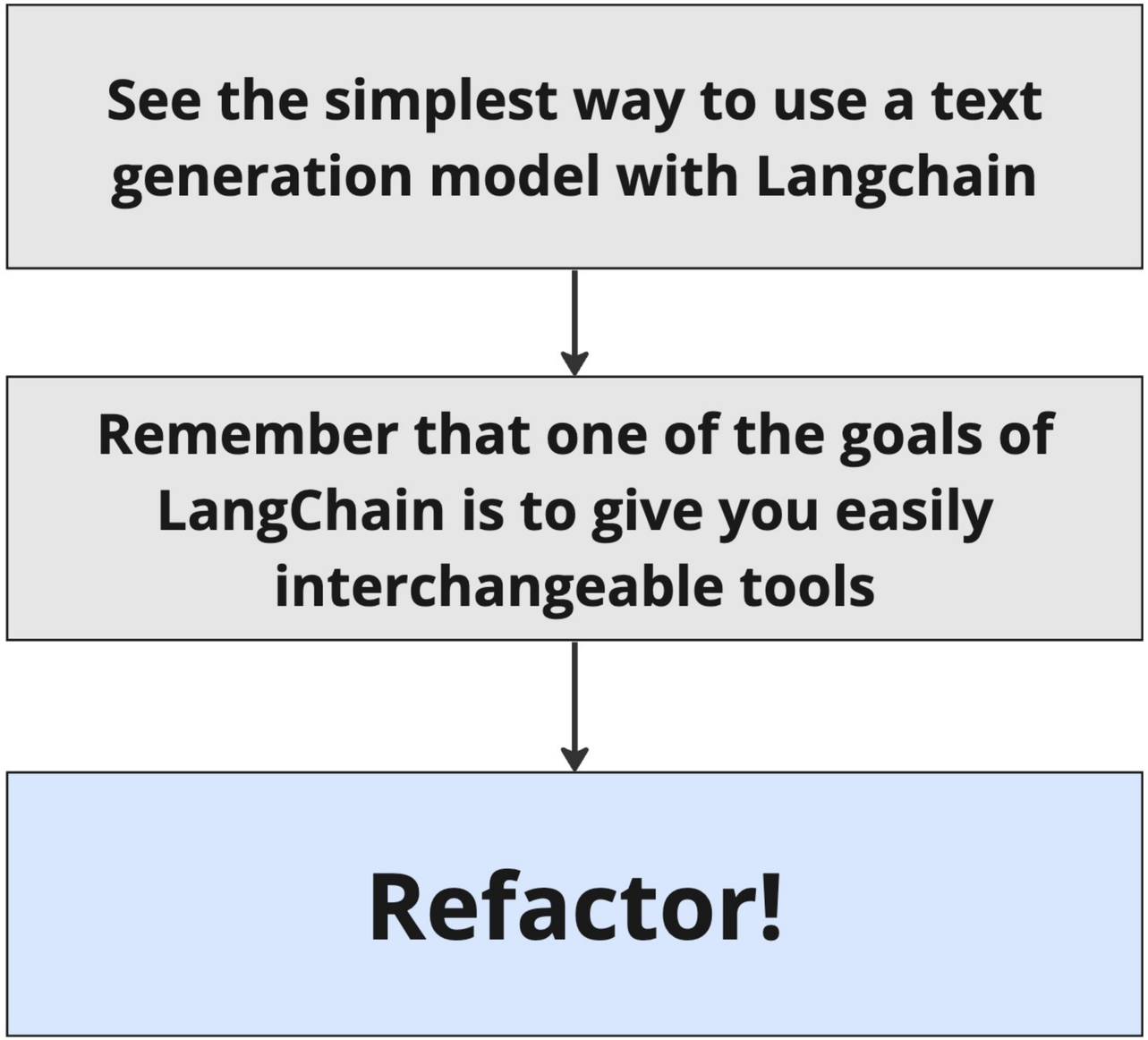
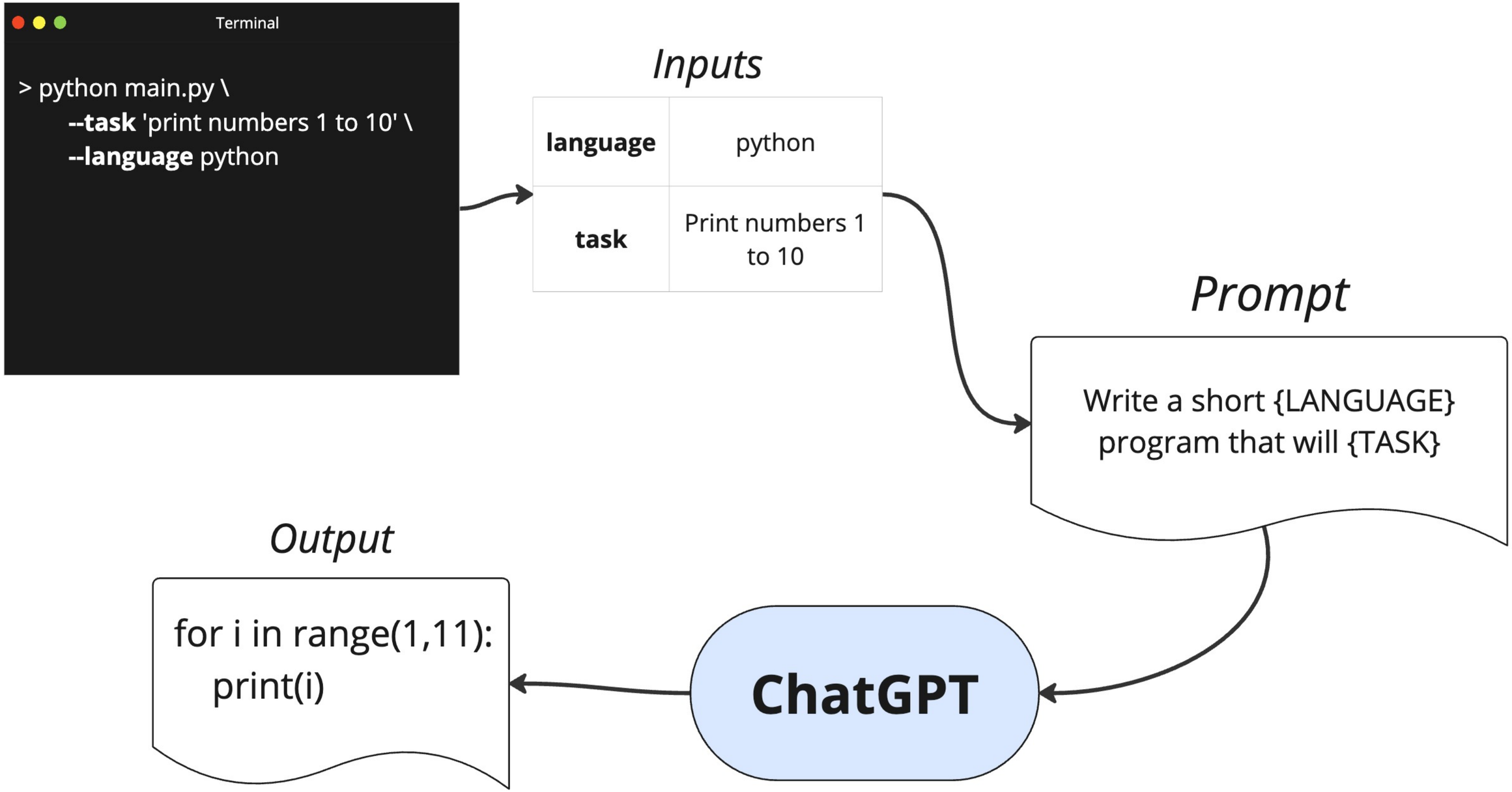
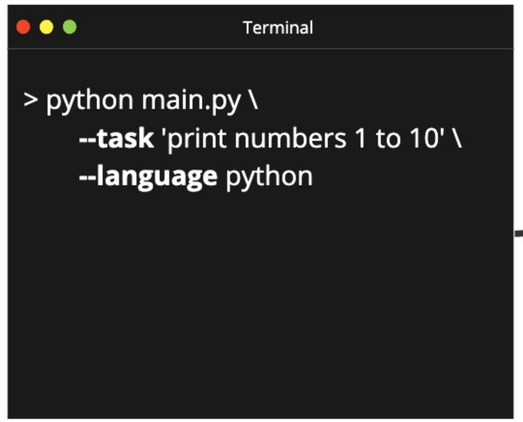

See the simplest way to use a text generation model with Langchain
Remember that one of the goals of LangChain is to give you easily interchangeable tools
Refactor!


