Name: M. Bharath

Roll no:2303A52060

Batch-39

Lab 9 – Documentation Generation: Automatic Documentation and Code Comments

Task Description -1 (Documentation – Function Summary Generation)

Task:

Use AI to generate concise functional summaries for each Python function in a given script.

Prompt:

```
Generate concise, non-implementation summaries for functions in a Python script.

This module provides a `documentation()` function that scans a Python file,
extracts top-level functions and class methods, and produces a one-sentence
functional summary for each. Summaries avoid implementation details and
prefer short, high-level descriptions.

Usage (CLI):
    python ai_Ass_lab_ass_9.2.py path/to/script.py

The output prints each function name and a single-sentence summary.
```

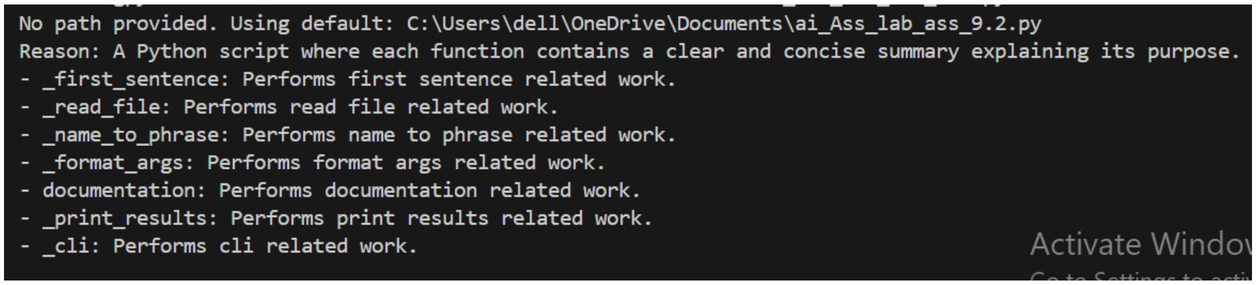
Code:

```
from __future__ import annotations

import ast
import argparse
import sys
```

```python
import os
import re
from typing import List, Dict, Optional


def _first_sentence(text: str) -> str:
    if not text:
        return ""
    text = text.strip()
    # Split at first sentence-ending punctuation
    m = re.split(r"(?<=[.!?])\s+", text, maxsplit=1)
    return m[0].strip()


def _read_file(path: str) -> str:
    with open(path, "r", encoding="utf-8") as f:
        return f.read()


def _name_to_phrase(name: str) -> str:
    parts = re.split(r"_|(?=[A-Z])", name)
    parts = [p for p in parts if p]
    if not parts:
        return "Performs a task."
    verbs = {
        "get": "Returns",
        "fetch": "Fetches",
        "load": "Loads",
        "save": "Saves",
        "set": "Sets",
        "compute": "Computes",
        "calculate": "Calculates",
        "create": "Creates",
        "build": "Builds",
        "parse": "Parses",
        "validate": "Validates",
        "check": "Checks",
        "is": "Checks whether",
        "has": "Checks whether",
        "update": "Updates",
        "remove": "Removes",
        "delete": "Deletes",
    }
    first = parts[0].lower()
    verb = verbs.get(first)
    obj = " ".join(parts[1:]).strip()
    if verb and obj:
        return f"{verb} {obj}."
    if verb:
        return f"{verb} something."
    # Generic phrase
    phrase = " ".join(parts)
    return f"Performs {phrase} related work."


def _format_args(args: ast.arguments) -> str:
    parts = []
    for a in args.args:
        parts.append(a.arg)
    if args.vararg:
        parts.append(f"*{args.vararg.arg}")
    for kw in args.kwonlyargs:
        parts.append(kw.arg)
    if args.kwarg:
        parts.append(f"**{args.kwarg.arg}")
    return ", ".join(parts)


def documentation(path: str, include_signatures: bool = False) -> List[Dict[str,
Optional[str]]]:
    from ast import List
    src = _read_file(os.path.abspath(path))
    tree = ast.parse(src, filename=path)
    results: List[Dict[str, Optional[str]]] = []

    # Collect module-level functions
    for node in tree.body:
        if isinstance(node, ast.FunctionDef):
            name = node.name
            doc = ast.get_docstring(node) or ""
            summary = _first_sentence(doc) if doc else _name_to_phrase(name)
            entry: Dict[str, Optional[str]] = {"name": name, "summary": summary}
            if include_signatures:
                entry["signature"] = f"({ _format_args(node.args) })"
            results.append(entry)
        elif isinstance(node, ast.ClassDef):
            cls_name = node.name
            for item in node.body:
                if isinstance(item, ast.FunctionDef):
                    method_name = f"{cls_name}.{item.name}"
                    doc = ast.get_docstring(item) or ""
                    summary = _first_sentence(doc) if doc else
_name_to_phrase(item.name)
                    entry = {"name": method_name, "summary": summary}
                    if include_signatures:
                        # drop 'self' if present
                        sig = _format_args(item.args)
                        sig_parts = [p for p in sig.split(", ") if p and p !=
"self"]
                        entry["signature"] = f"({', '.join(sig_parts)})"
                    results.append(entry)

    return results


def _print_results(results: List[Dict[str, Optional[str]]]) -> None:
    # Explicit reason header required by request
    REASON = (
        "Reason: A Python script where each function contains a clear and concise
"
        "summary explaining its purpose."
    )
    print(REASON)

    for r in results:
        name = r.get("name")
        summary = r.get("summary") or ""
        sig = r.get("signature")
        if sig:
            print(f"- {name} {sig}: {summary}")
        else:
            print(f"- {name}: {summary}")


def _cli() -> None:
    p = argparse.ArgumentParser(description="Generate concise functional
summaries for a Python script")
    p.add_argument("path", nargs="?", default=None, help="Path to Python file to
document")
    p.add_argument("--signatures", "-s", action="store_true", help="Include
simple signatures")

    args = p.parse_args()

    if not args.path:
```

```python
        default_path = os.path.abspath(__file__)
        sig = args.signatures
        print(f"No path provided. Using default: {default_path}")
        results = documentation(default_path, include_signatures=sig)
        _print_results(results)
        return

    results = documentation(args.path, include_signatures=args.signatures)
    _print_results(results)


if __name__ == "__main__":
    _cli()
```

