COL1000: Introduction to Programming

Using GenAI Tools for Programming

Subodh Sharma | Lec 37 | Nov 12

भारतीय प्रौद्योगिकी संस्थान दिल्ली
INDIAN INSTITUTE OF TECHNOLOGY DELHI

GenAI Tools

The problem

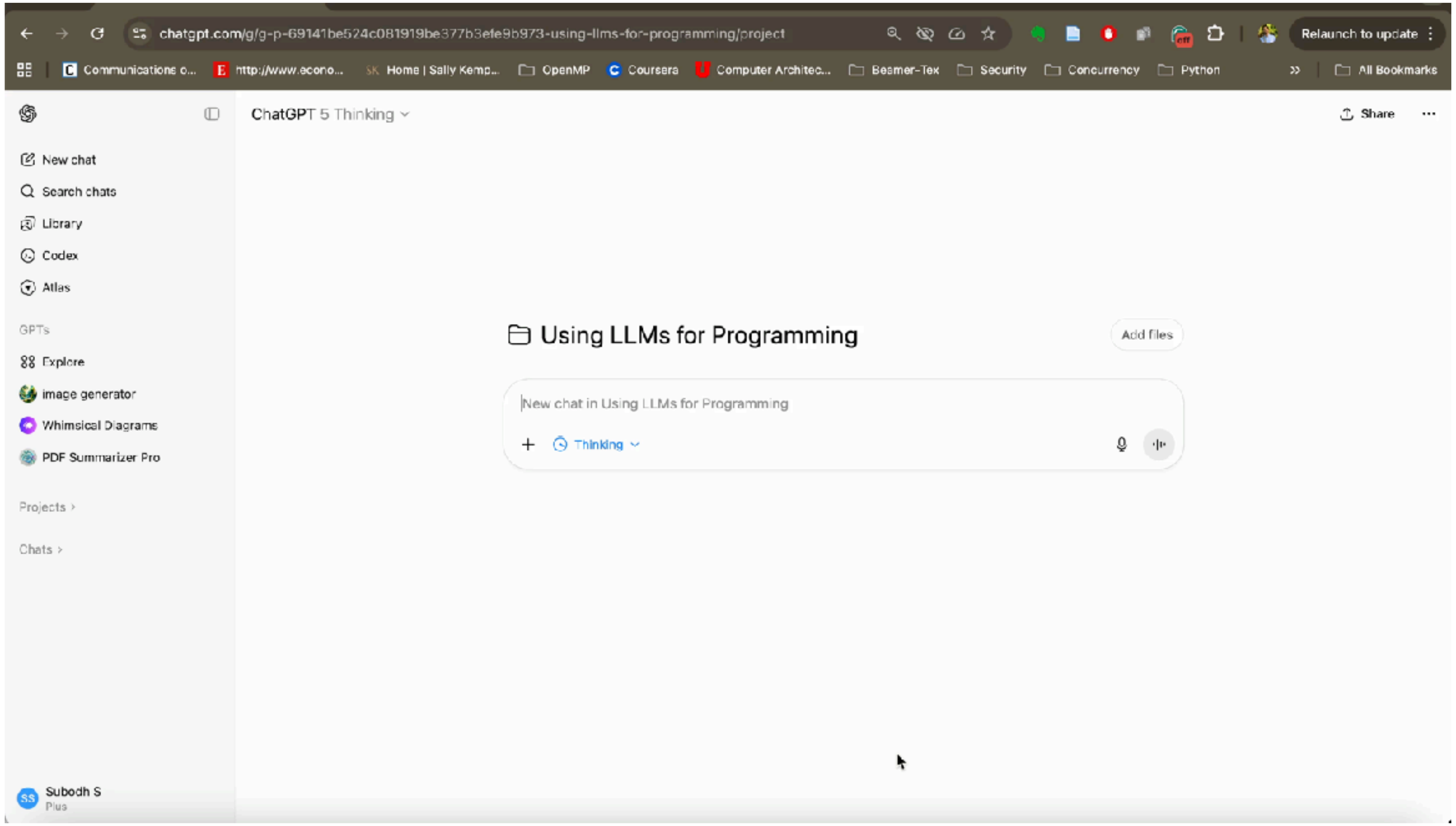
- GenAI Tools — Eg: ChatGPT, Gemini, Claude, etc.
  - Trained on large data sets; Predict next-tokens;
  - **Problems:**
    - Will make (possibly incorrect) assumptions on under-specified prompts (partial specs)
    - May end up using deprecated (ie OLD) programming language features
    - Optimises for **happy-path, may fail to report edge cases**
    - **Patterns may include bad coding practices**
    - **….**

How to Effectively Use GenAI for Programming Tasks

- Start with a prompt that explains the problem — essentially **a specification**
  - **Narrow the field of GenAI's o/p by restricting: For instance,**
    - **language: Python**
    - **allowed libraries: No use of iterTools**
    - **complexity targets: in** O($n\ logn$)
    - **style: Imperative (or Object Oriented or Functional, or logic or …)**
  - **1st Spec: Write a python program to implement a higher order function, where the input function, f, is applied to the corresponding elements of other iterable arguments.**

How to Effectively Use GenAI for Programming Tasks

- **1st Spec**: Write a python program to implement a higher order function, where the input function, f, is applied to the corresponding elements of other iterable arguments.