Terminal
> python main.py \
--task 'print numbers 1 to 10' \
--language python
Inputs
language
python
task
Print numbers 1 to 10
Prompt
Write a short {LANGUAGE} program that will {TASK}
ChatGPT
Output
for i in range(1,11):
print(i)



Inputs

language	python
task	Print numbers 1 to 10

Prompt to use

ChatGPT

Output

```
for i in range(1,11):
    print(i)
```

Prompt A

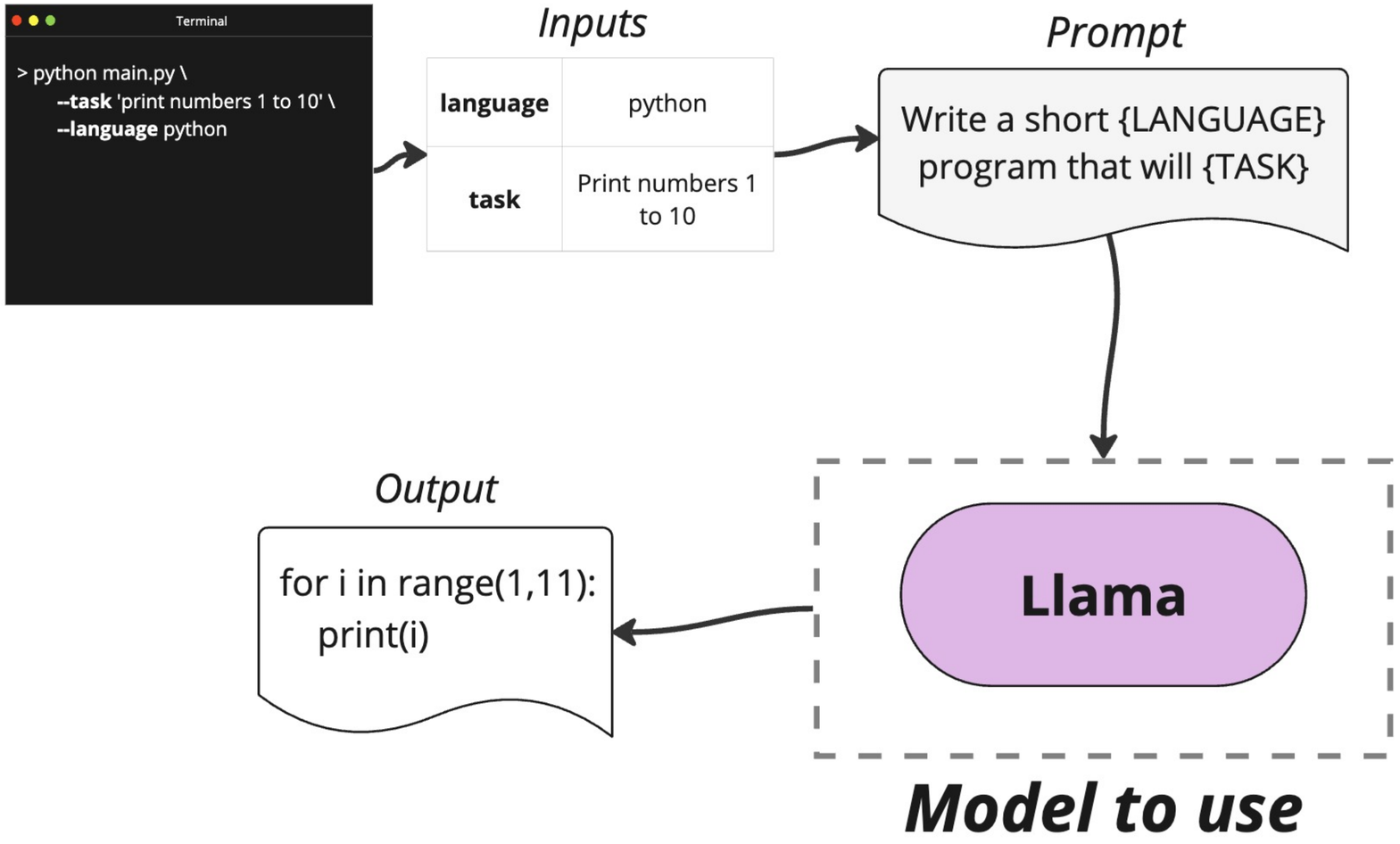
Write a short {LANGUAGE} program that will {TASK}

