3.2.7 編寫第一個 Python 程式

經過前面的步驟，我們已經成功安裝了 Python 直譯器、VS Code 整合開發環境和 GitHub Copilot。現在，是時候編寫我們的第一個 Python 程式了！這一刻，我們終於要從程式編寫小白變成真正的 Python 開發者了。

雖然編寫第一個程式看似簡單，但對於初學者來說，可能會有很多疑問：程式應該包含哪些內容？如何執行程式？出錯了應該如何偵錯？別擔心，我們可以向 GitHub Copilot 求助，它會一步一步教我們編寫和執行第一個 Python 程式。

> 嗨，GitHub Copilot，我想編寫第一個 Python 程式，可以列印出「Hello, World!」。請問應該怎麼做呢？

> 太好了，你已經邁出了編寫程式的第一步！要編寫一個列印「Hello, World!」的 Python 程式，非常簡單。
>
> 1. 開啟 VS Code，建立一個新檔案，將其命名為「hello.py」。
> 2. 在檔案中輸入以下程式碼。
>
> ```
> print('Hello, World!')
> ```
>
> 3. 儲存檔案，然後按 F5 鍵或者選擇功能表列的「Run」→「Start Debugging」命令來執行程式。
> 4. 在 VS Code 的終端機中，你應該能看到輸出的「Hello, World!」。

就是這麼簡單！ print() 函式是 Python 中 常用的輸出函式，它可以將括號內的內容列印到主控台上。你可以嘗試列印其他內容，或者多列印幾行。記住，Python 中的字串需要用單引號或雙引號括起來。

在 GitHub Copilot 的指導下，我們輕鬆地完成了第一個 Python 程式的編寫和執行。我們學會了如何使用最基本的 print() 函式來列印內容。透過向 GitHub Copilot 提問，我們得到了清楚、全面的解答，並且還瞭解到 print() 函式還有更多用法等待我們去探索。

圖 3-1

在檔案起始處輸入包含提示詞的註解，描述所需的樣式。

```
/* 為 Chrome 擴充功能的快顯視窗設定更寬、更高的樣式

使用易於存取的顏色和字體

使 h1 元素易於閱讀

滑鼠指標停在按鈕上時進行高亮顯示
```

6 提示工程：利用 GitHub Copilot 快速編寫程式碼

儘管像 GitHub Copilot 這樣的工具可以透過提供智慧的程式碼建議來輔助程式編寫，但關鍵仍在於我們要能夠清楚地表達自己的程式編寫目標和需求，工具只能根據我們提供的資訊來產生程式碼。因此，在學習 Python 的過程中，還需要培養將複雜問題分解為簡單、明確步驟的能力。為此，我們將在本章探討如何在 IDE 整合開發環境（使用 VS Code）中應用提示工程，讓你能更有效率地與程式輔助工具互動。

6.1 提示工程概念詳解

在使用 GitHub Copilot 的過程中，許多開發者經常會遇到一些挫折和困惑。例如，有時他們期望 GitHub Copilot 能夠根據簡單的註解或函式名稱自動產生完整的函式，但實際得到的是不準確或無關的程式碼片段。這些挫折和困惑常常源於開發者對 GitHub Copilot 能力的誤解，或者沒有提供足夠清楚和詳細的上下文資訊。

另外，GitHub 官方的一篇部落格文章中也提到了這種問題。文章作者分享了他們使用 GitHub Copilot 輔助編寫繪製霜淇淋甜筒程式碼的經歷。起初，無論作者如何嘗試，GitHub Copilot 提供的建議不是毫無關聯，就是沒有任何建議。這一經歷讓他們對 GitHub Copilot 的實用性產生了懷疑。但當作者深入研究 GitHub Copilot 處理資訊的方式後，有了新的認識。他們意識到，GitHub Copilot 並非一個黑盒工具，而是需要開發者學會與其有效溝通。他們透過調整提示的方法，針對輸入資訊的品質和重點給予最佳化，最終得到了滿意的結果。

面對這些挑戰，越來越多的開發者像這篇文章的作者一樣，意識到 GitHub Copilot 的「提示工程」的重要性。他們發現，充分發揮 GitHub Copilot 的潛力，僅依賴 GitHub Copilot 本身是不夠的，他們需要掌握一項新的技能——如何設計出清楚、準確、資訊充足的提示工程，進一步引導 GitHub Copilot 產生符合預期的程式碼。