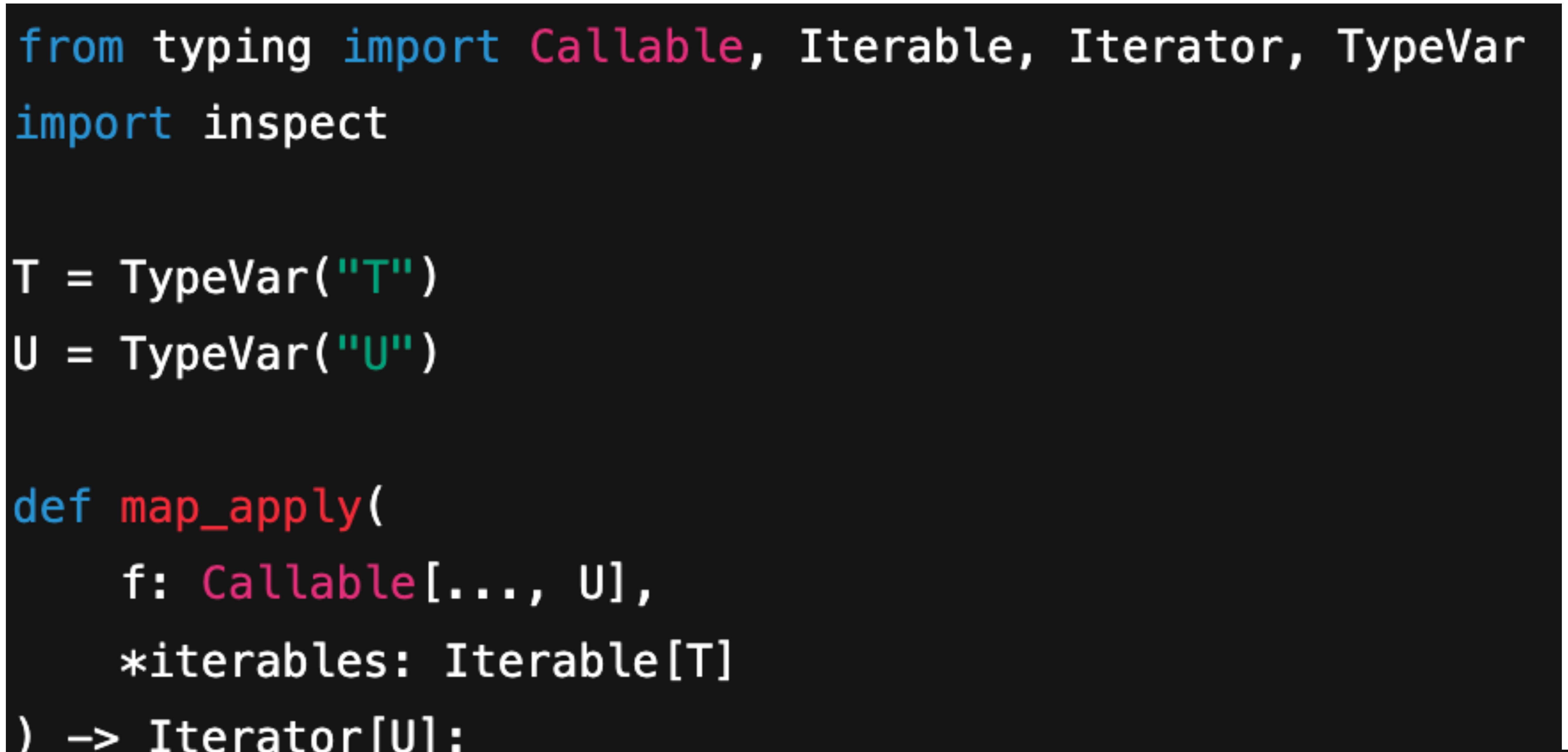


How to Effectively Use GenAI for Programming Tasks

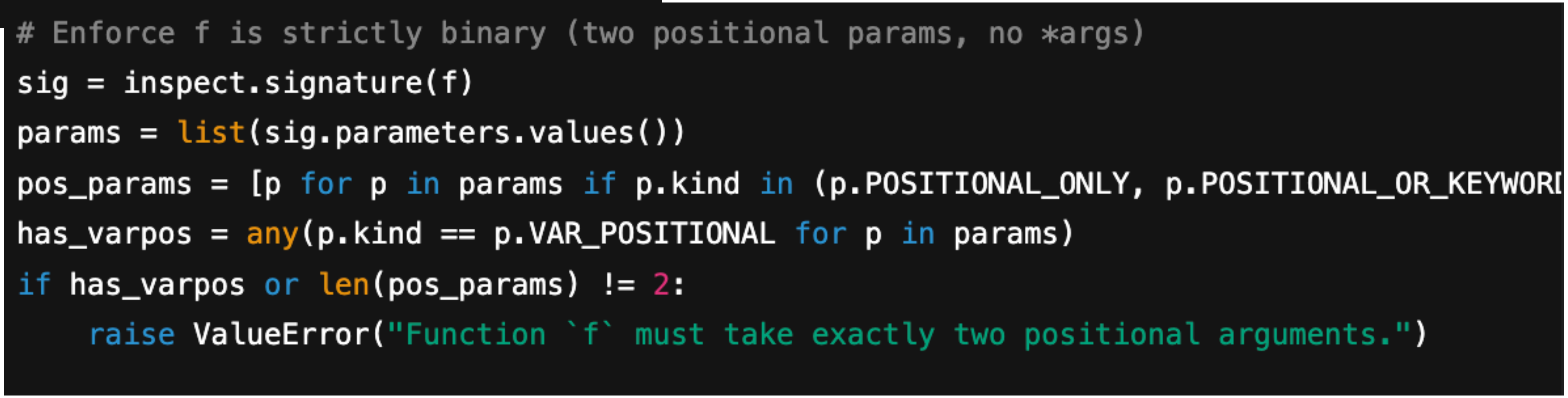
- **2nd Spec Refinement:**
  - Don't use libraries,
  - A two-variable function would require exactly two iterables of equal size
  - stick to the function interace:
```
def map_apply( f:
Callable[..., U],
*iterables: Iterable[T] )
-> Iterator[U]:
```