Prompt B

Write a short snippet of code to {TASK} using {LANGUAGE}

Prompt C

Using {LANGUAGE}, write a long program that will {TASK}

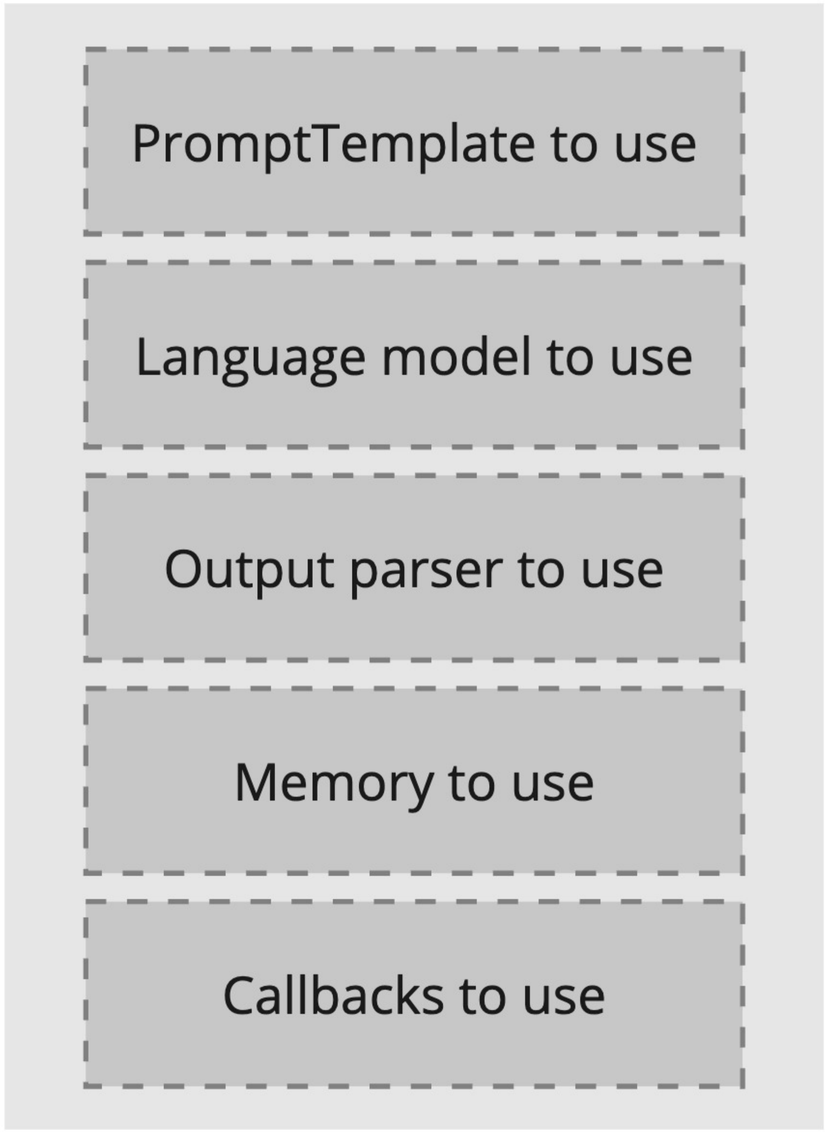


Different language models we might have access to

ChatGPT

MosaicML

Chain

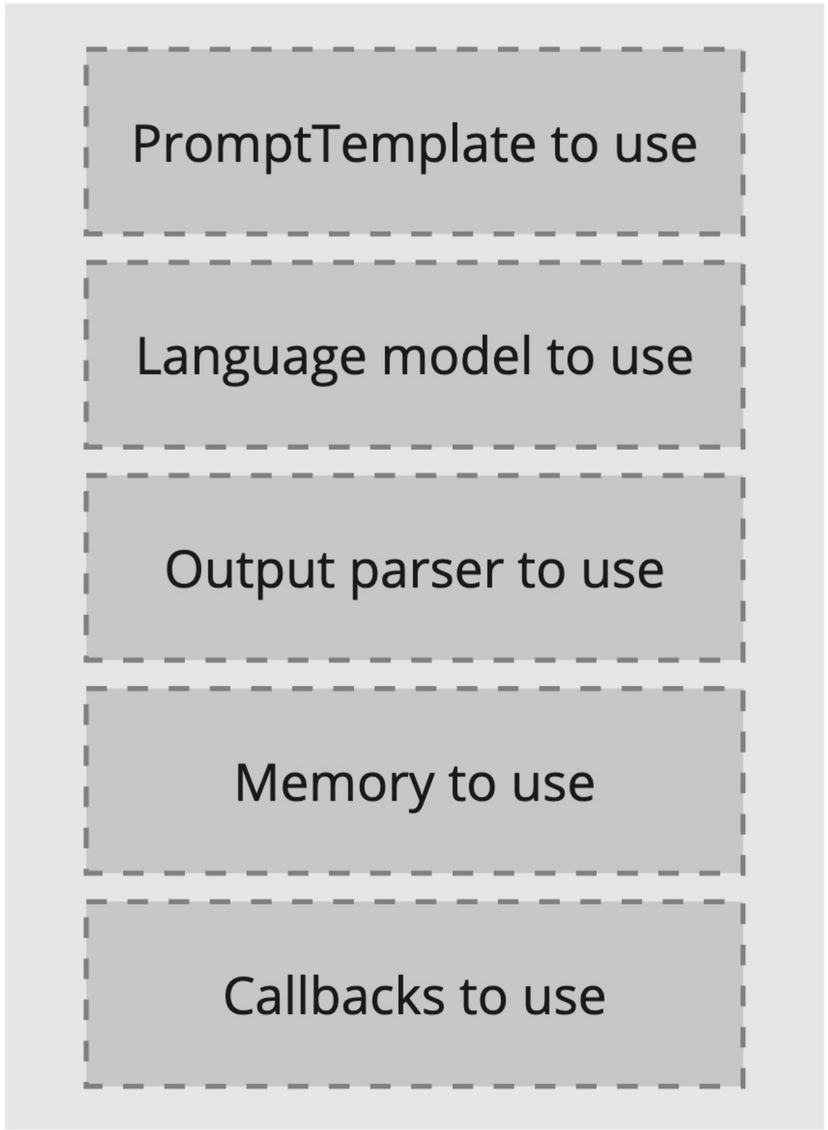


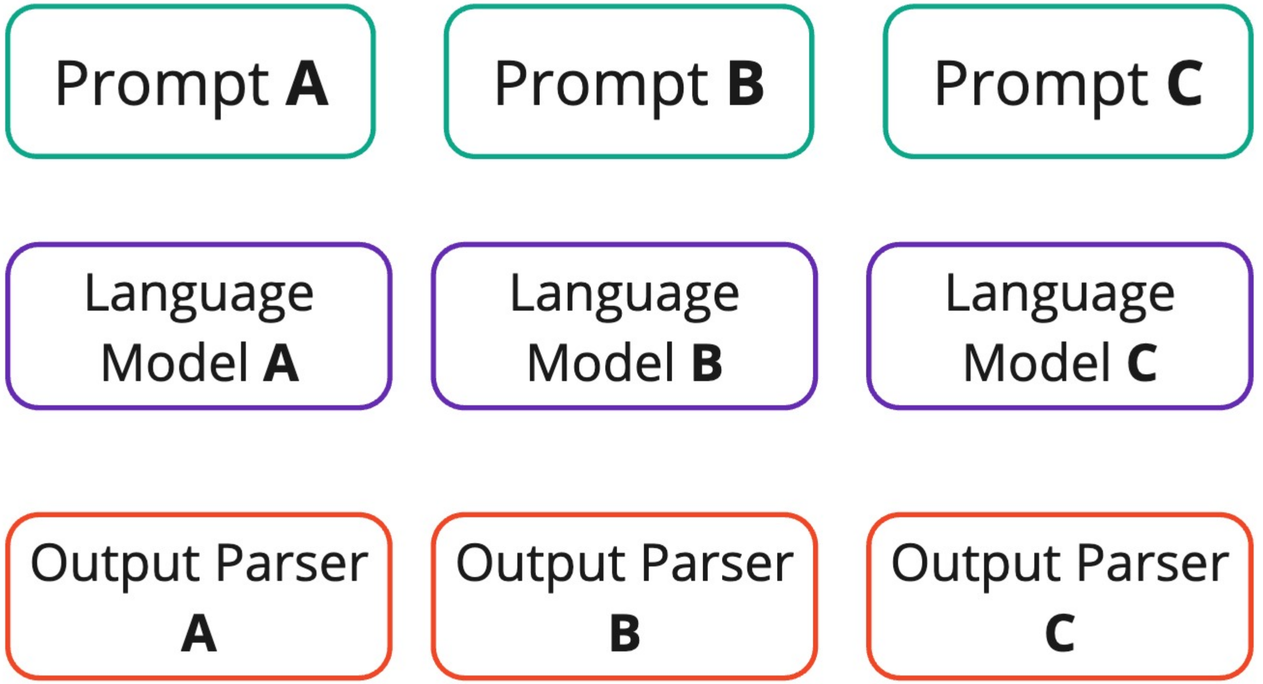
A Chain is an object that specifies the template, language model, output parser, memory, and callbacks to use for text generation

We use chains so we can easily configure different parts without changing a ton of code

Multiple chains can also be connected together to make really, really interesting applications

