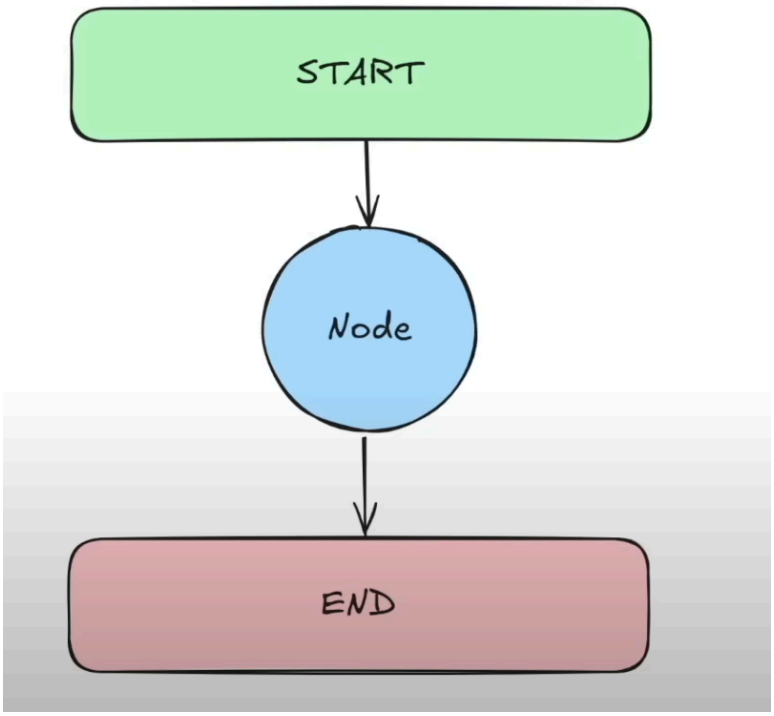
Simple Graphs

```
pip install langgraph
```

Steps:

1. Create AgentState **Class** (`AgentState(TypedDict)`)
    - It has the keys in the state
2. Define node (`greeting_node`)
    - It always takes and return `state`
3. Define stategraph (`StateGraph(AgentState)`)
4. Add nodes
5. Set entry and exit points
6. Compile the graph (`graph.compile()`)
7. Invoke → `app.invoke({"message": "Bob"})`




```
from typing import Dict, TypedDict
from langgraph.graph import StateGraph # framework that helps you design a
nd manage the flow of tasks in your application using a graph structure
```

Crate agent state

- The **input** & **output** both should be **state**

```
 #We now create an AgentState - shared data structure that keeps track of inf
ormation as your application runs.

class AgentState(TypedDict): # Our state schema
    message : str

def greeting_node(state: AgentState) → AgentState:
    """simple node that adds a greetig method to the satte"""
```

```
    state['message'] = "Hey " + state["message"] + ", how is your day going?"
    return state
```

- It takes the `state` (a dictionary with a `message` key) and modifies the message by **adding a greeting**.
- For example, if the input state is:

  ```
  {"message": "Alice"}
  ```

  Then the function would update it to:

  ```
  {"message": "Hey Alice, how is your day going?"}
  ```

- Finally, it **returns the modified state**.

Create StateGraph

```
graph = StateGraph(AgentState)

graph.add_node("greeter", greeting_node)

graph.set_entry_point("greeter")
graph.set_finish_point("greeter")

app = graph.compile()
```

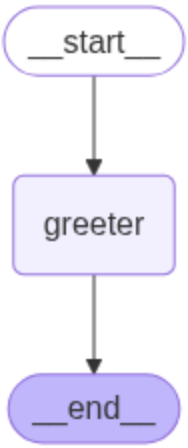
- You pass state (***AgentState***)
- Add node → `graph.add_node()` **takes 2 parameters:**
  1. Name of node
  2. Action it performs
- Create start & finish point

- `graph.set_entry_point("greeter")`
  `graph.set_finish_point("greeter")`

- In this case, start and end node both are connecting to `greeter` because it's a simple graph
- Compile the graph → `app = graph.compile()`

Visualize:

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



Invoke:

```
app.invoke({"message": "Bob"})
```

```
{'message': 'Hey Bob, how is your day going?'}
```

```
result = app.invoke({"message": "Bob"})
result['message']
```

```
'Hey Bob, how is your day going?'
```