  - avoid the use of yield statement

- **3rd Spec Refinement:**
  - Error if argument function has argument count different than 2

ChatGPT Output

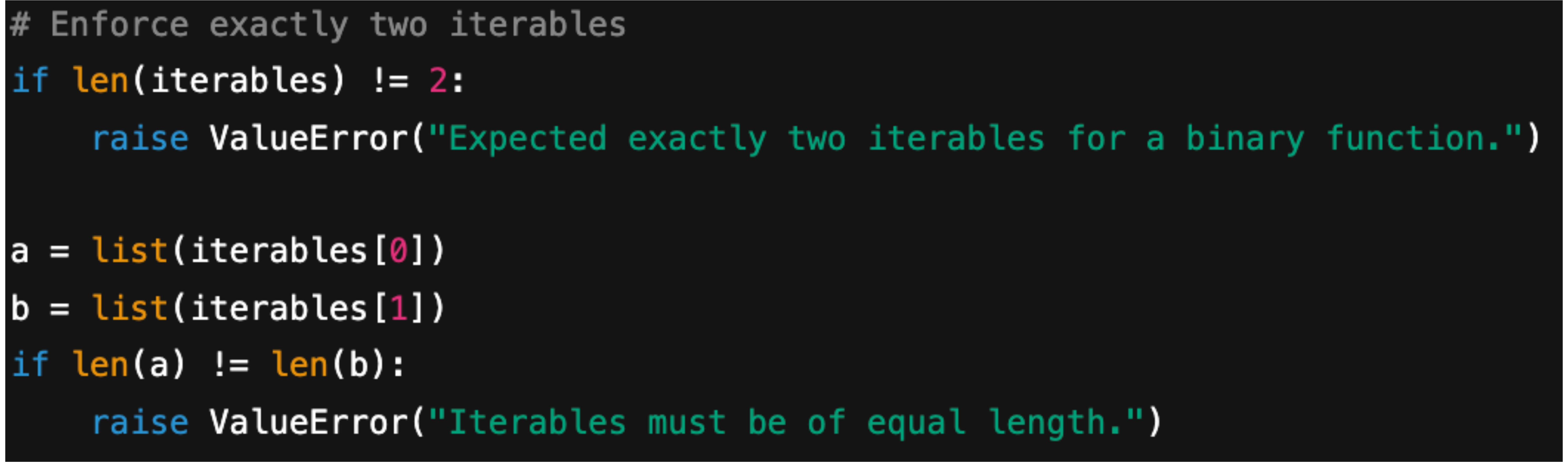


```
from typing import Callable, Iterable, Iterator, TypeVar
import inspect

T = TypeVar("T")
U = TypeVar("U")

def map_apply(
    f: Callable[..., U],
    *iterables: Iterable[T]
) -> Iterator[U]:
```