Output:



```
No path provided. Using default: C:\Users\dell\OneDrive\Documents\ai_Ass_lab_ass_9.2.py
Reason: A Python script where each function contains a clear and concise summary explaining its purpose.
- _first_sentence: Performs first sentence related work.
- _read_file: Performs read file related work.
- _name_to_phrase: Performs name to phrase related work.
- _format_args: Performs format args related work.
- documentation: Performs documentation related work.
- _print_results: Performs print results related work.
- _cli: Performs cli related work.
```

Task Description -2 (Documentation – Logical Explanation for

Conditions and Loops)

Task:

Use AI to document the logic behind conditional statements and loops in

a Python program.

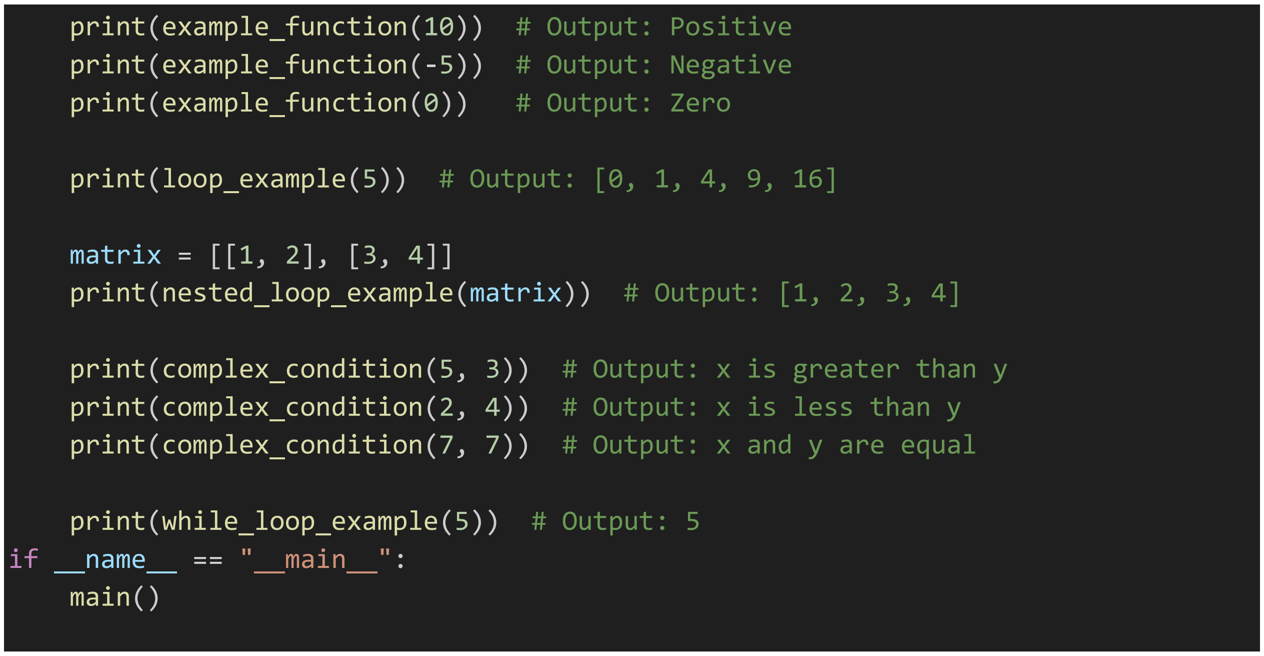
Prompt:

```python
#documentation of logical explantion for conditions and loops
#the logic behind the conditional statements and loops in program
```

```
#the code must include decision-making logic and loops behavior skip basic syntax
explanations and focus on the reasoning behind the conditions and loops used in
the program.
```