Graph with Multiple Inputs:

- More complex
- List data

```
from typing import TypedDict, List
from langgraph.graph import StateGraph
```

- The `List` is used for type annotations that specify **a list of elements of a particular type**.
    - For example, if you use `List[int]`, it means a list that holds integers. Similarly, `List[str]` is a list of strings.

```
class AgentState(TypedDict):
    values : List[int]
    name : str
    result : str
```

Build Nodes:

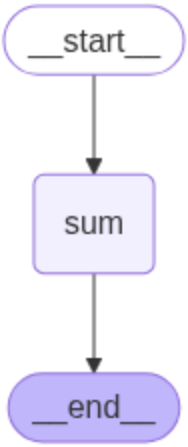
```
def process_values(state: AgentState) -> AgentState:
    """This function handles multiple different inputs"""

    state['result'] = f"Hi there {state['name']}! Your sum = {sum(state['value
s'])}"
    return state
```

Create Graph:

```
graph = StateGraph(AgentState)

graph.add_node("sum", process_values)

graph.set_entry_point("sum")
graph.set_finish_point("sum")

app= graph.compile()
```

Visualize:

```
from IPython.display import Image, display.

display(Image(app.get_graph().draw_mermaid_png()))
```



Invoke:

```
result = app.invoke({"values": [1,2,3,4,5], "name": "Steve"})
result['result']
```

```
'Hi there Steve! Your sum = 15'
```

```
result
✓ 0.0s
{'values': [1, 2, 3, 4, 5],
 'name': 'Steve',
 'result': 'Hi there Steve! Your sum = 15'}
```

💡 **We did not pass result. LangGraph assigns a null value to result.**

Task:

🏗️ Your task:

Create a **Graph** where you pass in a single list of integers along with a name and an operation. If the operation is a "+", you **add** the elements and if it is a "*", you **multiply** the elements, **all within the same node.**

Input: {"name": "Jack Sparrow","values": [1,2,3,4] , "operation": "*"}

Output: "Hi Jack Sparrow, your answer is: 24"

```
import math

class AgentState(TypedDict):
    name : str
    values: List[int]
    operation : str
    result : str
```

```
def operation(state):
    if state['operation'] == "*":
        state["result"] = f"Hey {state['name']}, your multiplication is {math.prod(s
tate['values'])}"
    else:
        state["result"] = f"Hey {state['name']}, your sum is {sum(state['value
s'])}"
    return state
```

```
graph = StateGraph(AgentState)

graph.add_node("operation", operation)

graph.set_entry_point("operation")
graph.set_entry_point("operation")

app= graph.compile()

app.invoke({"name": "John", "values": [1,2,3,4], "operation": "*"})
```

```
{'name': 'John',
 'values': [1, 2, 3, 4],
 'operation': '*',
 'result': 'Hey John, your multiplication is 24'}
```

```
app.invoke({"name": "John", "values": [1,2,3,4], "operation": "+"})
✓ 0.0s
{'name': 'John',
 'values': [1, 2, 3, 4],
 'operation': '+',
 'result': 'Hey John, your sum is 10'}
```

Sequential Graph

- **Multiple nodes**

```
from typing import TypedDict # Imports all the data types we need
from langgraph.graph import StateGraph
```

```
class AgentState(TypedDict):
    name : str
    age : str
    final : str

def first_node(state:AgentState) -> AgentState:
    """This is the first node of our sequence"""

    state["final"] = f"Hi {state['name']}!"
    return state

def second_node(state:AgentState) -> AgentState:
    """This is the second node of our sequence"""

    state["final"] = state["final"] + f" You are {state['age']} years old!"

    return state
```

- We concatenated both the states because the `second_node` will replace all the content in `final`

Build Graph:

- Join the nodes → `EDGE` → `graph.add_edge(start, end)`

```
graph = StateGraph(AgentState)

graph.add_node("first_node", first_node)
```

```
graph.add_node("second_node", second_node)

graph.set_entry_point("first_node")
graph.set_finish_point("second_node")

graph.add_edge("first_node", "second_node")

app = graph.compile()
```