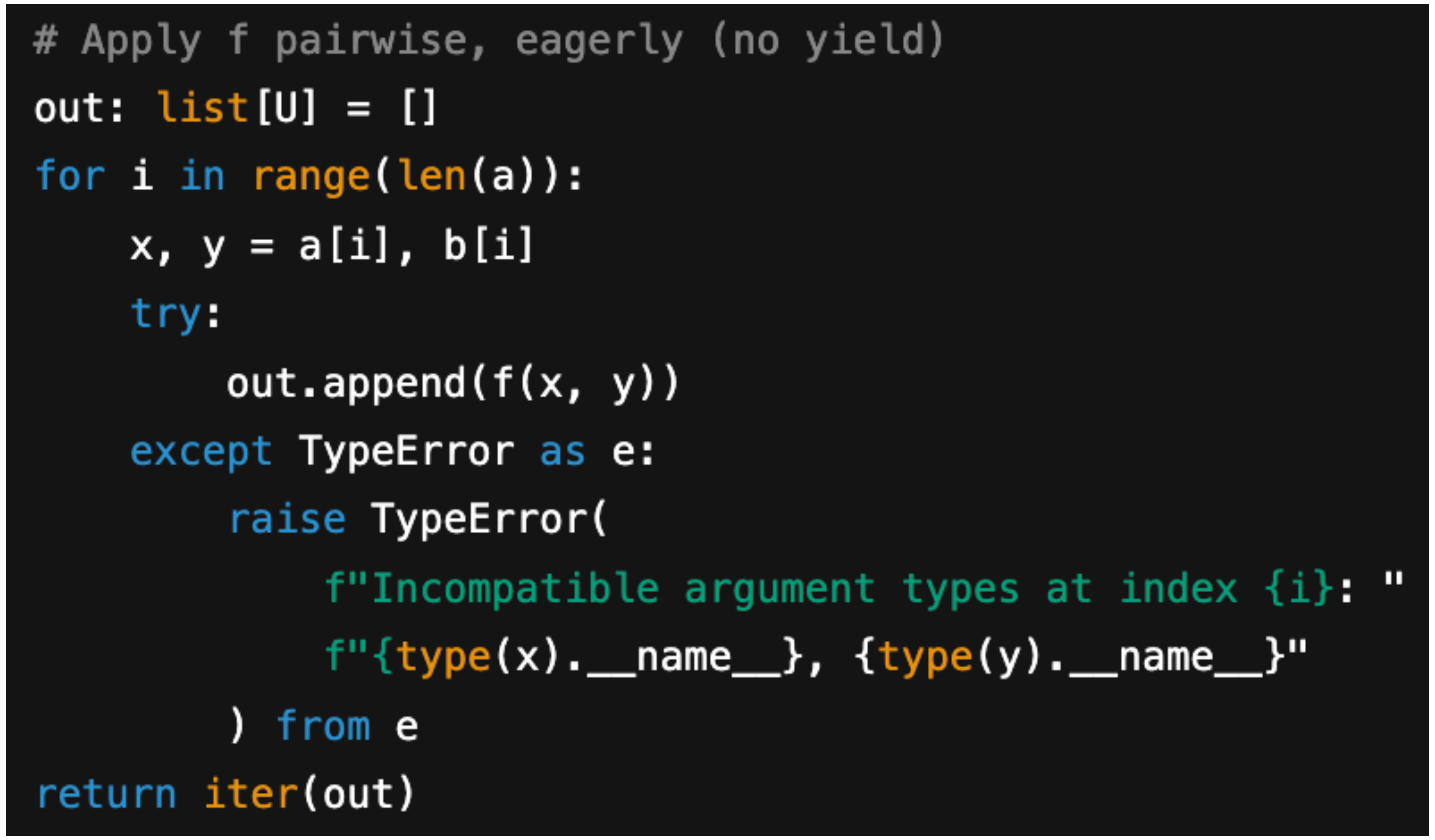


```
# Enforce f is strictly binary (two positional params, no *args)
sig = inspect.signature(f)
params = list(sig.parameters.values())
pos_params = [p for p in params if p.kind in (p.POSITIONAL_ONLY, p.POSITIONAL_OR_KEYWOR
has_varpos = any(p.kind == p.VAR_POSITIONAL for p in params)
if has_varpos or len(pos_params) != 2:
    raise ValueError("Function `f` must take exactly two positional arguments.")
```

ChatGPT Output



```
# Enforce exactly two iterables
if len(iterables) != 2:
    raise ValueError("Expected exactly two iterables for a binary function.")

a = list(iterables[0])
b = list(iterables[1])
if len(a) != len(b):
    raise ValueError("Iterables must be of equal length.")
```