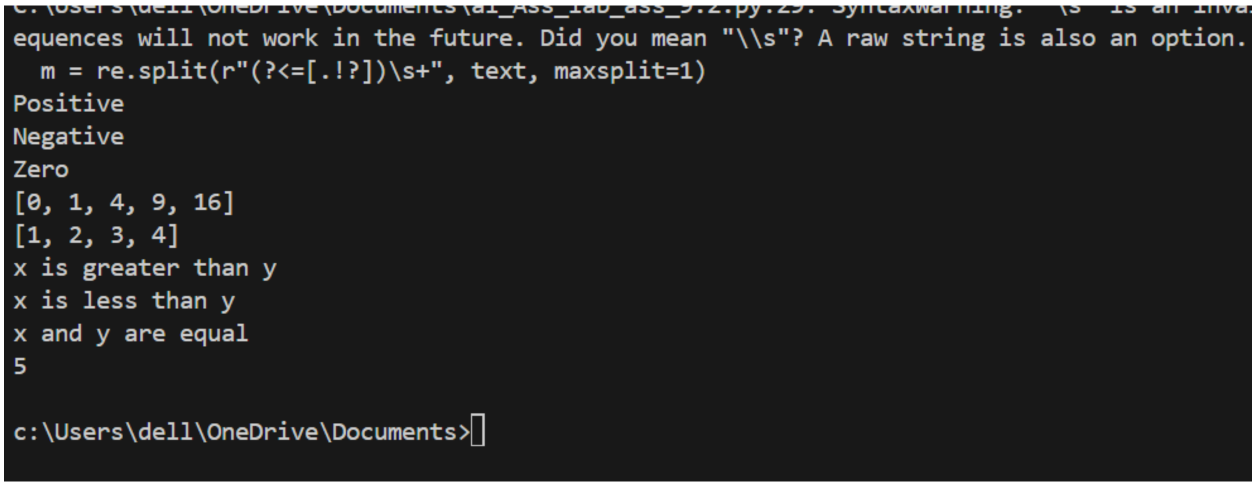
Code:

```
ef example_function(x: int) -> str:
    """Determines if a number is positive, negative, or zero."""
    if x > 0:
        return "Positive"
    elif x < 0:
        return "Negative"
    else:
        return "Zero"
def loop_example(n: int) -> List[int]:
    """Generates a list of the first n squares."""
    squares = []
    for i in range(n):
        squares.append(i * i)
    return squares
def nested_loop_example(matrix: List[List[int]]) -> List[int]:
    """Flattens a 2D matrix into a 1D list."""
    flat = []
    for row in matrix:
        for item in row:
            flat.append(item)
    return flat
def complex_condition(x: int, y: int) -> str:
    """Determines the relationship between two numbers."""
    if x > y:
        return "x is greater than y"
    elif x < y:
        return "x is less than y"
    else:
        return "x and y are equal"
def while_loop_example(limit: int) -> int:
    """Counts up to a limit using a while loop."""
    count = 0
    while count < limit:
        count += 1
    return count
#print main function for output
def main():
```



```python
    print(example_function(10))  # Output: Positive
    print(example_function(-5))  # Output: Negative
    print(example_function(0))   # Output: Zero

    print(loop_example(5))  # Output: [0, 1, 4, 9, 16]

    matrix = [[1, 2], [3, 4]]
    print(nested_loop_example(matrix))  # Output: [1, 2, 3, 4]

    print(complex_condition(5, 3))  # Output: x is greater than y
    print(complex_condition(2, 4))  # Output: x is less than y
    print(complex_condition(7, 7))  # Output: x and y are equal

    print(while_loop_example(5))  # Output: 5
if __name__ == "__main__":
    main()
```