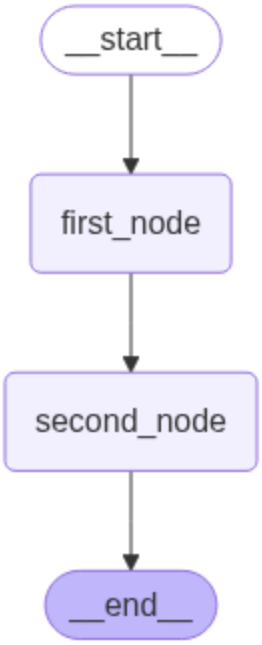
Invoke:

```
app.invoke({"name": "Phil", "age": "47"})
```

```
{'name': 'Phil', 'age': '47', 'final': 'Hi Phil! You are 47 years old!'}
```

Visualize:

```
from IPython.display import display, Image

display(Image(app.get_graph().draw_mermaid_png()))
```



Exercise:

Your task:

1. Accept a user's name, age, and a list of their skills.
2. Pass the state through **three nodes** that:
   - **First node:** Personalizes the name field with a greeting.
   - **Second node:** Describes the user's age.
   - **Third node:** Lists the user's skills in a formatted string.
3. The final output in the result field should be a **combined message** in this format:

Output: "Linda, welcome to the system! You are 31 years old! You have skills in: Python Machine Learning, and LangGraph"

```
class AgentState(TypedDict):
    name : str
    age: int
    skills : List[str]
    final :  str
```

```
def welcome(state):
    state['final'] = f"Hi {state['name']}"
    return state
def age(state):
    state['final'] =  state['final'] + f", you are {state['age']} years old."
    return state
def skills(state):
    state['final'] = state['final'] + f" Your skills are: {(state['skills'])}"
    return state

graph = StateGraph(AgentState)

graph.add_node("welcome", welcome)
graph.add_node("age", age)
graph.add_node("skills", skills)

graph.add_edge("welcome", "age")
graph.add_edge("age", "skills")

graph.set_entry_point("welcome")
graph.set_finish_point("skills")

app=graph.compile()
```

```
app.invoke({"name": "Phil", "age": "47", "skills": ["python", "ML", "Gen AI"] })
```

```
{'name': 'Phil',
 'age': '47',
 'skills': ['python', 'ML', 'Gen AI'],
 'final': "Hi Phil, you are 47 years old. Your skills are: ['python', 'ML', 'Gen AI']"}
```

- Even though you pass age as `str`, you won't get any error

Change `['python', 'ML', 'Gen AI']` to normal string:

- `', '.join(state['skills'])`

```
class AgentState(TypedDict):
    name : str
    age: int
    skills : List[str]
    final :  str

def welcome(state):
    state['final'] = f"Hi {state['name']}"
    return state
def age(state):
    state['final'] =  state['final'] + f", you are {state['age']} years old."
    return state
def skills(state):
    state['final'] = state['final'] + f" Your skills are: {', '.join((state['skills']))}"
    return state

graph = StateGraph(AgentState)

graph.add_node("welcome", welcome)
graph.add_node("age", age)
graph.add_node("skills", skills)

graph.add_edge("welcome", "age")
graph.add_edge("age", "skills")

graph.set_entry_point("welcome")
graph.set_finish_point("skills")

app=graph.compile()
```

```
app.invoke({"name": "Phil", "age": "47", "skills": ["python", "ML", "Gen AI"] })
```

```
{'name': 'Phil',
 'age': '47',
 'skills': ['python', 'ML', 'Gen AI'],
 'final': 'Hi Phil, you are 47 years old. Your skills are: python, ML, Gen AI'}
```

The expression `', '.join(state['skills'])` is used to **combine** the elements of a list (`state['skills']` in this case) into a **single string**, where each element is separated by a comma followed by a space `', '`.

- `.join()`: This is a **string method** in Python. It takes an iterable (like a list or tuple) and **joins** each element into a single string, inserting the string you call `.join()` on between each element of the iterable.
- `', '`: This is the string that will be inserted between each element of the list. In this case, it's a comma followed by a space.

```
skills = ['python', 'ML', 'Gen AI']
skills_string = ', '.join(skills)
print(skills_string)
```

```
python, ML, Gen AI
```