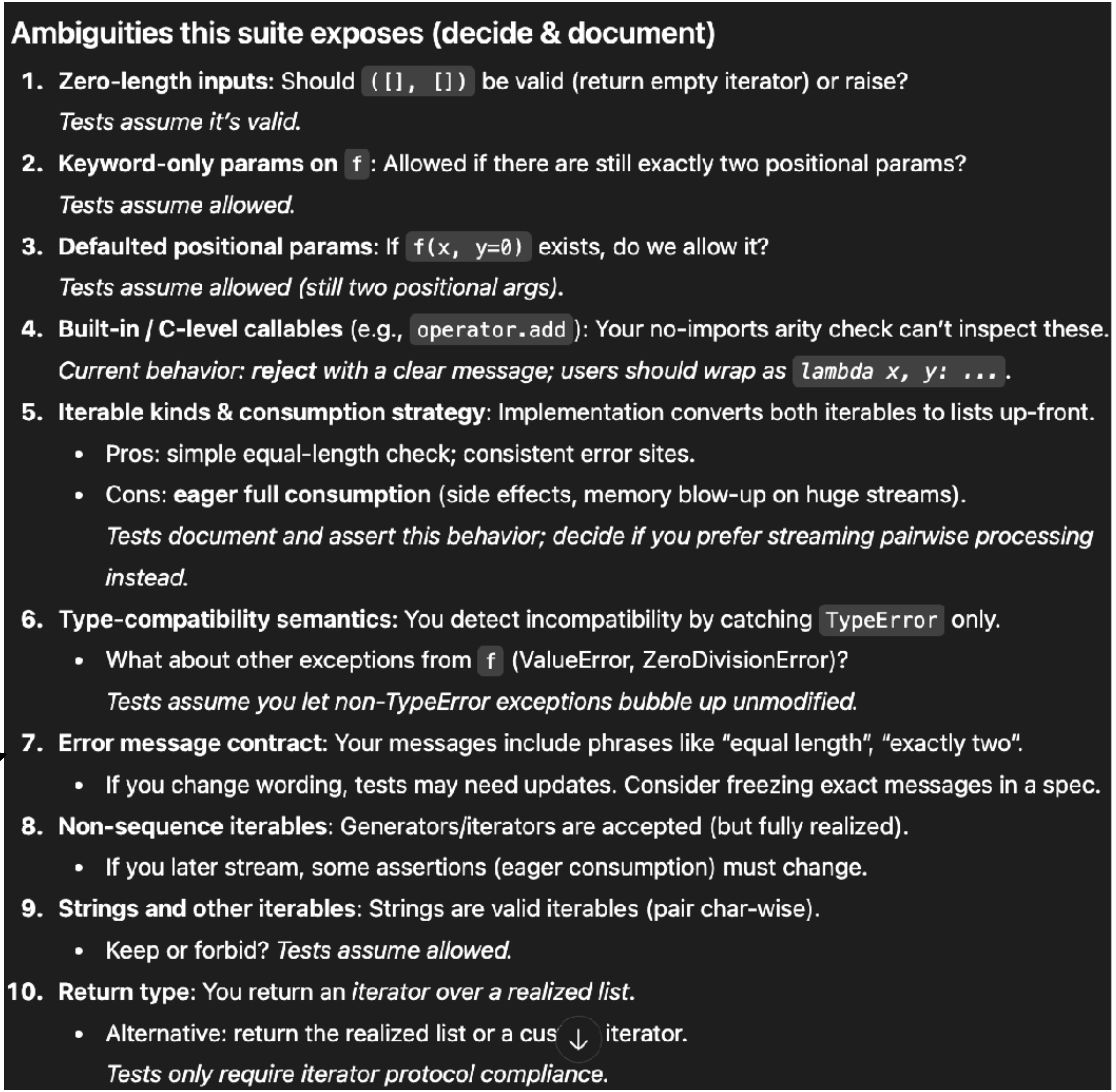
```
# Apply f pairwise, eagerly (no yield)
out: list[U] = []
for i in range(len(a)):
    x, y = a[i], b[i]
    try:
        out.append(f(x, y))
    except TypeError as e:
        raise TypeError(
            f"Incompatible argument types at index {i}: "
            f"{type(x).__name__}, {type(y).__name__}"
        ) from e
return iter(out)
```

How to Effectively Use GenAI for Programming Tasks

- **4th Spec Refinement:**
  - **Generate Unit Tests**
  - **List and resolve all ambiguities and corner cases**

Ambiguities listed by ChatGPT

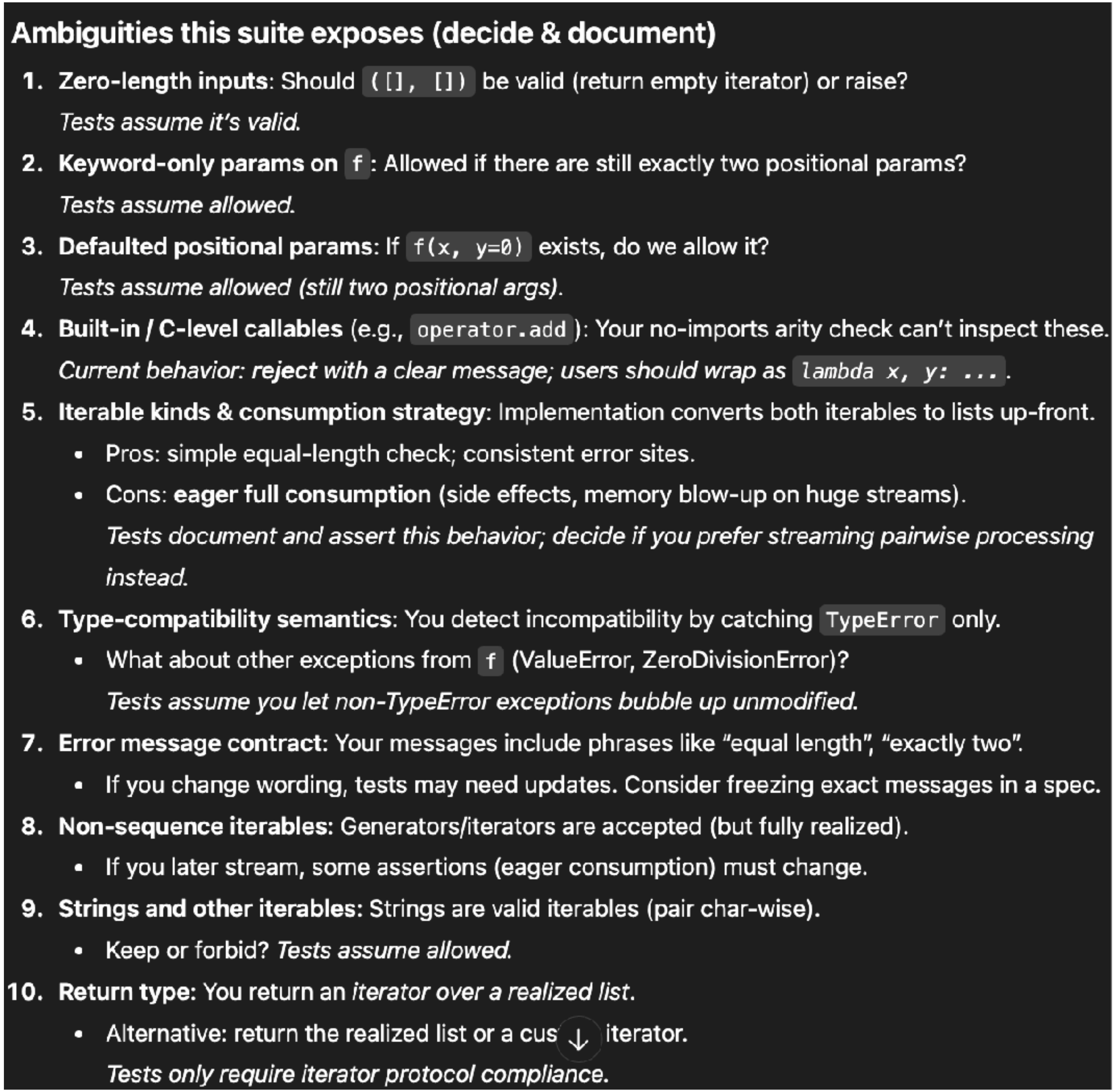
Some are clarifications others are clear underspecifications!