Chain A

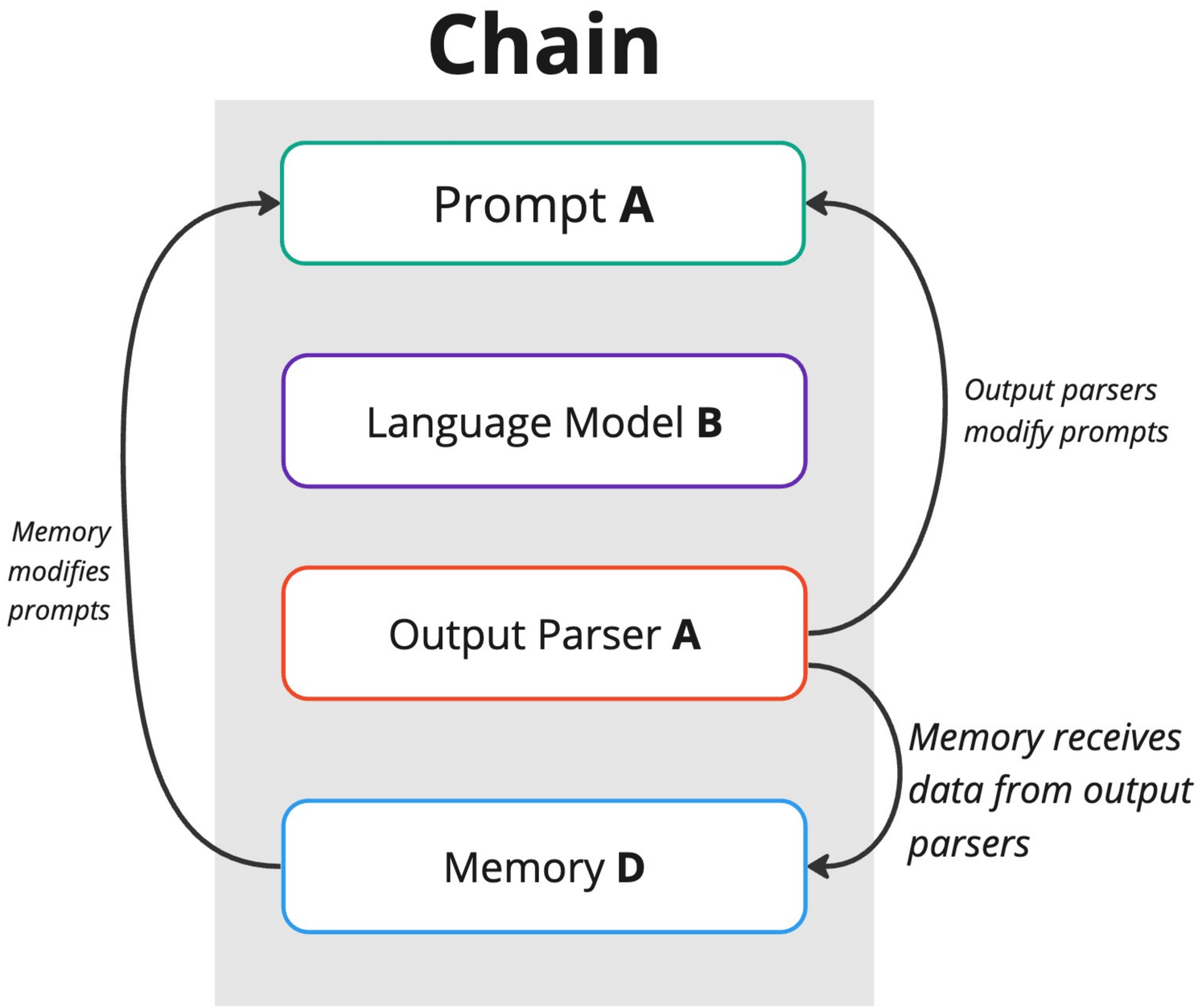

PromptTemplate to use
Language model to use
Output parser to use
Memory to use
Callbacks to use


Prompt A
Prompt B
Prompt C
Language Model A
Language Model B
Language Model C
Output Parser A
Output Parser B
Output Parser C

```python
def generate_text(language, task):
    prompt = PromptSLKDFJ()
    model = ModelCHATGPT()
    output_parser = OutputParserC()

    formatted_prompt = prompt(language, task)

    output = model(formatted_prompt)

    return output_parser(output)
```

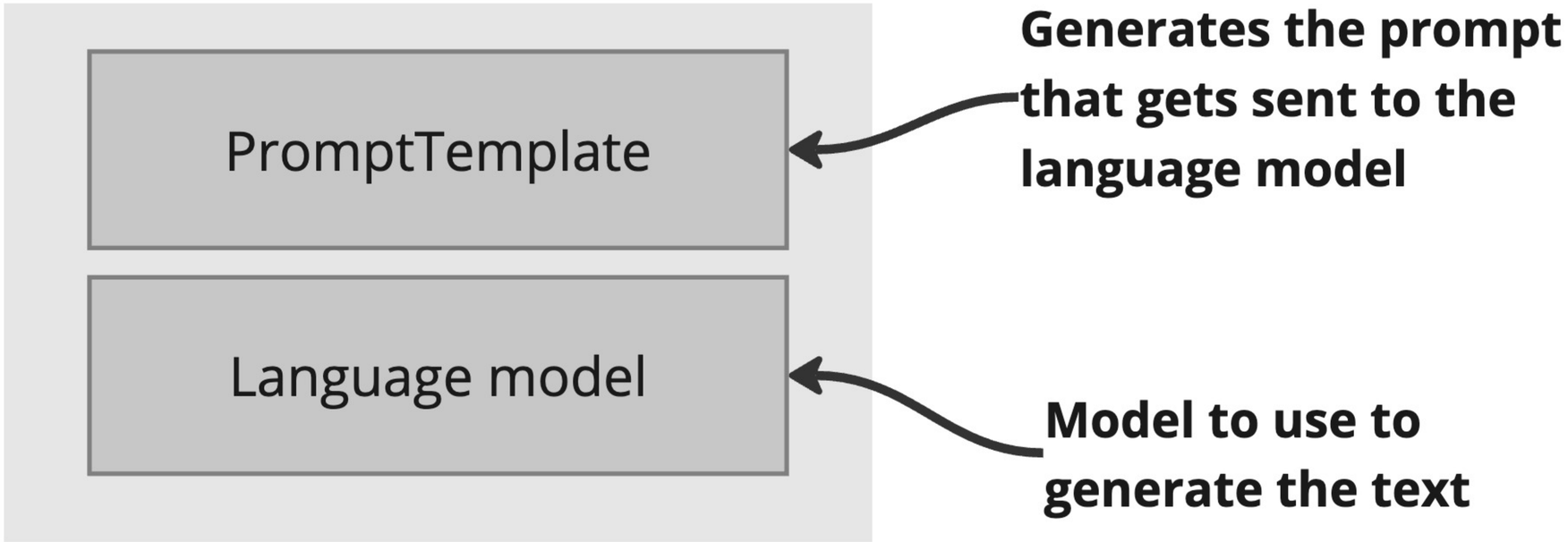
Why can't we do something like this?

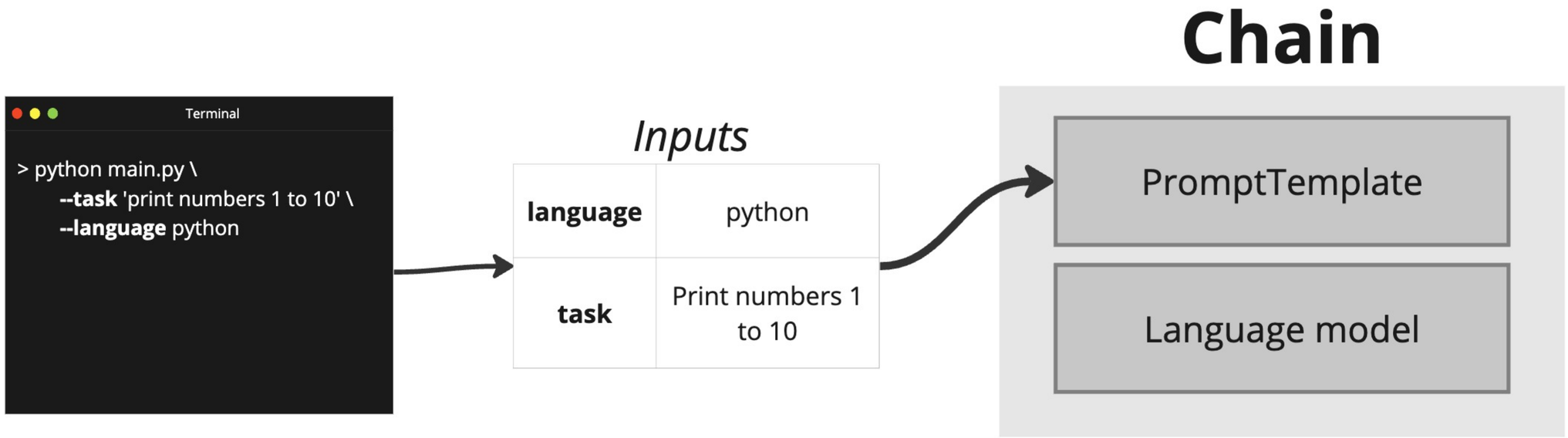
Still seems easy to swap out the prompt, model, etc, right?