提示工程涉及如何有效地建構和表達提示，以引導語言模型產生我們期望的輸出。它需要深入瞭解語言模型的能力和局限性，並運用技巧來充分利用模型的潛力。

我們使用 GitHub Copilot 程式編寫的方式是提示工程，而不僅是提示詞。提示詞只是輸入的內容，在最終提交給底層模型之前，這些提示詞都會經過提示工程的處理和最佳化。

GitHub Copilot 的提示工程整合了提示詞、上下文資訊和 IDE 環境資訊，透過 Codex 模型來瞭解使用者的需求並產生相關的程式碼建議，以輔助程式編寫。圖 6-1 展示了 GitHub Copilot 的完整提示工程，整合了三種輸入，由大型語言模型（OpenAI 的 Codex）產生程式碼建議的工作機制。

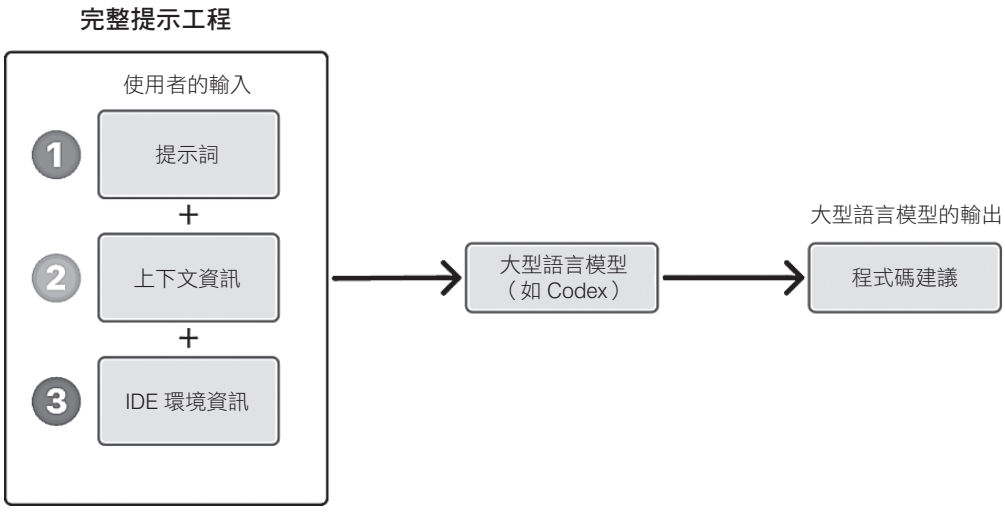

圖 6-1

詳細步驟說明：

1. IDE 環境資訊和上下文資訊：在 VS Code 中編寫程式碼時，會擷取程式碼和相關上下文資訊（例如檔案名稱、檔案路徑、程式註解，以及周邊的程式區塊等）（如圖 6-2 中編號 2 和 3 所示）。
2. 提示產生和編譯過程：使用者的輸入（圖 6-2 所示的編號 1）、IDE 環境資訊和上下文被編譯成一個「提示」（Prompt），這個提示將被輸入機器學習模型。
3. 輸入 GitHub Copilot 模型：提示被輸入 GitHub Copilot 的後台模型（圖 6-2 所示的大型語言模型節點）。