Ambiguities this suite exposes (decide & document)

1. **Zero-length inputs**: Should `([], [])` be valid (return empty iterator) or raise?
   Tests assume it's valid.
2. **Keyword-only params on `f`**: Allowed if there are still exactly two positional params?
   Tests assume allowed.
3. **Defaulted positional params**: If `f(x, y=0)` exists, do we allow it?
   Tests assume allowed (still two positional args).
4. **Built-in / C-level callables** (e.g., `operator.add`): Your no-imports arity check can't inspect these.
   *Current behavior: **reject** with a clear message; users should wrap as* `lambda x, y: ...`.
5. **Iterable kinds & consumption strategy**: Implementation converts both iterables to lists up-front.
   - Pros: simple equal-length check; consistent error sites.
   - Cons: **eager full consumption** (side effects, memory blow-up on huge streams).
     Tests document and assert this behavior; decide if you prefer streaming pairwise processing instead.
6. **Type-compatibility semantics:** You detect incompatibility by catching `TypeError` only.
   - What about other exceptions from `f` (ValueError, ZeroDivisionError)?
     Tests assume you let non-TypeError exceptions bubble up unmodified.
7. **Error message contract**: Your messages include phrases like "equal length", "exactly two".
   - If you change wording, tests may need updates. Consider freezing exact messages in a spec.
8. **Non-sequence iterables**: Generators/iterators are accepted (but fully realized).
   - If you later stream, some assertions (eager consumption) must change.
9. **Strings and other iterables**: Strings are valid iterables (pair char-wise).
   - Keep or forbid? *Tests assume allowed.*
10. **Return type**: You return an *iterator over a realized list*.
    - Alternative: return the realized list or a cus ↓ iterator.
      Tests only require iterator protocol compliance.