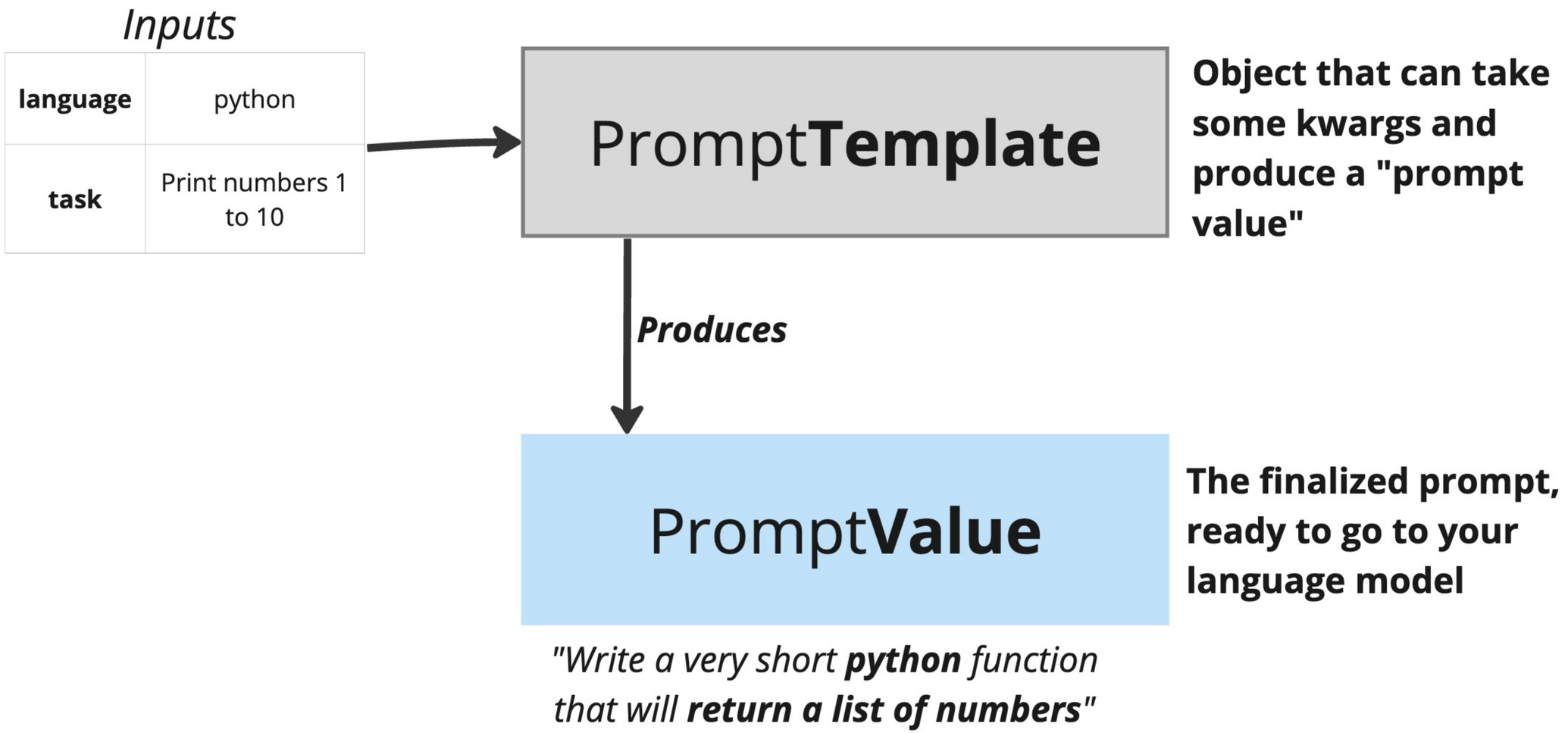


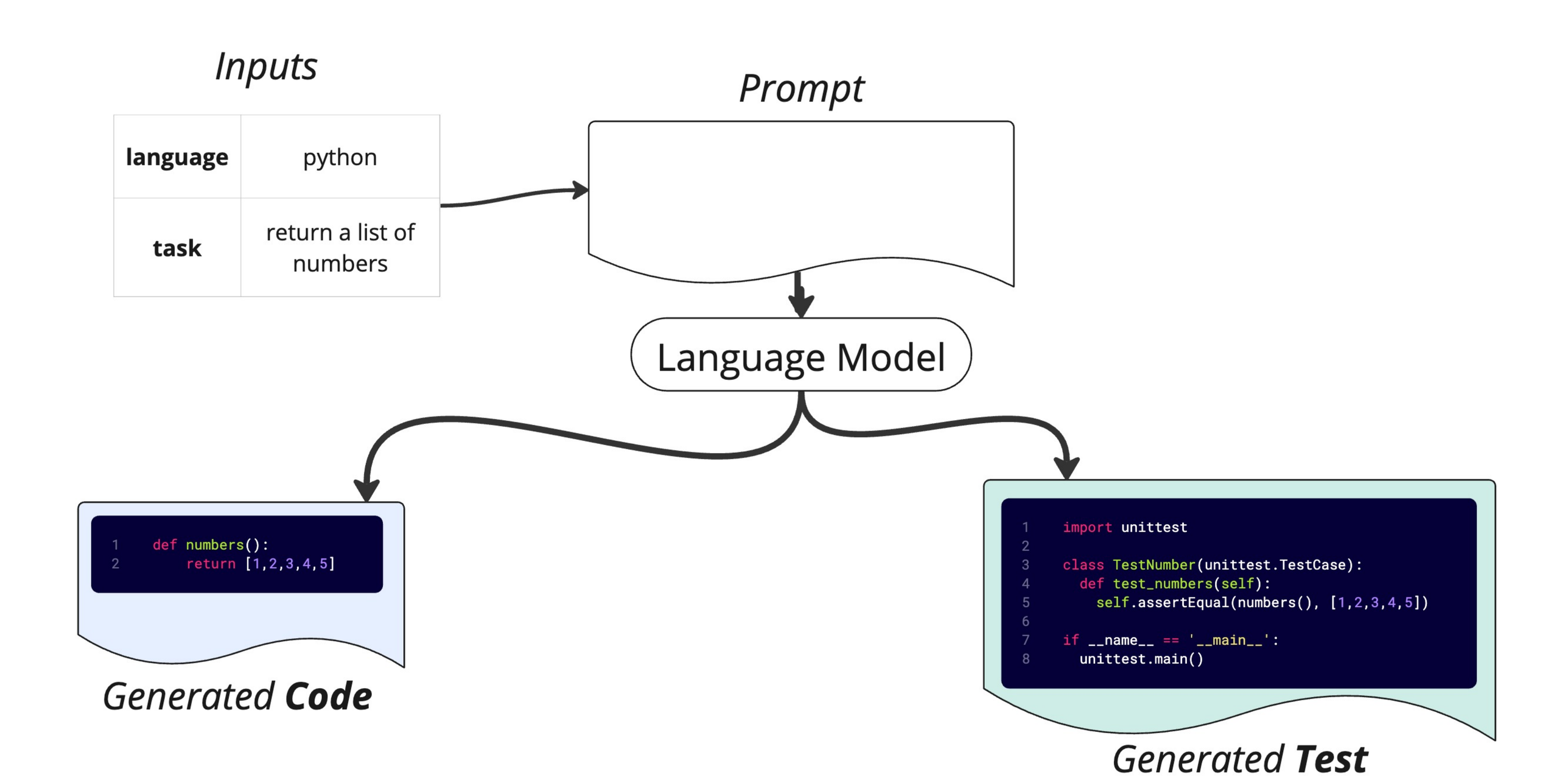
Different parts of a chain talk to other parts, sometimes in unexpected ways

Chain


PromptTemplate
Language model
Generates the prompt that gets sent to the language model
Model to use to generate the text


Terminal
> python main.py \
--task 'print numbers 1 to 10' \
--language python
Inputs
language
python
task
Print numbers 1 to 10
Chain
PromptTemplate
Language model


