 end conditions - either 7 is reached or the correct number is
guessed

    latest_guess = state["guesses"][-1]
```

```
    if latest_guess == state["target_number"]:
        print(f"GAME OVER: Number found!")
        return "end"
    elif state["attempts"] >= 7:
        print(f"GAME OVER: Maximum attempts reached! The number was {state
['target_number']}")
        return "end"
    else:
        print(f"CONTINUING: {state['attempts']}/7 attempts used")
        return "continue"
```

d) should_continue (Conditional Function)

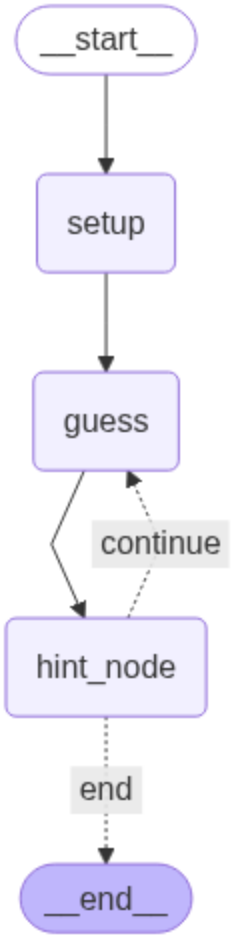
- Decides **whether to loop or end**.
- Ends if:
  1. Guess is correct, or
  2. 7 attempts are used.
- Otherwise, returns `"continue"` to guess again.

Purpose: Controls the looping in the graph.

Graph Structure

```
setup → guess → hint_node → should_continue:
   - "continue" → guess
   - "end" → END
```

```
from IPython.display import Image, display
display(Image(app.get_graph().draw_mermaid_png()))
```



Invoke:

```
result = app.invoke({"player_name": "Student", "guesses": [], "attempts": 0, "lower_bound": 1, "upper_bound": 20})
```

```
Welcome, Student! The game has begun. I'm thinking of a number between 1 and 20.
Attempt 1: Guessing 19 (Current range: 1-20)
Hint: The number 19 is too high. Try lower!
CONTINUING: 1/7 attempts used
Attempt 2: Guessing 11 (Current range: 1-18)
Hint: The number 11 is too high. Try lower!
CONTINUING: 2/7 attempts used
Attempt 3: Guessing 6 (Current range: 1-10)
Hint: The number 6 is too low. Try higher!
CONTINUING: 3/7 attempts used
Attempt 4: Guessing 9 (Current range: 7-10)
Hint: The number 9 is too high. Try lower!
CONTINUING: 4/7 attempts used
Attempt 5: Guessing 7 (Current range: 7-8)
Hint: The number 7 is too low. Try higher!
CONTINUING: 5/7 attempts used
Attempt 6: Guessing 8 (Current range: 8-8)
Success! Correct! You found the number 8 in 6 attempts.
GAME OVER: Number found!
```

💡 You can pass only **name**

```
result = app.invoke({"player_name": "Cody"})
```

```
Welcome, Cody! The game has begun. I'm thinking of a number between 1 and 20.
Attempt 1: Guessing 14 (Current range: 1-20)
Hint: The number 14 is too high. Try lower!
CONTINUING: 1/7 attempts used
Attempt 2: Guessing 4 (Current range: 1-13)
Hint: The number 4 is too low. Try higher!
CONTINUING: 2/7 attempts used
Attempt 3: Guessing 10 (Current range: 5-13)
Hint: The number 10 is too high. Try lower!
CONTINUING: 3/7 attempts used
Attempt 4: Guessing 7 (Current range: 5-9)
Hint: The number 7 is too high. Try lower!
CONTINUING: 4/7 attempts used
Attempt 5: Guessing 5 (Current range: 5-6)
Hint: The number 5 is too low. Try higher!
CONTINUING: 5/7 attempts used
Attempt 6: Guessing 6 (Current range: 6-6)
Success! Correct! You found the number 6 in 6 attempts.
GAME OVER: Number found!
```