How to Effectively Use GenAI for Programming Tasks

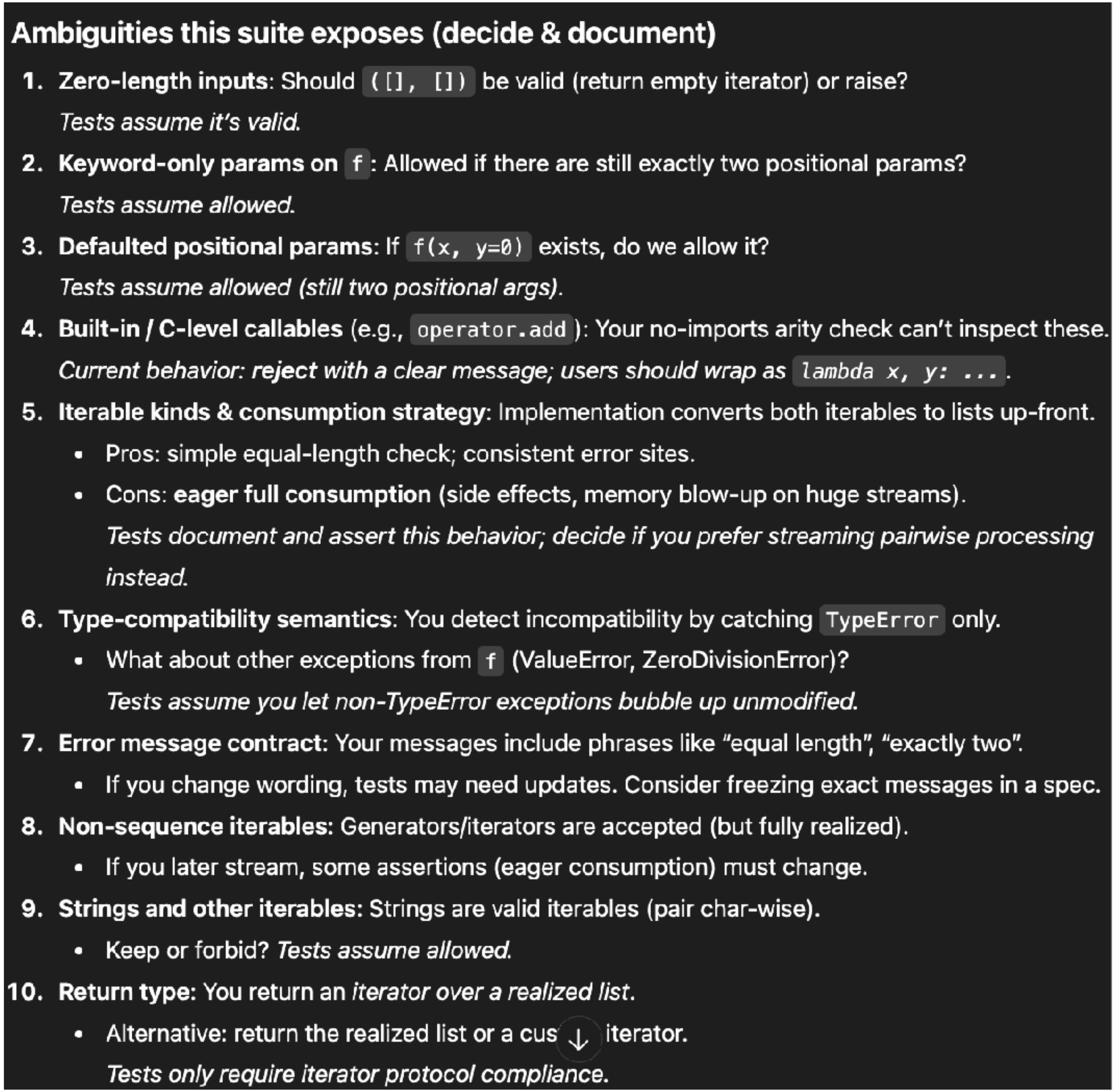


Ambiguities this suite exposes (decide & document)

1. **Zero-length inputs**: Should `([], [])` be valid (return empty iterator) or raise?
   Tests assume it's valid.
2. **Keyword-only params on** `f`: Allowed if there are still exactly two positional params?
   Tests assume allowed.
3. **Defaulted positional params**: If `f(x, y=0)` exists, do we allow it?
   Tests assume allowed (still two positional args).
4. **Built-in / C-level callables** (e.g., `operator.add`): Your no-imports arity check can't inspect these.
   Current behavior: ***reject with a clear message****; users should wrap as* `lambda x, y: ...`.
5. **Iterable kinds & consumption strategy**: Implementation converts both iterables to lists up-front.
   - Pros: simple equal-length check; consistent error sites.
   - Cons: **eager full consumption** (side effects, memory blow-up on huge streams).
     Tests document and assert this behavior; decide if you prefer streaming pairwise processing instead.
6. **Type-compatibility semantics**: You detect incompatibility by catching `TypeError` only.
   - What about other exceptions from `f` (ValueError, ZeroDivisionError)?
     Tests assume you let non-TypeError exceptions bubble up unmodified.
7. **Error message contract**: Your messages include phrases like "equal length", "exactly two".
   - If you change wording, tests may need updates. Consider freezing exact messages in a spec.
8. **Non-sequence iterables:** Generators/iterators are accepted (but fully realized).
   - If you later stream, some assertions (eager consumption) must change.
9. **Strings and other iterables:** Strings are valid iterables (pair char-wise).
   - Keep or forbid? *Tests assume allowed.*
10. **Return type:** You return an *iterator over a realized list*.
    - Alternative: return the realized list or a cus iterator.
      Tests only require iterator protocol compliance.

- **Validation Task:**
  - **Either validate the results on your own**
  - **Or ask ChatGPT to fix the "known" misses**