Inputs
language
python
task
Print numbers 1 to 10
PromptTemplate
Object that can take some kwargs and produce a "prompt value"
Produces
PromptValue
The finalized prompt, ready to go to your language model
"Write a very short python function that will return a list of numbers"


Inputs
language
python
task
return a list of numbers
Prompt
Language Model
1 def numbers():
2 return [1,2,3,4,5]
Generated Code
1 import unittest
2
3 class TestNumber(unittest.TestCase):
4 def test_numbers(self):
5 self.assertEqual(numbers(), [1,2,3,4,5])
6
7 if __name__ == '__main__':
8 unittest.main()
Generated Test

Option #1
Update the prompt template

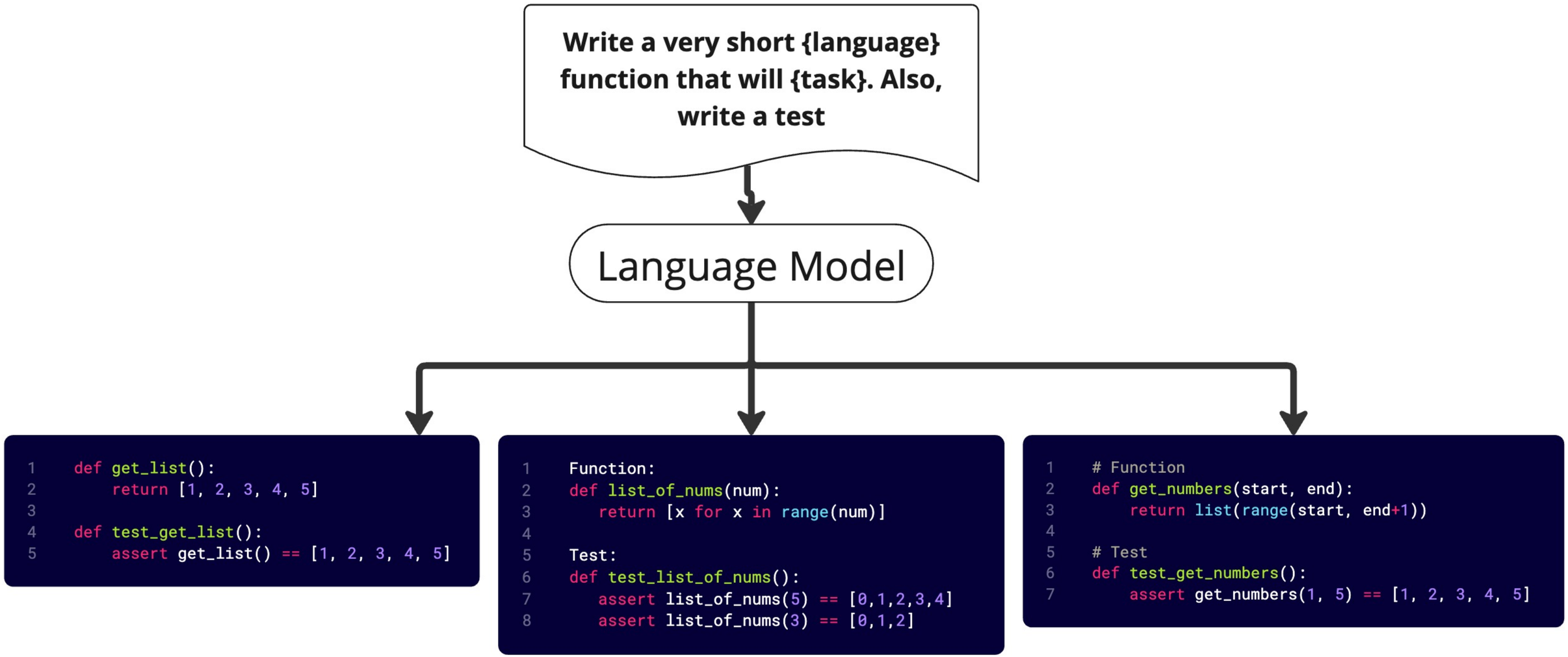
Inputs

language	python
task	return a list of numbers

Prompt

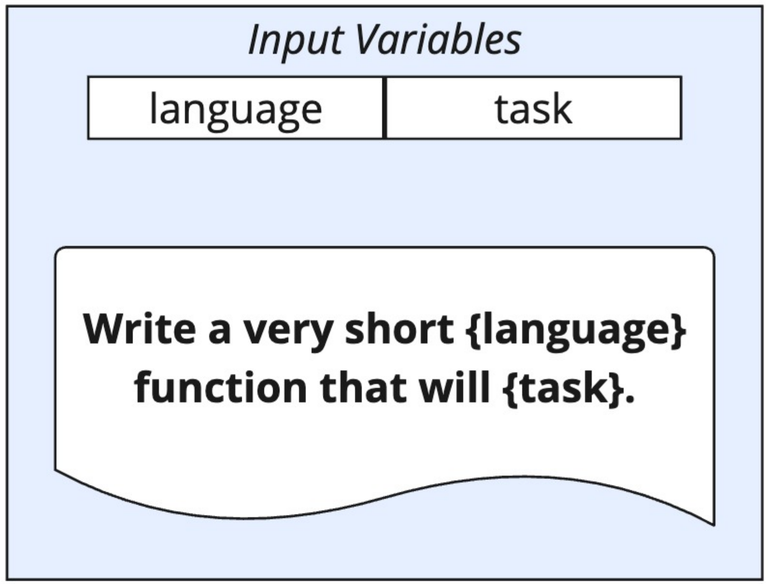
Write a very short {language} function that will {task}. Also, write a test for it.

Language Model



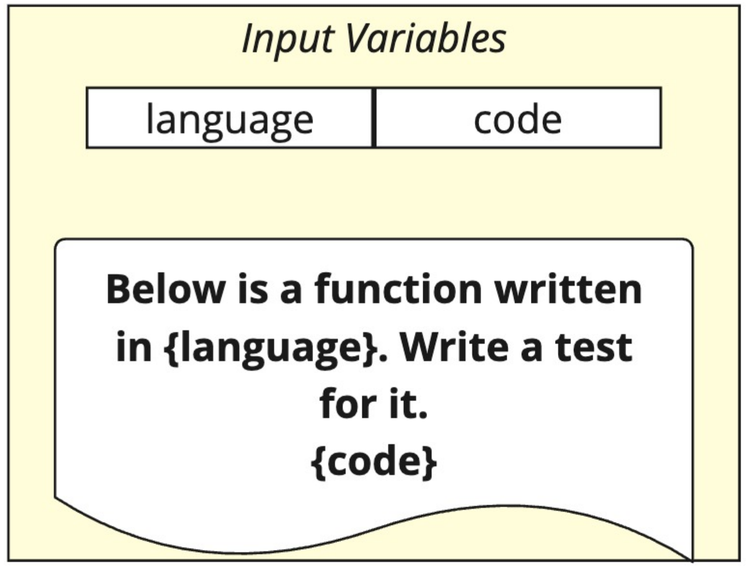
Where does the implementation end and where does the test begin?