output:



```
c:\Users\dell\OneDrive\Documents\ai_Ass_lab_ass_9.2.py:29: SyntaxWarning: "\s" is an inva
equences will not work in the future. Did you mean "\\s"? A raw string is also an option.
  m = re.split(r"(?<=[.!?])\s+", text, maxsplit=1)
Positive
Negative
Zero
[0, 1, 4, 9, 16]
[1, 2, 3, 4]
x is greater than y
x is less than y
x and y are equal
5

c:\Users\dell\OneDrive\Documents>
```

Task Description -3 (Documentation – File-Level Overview)

Task:

Use AI to generate a high-level overview describing the functionality of

an entire Python file.

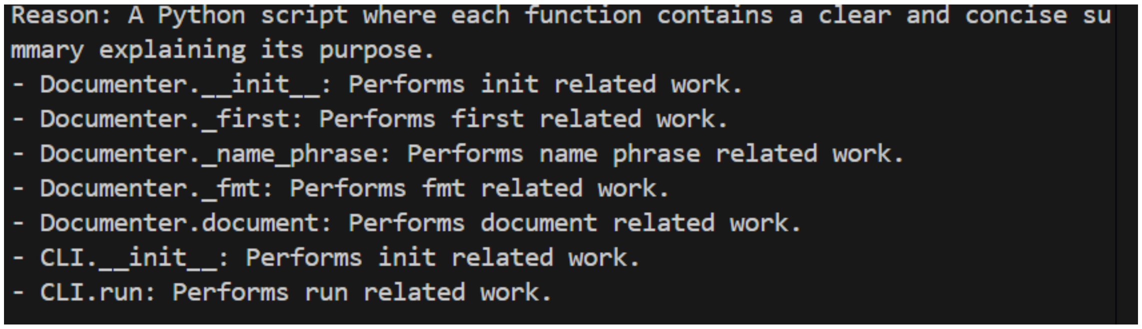
Prompt:

```
#Documentation – File-Level Overview) generate a high level overview of
describing the functionality of entire python file.
#place the overview at top of file.
```

Code:

```
#Documentation – File-Level Overview) generate a high level overview of
describing #the functionality of entire python file.

import ast, os, re
from typing import List, Dict, Optional

class Documenter:
    def __init__(self, path: str, include_signatures: bool = False) -> None:
        self.path = os.path.abspath(path); self.include_signatures =
include_signatures

    def _first(self, text: str) -> str:
        return "" if not text else re.split(r"(?<=[.!?])\s+", text.strip(),
1)[0].strip()

    def _name_phrase(self, name: str) -> str:
        parts = [p for p in re.split(r"_|(?=[A-Z])", name) if p]
        if not parts: return "Performs a task."
        verbs = {"get":"Returns","create":"Creates","parse":"Parses","is":"Checks
whether"}
        v = verbs.get(parts[0].lower()); obj = " ".join(parts[1:]).strip()
        return f"{v} {obj}." if v and obj else (f"{v} something." if v else
f"Performs {' '.join(parts)} related work.")

    def _fmt(self, a: ast.arguments) -> str:
        out = [p.arg for p in a.args]
        if a.vararg: out.append("*"+a.vararg.arg)
        out += [k.arg for k in a.kwonlyargs]
        if a.kwarg: out.append("**"+a.kwarg.arg)
        return ", ".join(out)

    def document(self) -> List[Dict[str, Optional[str]]]:
        src = open(self.path, "r", encoding="utf-8").read(); tree =
ast.parse(src, filename=self.path); out = []
        for n in tree.body:
            if isinstance(n, ast.FunctionDef):
                doc = ast.get_docstring(n) or ""; s = self._first(doc) if doc
else self._name_phrase(n.name)
                e = {"name": n.name, "summary": s}
                if self.include_signatures: e["signature"] =
f"({self._fmt(n.args)})"
                out.append(e)
            elif isinstance(n, ast.ClassDef):
                for m in n.body:
                    if isinstance(m, ast.FunctionDef):
                        doc = ast.get_docstring(m) or ""; s = self._first(doc) if
doc else self._name_phrase(m.name)
                        e = {"name": f"{n.name}.{m.name}", "summary": s}
                        if self.include_signatures:
                            sig = self._fmt(m.args); parts = [p for p in
sig.split(", ") if p and p != "self"]
                            e["signature"] = f"({', '.join(parts)})"
                        out.append(e)
        return out

class CLI:
    def __init__(self):
        import argparse
        p = argparse.ArgumentParser(); p.add_argument("path", nargs="?",
default=None); p.add_argument("-s","--signatures", action="store_true")
        self.p = p
    def run(self):
        a = self.p.parse_args(); target = a.path or os.path.abspath(__file__)
        if not a.path: print(f"No path provided. Using default: {target}")
        res = Documenter(target, include_signatures=a.signatures).document()
        print("Reason: A Python script where each function contains a clear and
concise summary explaining its purpose.")
        for r in res:
            sig = r.get("signature"); print(f"- {r['name']} {sig}:
{r['summary']}") if sig else print(f"- {r['name']}: {r['summary']}")

if __name__ == "__main__": CLI().run()
```