How to Effectively Use GenAI for Programming Tasks



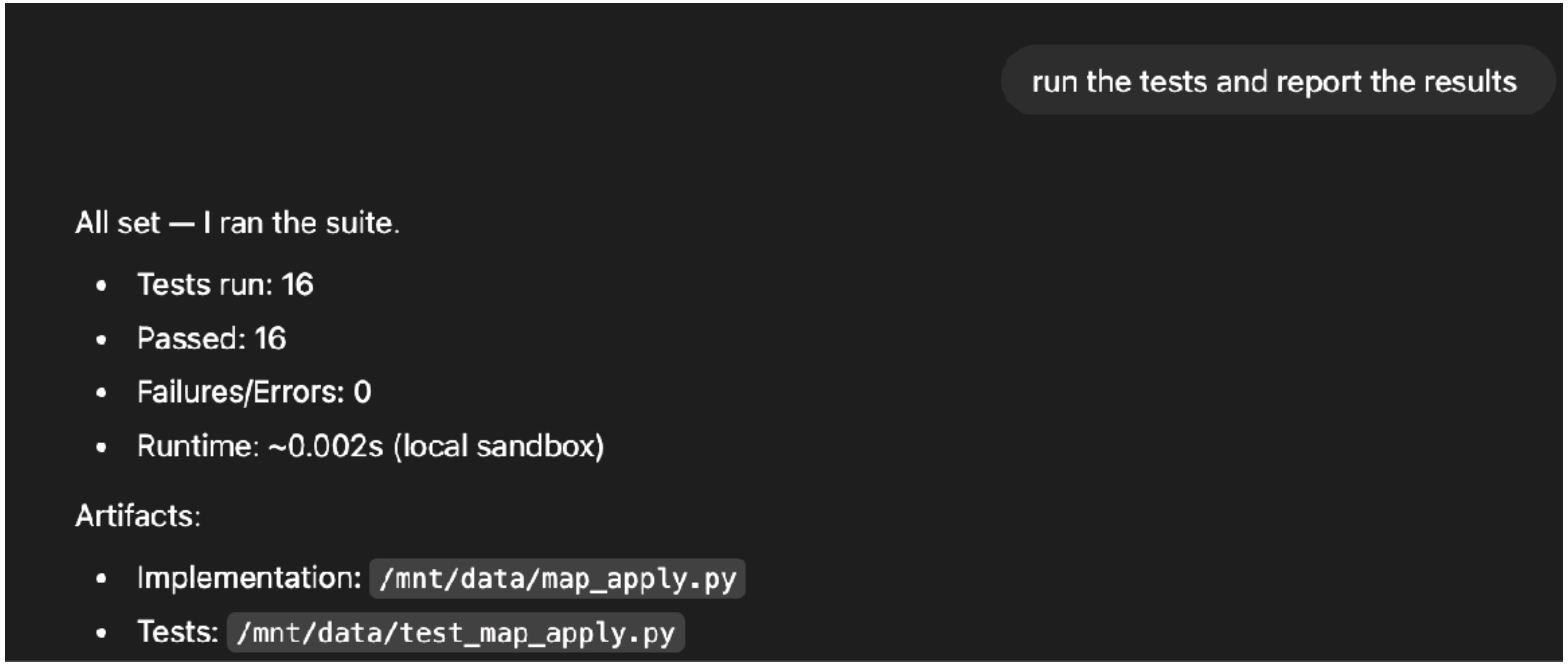
Ambiguities this suite exposes (decide & document)

1. **Zero-length inputs**: Should `([], [])` be valid (return empty iterator) or raise?
   Tests assume it's valid.
2. **Keyword-only params on `f`**: Allowed if there are still exactly two positional params?
   Tests assume allowed.
3. **Defaulted positional params**: If `f(x, y=0)` exists, do we allow it?
   Tests assume allowed (still two positional args).
4. **Built-in / C-level callables** (e.g., `operator.add`): Your no-imports arity check can't inspect these.
   *Current behavior: **reject with a clear message**; users should wrap as* `lambda x, y: ...`.
5. **Iterable kinds & consumption strategy**: Implementation converts both iterables to lists up-front.
   - Pros: simple equal-length check; consistent error sites.
   - Cons: **eager full consumption** (side effects, memory blow-up on huge streams).
     Tests document and assert this behavior; decide if you prefer streaming pairwise processing instead.
6. **Type-compatibility semantics**: You detect incompatibility by catching `TypeError` only.
   - What about other exceptions from `f` (ValueError, ZeroDivisionError)?
     Tests assume you let non-TypeError exceptions bubble up unmodified.
7. **Error message contract**: Your messages include phrases like "equal length", "exactly two".
   - If you change wording, tests may need updates. Consider freezing exact messages in a spec.
8. **Non-sequence iterables:** Generators/iterators are accepted (but fully realized).
   - If you later stream, some assertions (eager consumption) must change.
9. **Strings and other iterables:** Strings are valid iterables (pair char-wise).
   - Keep or forbid? *Tests assume allowed.*
10. **Return type:** You return an *iterator over a realized list*.
    - Alternative: return the realized list or a cus iterator.
      Tests only require iterator protocol compliance.

- **5th Spec Refinement:**
  - **1 return an empty iterator;**
  - **2 Kyeword only params not allowed;**
  - **3 yes allow them;**
  - **4 users must wrap as lambda;**
  - **5 keep the way it is;**
  - **6 Yes support other exceptions as well;**
  - **7 free the specs;**
  - **8 avoid nonsequence iterables;**
  - **9 keep strings;**
  - **10 return the realised list.**

How to Effectively Use GenAI for Programming Tasks

- To summarise:
  - Spec + Spec refinement
  - Ask the model to review the code for ambiguities or missed cases
    - Or validate yourself
  - Generate spec-based tests
  - Run the tests and obtain summary reports
    - Examine failed test cases and refine the spec, if needed



run the tests and report the results

All set — I ran the suite.

- Tests run: 16
- Passed: 16
- Failures/Errors: 0
- Runtime: ~0.002s (local sandbox)

Artifacts:

- Implementation: `/mnt/data/map_apply.py`
- Tests: `/mnt/data/test_map_apply.py`