Prompt Template



This is our existing prompt template

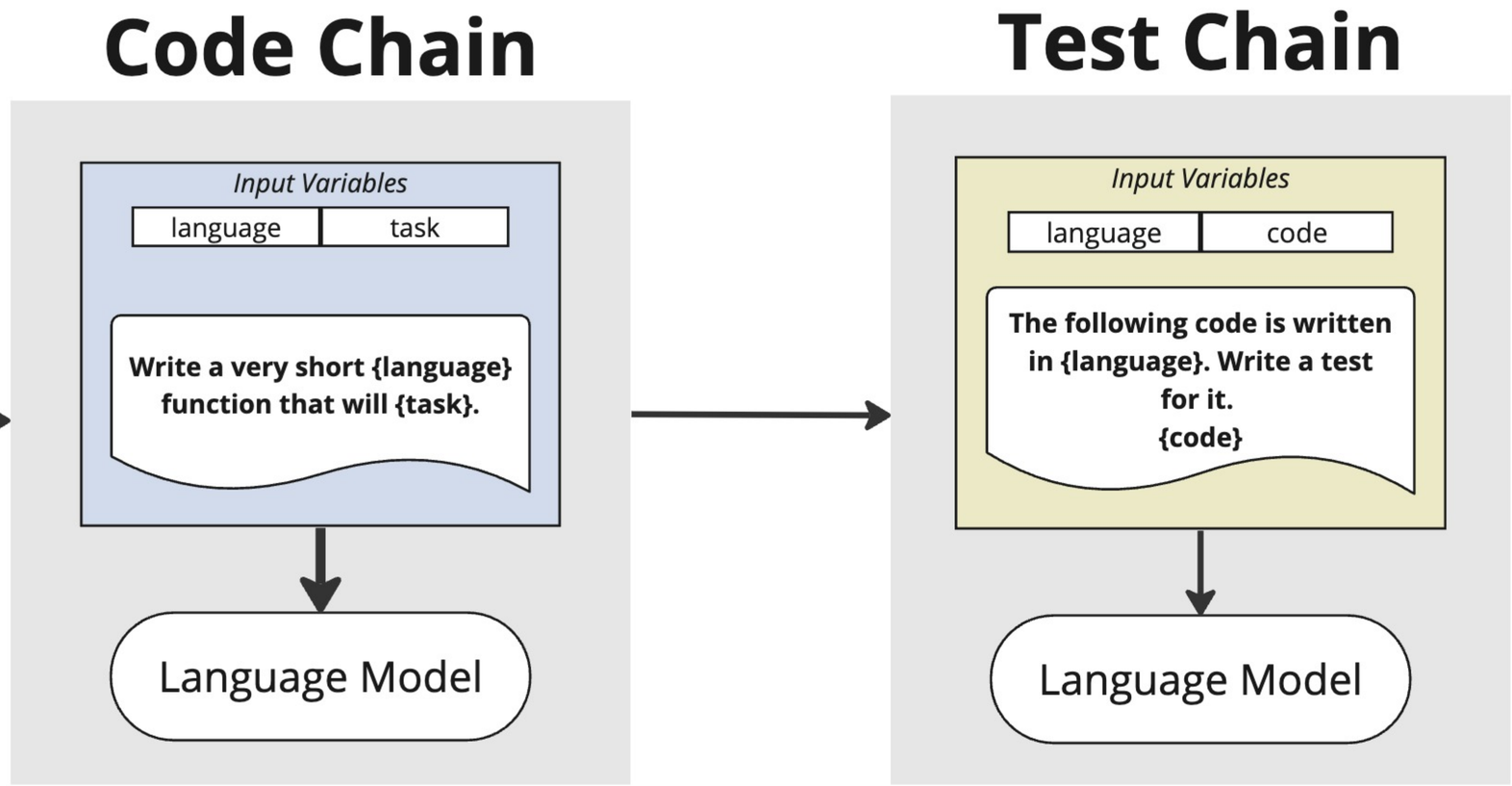
Prompt Template



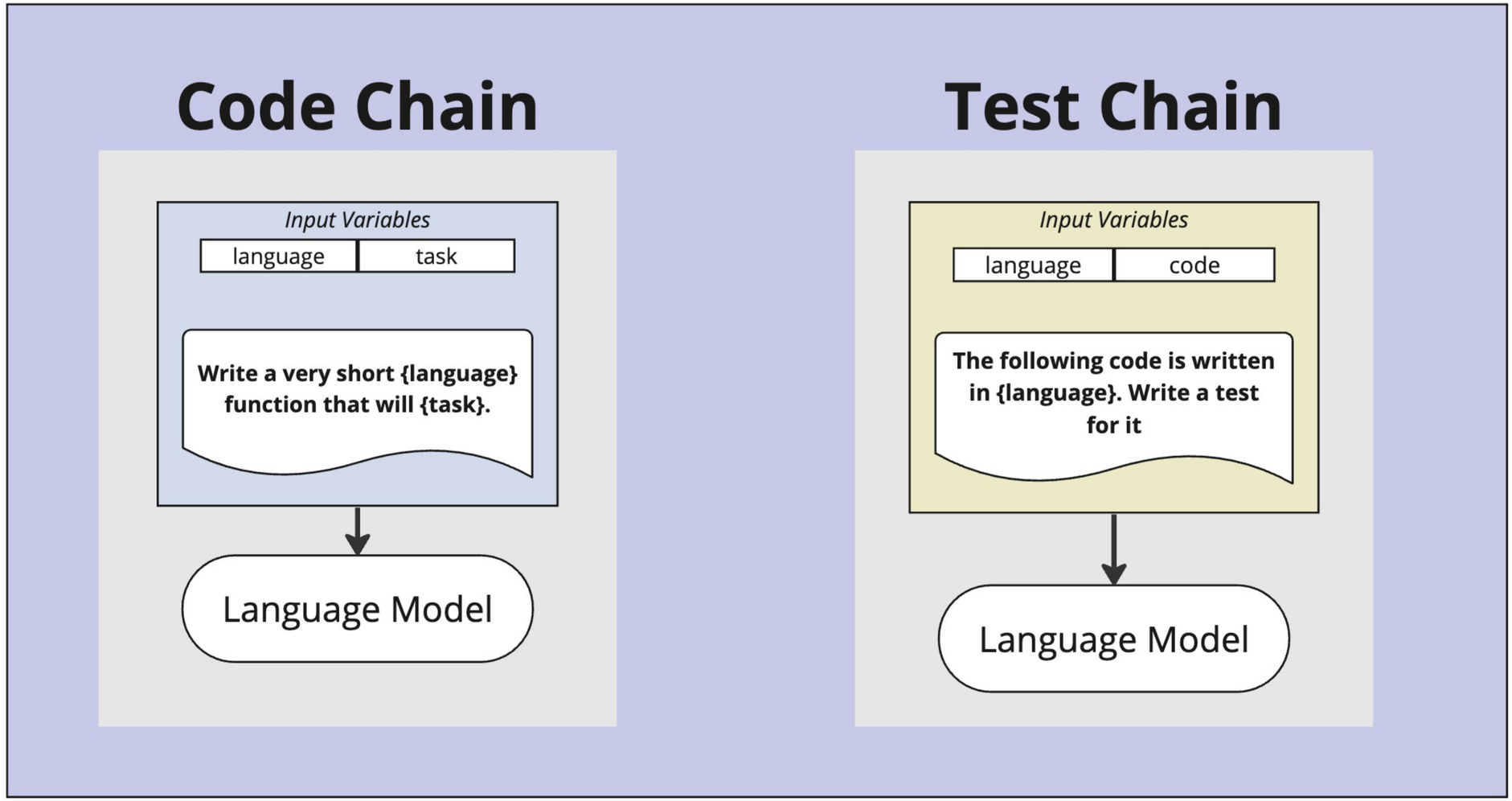
New one!