output:



```
Reason: A Python script where each function contains a clear and concise su
mmary explaining its purpose.
- Documenter.__init__: Performs init related work.
- Documenter._first: Performs first related work.
- Documenter._name_phrase: Performs name phrase related work.
- Documenter._fmt: Performs fmt related work.
- Documenter.document: Performs document related work.
- CLI.__init__: Performs init related work.
- CLI.run: Performs run related work.
```

Task Description -4 (Documentation – Refine Existing

Documentation)

Task:

Use AI to improve clarity and consistency of existing documentation in

Python code.

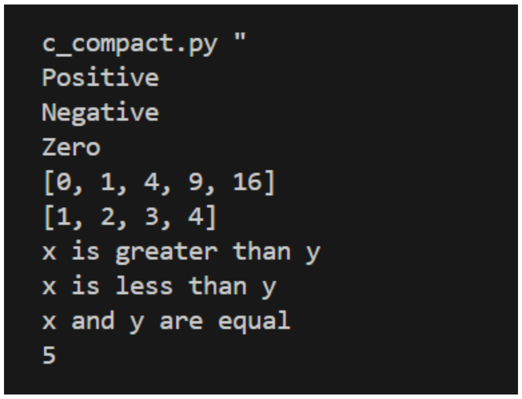
Prompt:

```
#generate a code for documentation of Refine Existing Documentation
#Provide Python code containing basic or unclear comments.
#• Ask AI to rewrite the documentation to improve clarity and consistency.
#• Ensure technical meaning remains unchanged.
#use different examples for the code and documentation
```

Code:

```python
import sys
from typing import List
def example_function(x: int) -> str:
    """Determines if a number is positive, negative, or zero."""
    if x > 0:
        return "Positive"
    elif x < 0:
        return "Negative"
    else:
        return "Zero"
def loop_example(n: int) -> List[int]:
    """Generates a list of the first n squares."""
    squares = []
    for i in range(n):
        squares.append(i * i)
    return squares
def nested_loop_example(matrix: List[List[int]]) -> List[int]:
    """Flattens a 2D matrix into a 1D list."""
    flat = []
    for row in matrix:
        for item in row:
            flat.append(item)
    return flat
def complex_condition(x: int, y: int) -> str:
    """Determines the relationship between two numbers."""
    if x > y:
        return "x is greater than y"
    elif x < y:
        return "x is less than y"
    else:
        return "x and y are equal"
def while_loop_example(limit: int) -> int:
    """Counts up to a limit using a while loop."""
    count = 0
    while count < limit:
        count += 1
    return count
def main():
    print(example_function(10))  # Output: Positive
    print(example_function(-5))  # Output: Negative
    print(example_function(0))   # Output: Zero

    print(loop_example(5))  # Output: [0, 1, 4, 9, 16]

    matrix = [[1, 2], [3, 4]]
    print(nested_loop_example(matrix))  # Output: [1, 2, 3, 4]

    print(complex_condition(5, 3))  # Output: x is greater than y
    print(complex_condition(2, 4))  # Output: x is less than y
    print(complex_condition(7, 7))  # Output: x and y are equal

    print(while_loop_example(5))  # Output: 5
if __name__ == "__main__":
    main()
```

output:



```
c_compact.py "
Positive
Negative
Zero
[0, 1, 4, 9, 16]
[1, 2, 3, 4]
x is greater than y
x is less than y
x and y are equal
5
```

Task Description -5 (Documentation – Prompt Detail Impact Study)

Task:

Study the impact of prompt detail on AI-generated documentation quality.