Inputs

language	python
task	return a list of numbers



SequentialChain



Ways to Run a Chain

Method	Input Type	Output Type	Example	Notes

__call__	Dictionary	Dictionary with inputs + output	`result = code_chain({ "language": args.language, "task": args.task})`
			
			`print(result) # {"language": ..., "task": ..., "text": ...}`	

run	Dictionary	String	`result = code_chain.run({ "language": args.language, "task": args.task})`
			
			`print(result) # "def numbers():\n  return [1,2,3,4]"`
				Only usable if you have one **output** variable

predict	keywords	String	`result = code_chain.predict( language=args.language, task=args.task)`
			
			`print(result) # "def numbers():\n  return [1,2,3,4]"`	

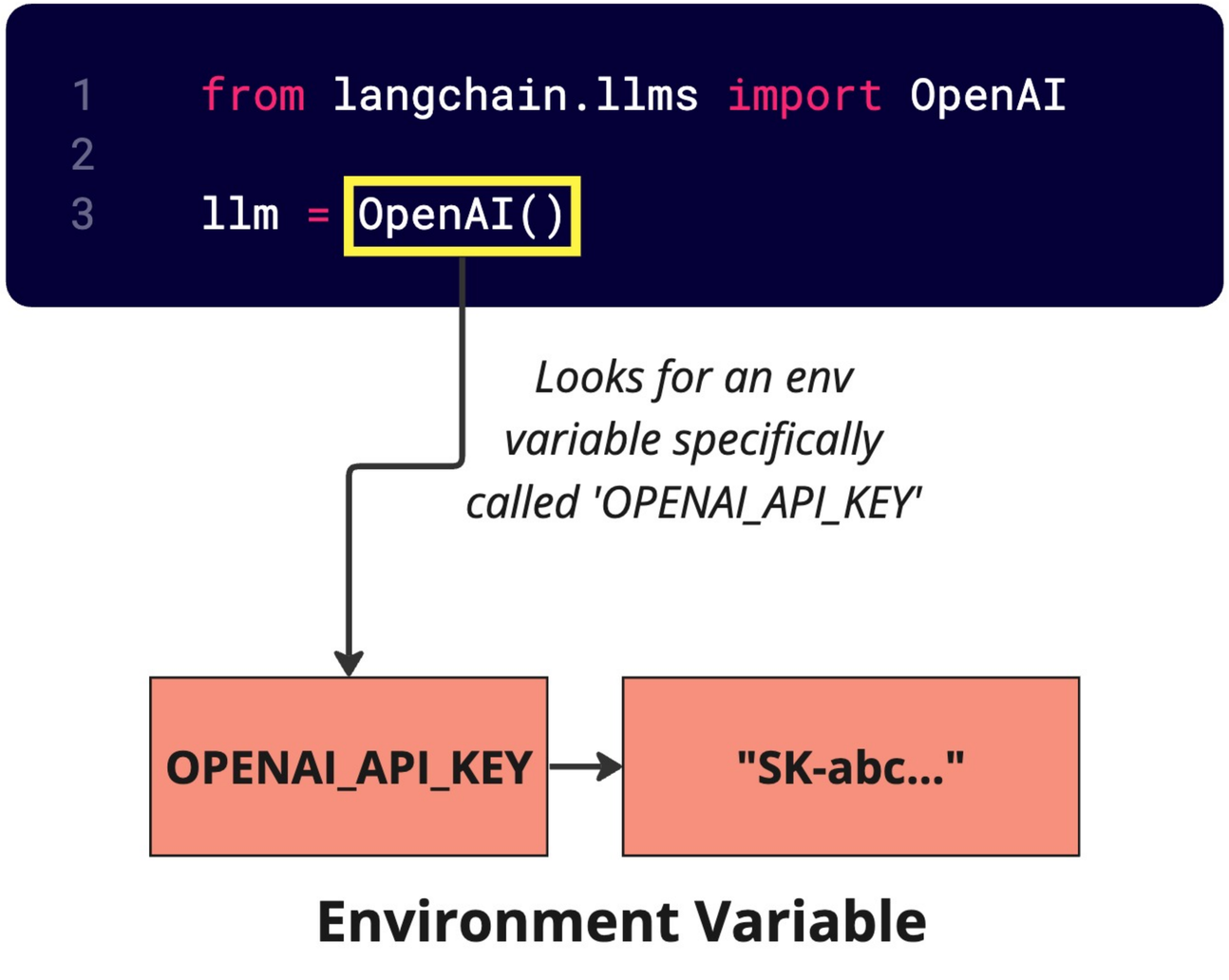
Ways to Run a Chain with

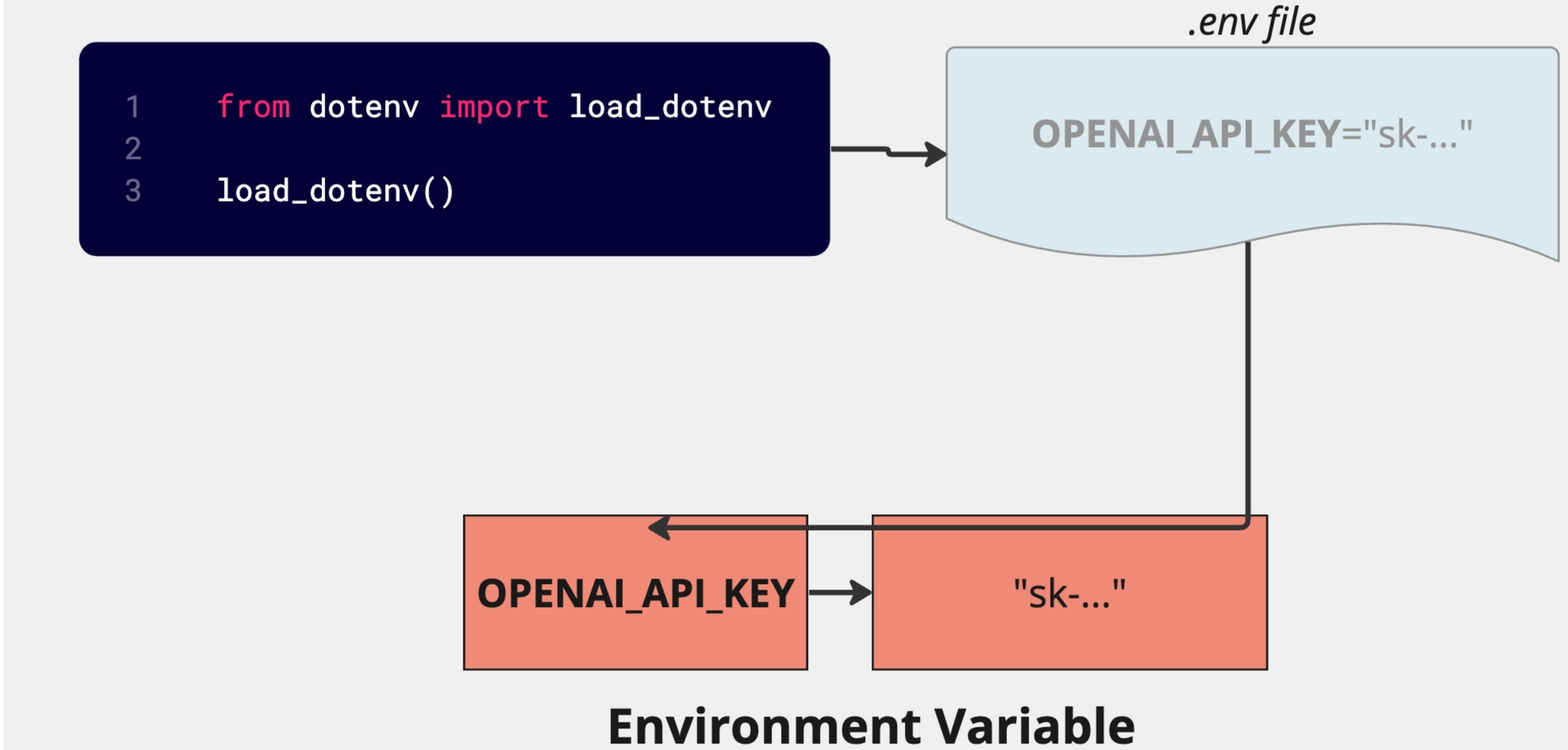
exactly 1 Input Variable

Method	Input Type	Output Type	Example	Notes

__call__	String	Dictionary with inputs + output	`result = code_chain(args.task)`
			
			`print(result) # {"language": ..., "task": ..., "text": ...}`	

run	String	String	`result = code_chain.run(args.task)`
			
			`print(result) # "def numbers():\n  return [1,2,3,4]"`
				Only usable if you have one **output** variable


1 from langchain.llms import OpenAI
2
3 llm = OpenAI()
Looks for an env variable specifically called 'OPENAI_API_KEY'
OPENAI_API_KEY
"SK-abc..."
Environment Variable


from dotenv import load_dotenv
load_dotenv()
.env file
OPENAI_API_KEY="sk-..."
OPENAI_API_KEY
"sk-..."
Environment Variable