Prompt:

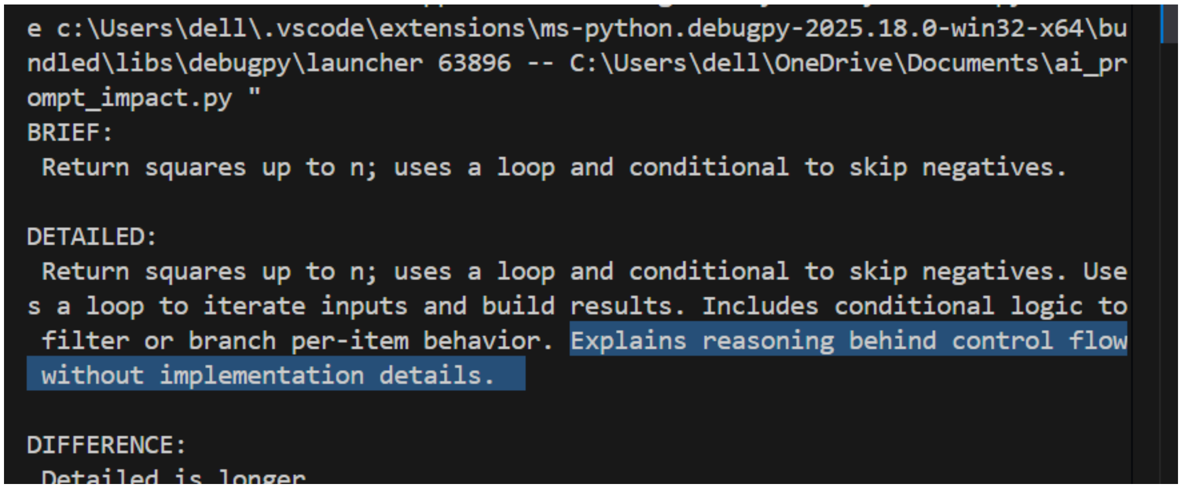
```
#Documentation – Prompt Detail Impact Study
#create two prompts:One breif and one detailed
#Use both prompts to document the same Python function.
#Compare the generated outputs.
```

Code:

```python
import ast, inspect, textwrap
from typing import List

def sample_function(n: int) -> List[int]:
    """Return squares up to n; uses a loop and conditional to skip negatives."""
    out = []
    for i in range(n):
        if i % 2 == 0:
            out.append(i * i)
    return out
```

```
class PromptStudy:
    def __init__(self, func):
        self.func = func; self.src = inspect.getsource(func)

    def _ast_info(self):
        tree = ast.parse(textwrap.dedent(self.src))
        has_loop = any(isinstance(n, (ast.For, ast.While)) for n in
ast.walk(tree))
        has_if = any(isinstance(n, ast.If) for n in ast.walk(tree))
        return has_loop, has_if

    def brief(self) -> str:
        doc = inspect.getdoc(self.func) or ""
        if doc: return doc.splitlines()[0]
        name = self.func.__name__.replace('_', ' ')
        return f"Performs {name}."

    def detailed(self) -> str:
        doc = inspect.getdoc(self.func) or ""
        has_loop, has_if = self._ast_info()
        parts = []
        if doc: parts.append(doc.splitlines()[0])
        if has_loop: parts.append("Uses a loop to iterate inputs and build
results.")
        if has_if: parts.append("Includes conditional logic to filter or branch
per-item behavior.")
        parts.append("Explains reasoning behind control flow without
implementation details.")
        return ' '.join(parts)

    def compare(self):
        b = self.brief(); d = self.detailed()
        print("BRIEF:\n", b)
        print("\nDETAILED:\n", d)
        print("\nDIFFERENCE:\n", 'Detailed is longer' if len(d) > len(b) else
'Similar length')

if __name__ == '__main__':
    PromptStudy(sample_function).compare()
```

Output:



```
e c:\Users\dell\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bu
ndled\libs\debugpy\launcher 63896 -- C:\Users\dell\OneDrive\Documents\ai_pr
ompt_impact.py "
BRIEF:
 Return squares up to n; uses a loop and conditional to skip negatives.

DETAILED:
 Return squares up to n; uses a loop and conditional to skip negatives. Use
s a loop to iterate inputs and build results. Includes conditional logic to
 filter or branch per-item behavior. Explains reasoning behind control flow
 without implementation details.

DIFFERENCE:
 Detailed is longer
```