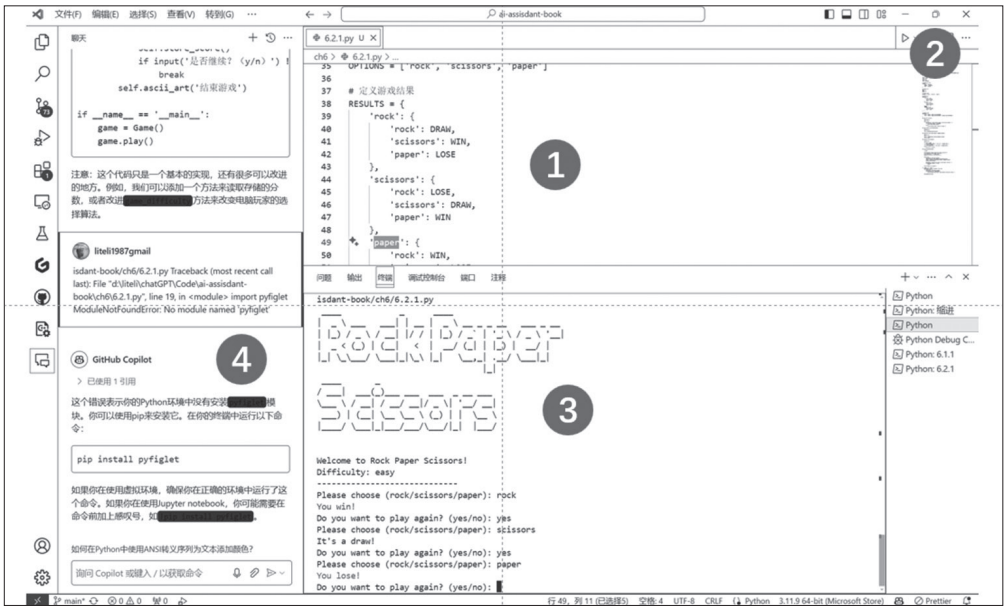

圖 6-2

4. 後台演算法產生程式碼建議：GitHub Copilot 的後台演算法處理輸入，即時產生程式碼建議（圖 6-2 所示的程式碼建議節點）。
5. 輸出程式碼建議：建議被顯示在 VS Code 中，幫助開發者更快地編寫程式碼或解決程式編寫問題。

漸進提供

隨著遊戲核心功能的完善，可以適時引入一些新的關鍵字，以拓展遊戲的功能和趣味性，如表 6-5 所示。

表 6-5

專業關鍵字	類型	描述	範例
遊戲難度層級	可自行設定	表達需要支援不同難度層級，以適應不同玩家	允許玩家透過命令列參數選擇遊戲難度，如「easy」「hard」等，以影響電腦隨機策略的產生演算法
遊戲資料本機存放區	資料持續性	提出需要在本機持續儲存一些遊戲資料	在本機 JSON 檔案中儲存使用者的遊戲勝負統計資訊，持續儲存遊戲紀錄

透過漸進式地引入新的專業關鍵字，我們引導 GitHub Copilot 持續擴展遊戲功能，提升了其趣味性和可玩性。

實作效果

在這個 Python 剪刀石頭布遊戲的開發過程中，我們在不同階段引入了「Python 3.x」、「使用者輸入驗證」、「ANSI 逸出序列」等專業關鍵字。結果表明，GitHub Copilot 產生的程式碼高度契合專案需求，遊戲基本功能和互動體驗都得到了有效實作，大幅減輕了開發人員的工作量。

實作的第一步：將以上表格中的提示詞註解組合為一個多行註解，寫到 6.2.1.py 的檔案開頭。

```
"""

使用 Python 3.9 開發剪刀石頭布遊戲。
遵循 PEP 8 規範組織遊戲的程式碼結構，提高維護性。
將遊戲邏輯、使用者互動和結果判定分別封裝在不同的函式或類別中。
使用 if 語句和字串方法驗證使用者輸入是否為合法的遊戲選擇（剪刀 / 石頭 / 布）。
使用 while 迴圈控制進行多輪遊戲，直到使用者主動退出。
利用 random 模組產生表示剪刀、石頭、布的亂數，模擬電腦玩家的選擇。
使用 ANSI 逸出序列為遊戲結果輸出添加顏色，提升視覺效果。
使用 pyfiglet 等函式庫產生炫酷的 ASCII 藝術字，作為遊戲開始和結束的提示文字。
允許玩家透過命令列參數選擇遊戲難度，如 "easy""hard" 等，影響電腦隨機策略的產生演算法。
在本機 JSON 檔案中儲存使用者的遊戲勝負統計資訊，持續儲存遊戲紀錄。
```

```
        'paper': WIN
    },
    'paper': {
        'rock': WIN,
        'scissors': LOSE,
        'paper': DRAW
    }
}

# 定義遊戲難度
DIFFICULTY = {
    'easy': lambda: random.choice(OPTIONS),
    'hard': lambda: random.choice(OPTIONS + [random.choice(OPTIONS)])
}

def get_user_choice():
    """
    取得使用者選擇
    """
    while True:
        user_choice = input("Please choose (rock/scissors/paper): ")
        if user_choice.lower() in OPTIONS:
            return user_choice.lower()
        else:
            print("Invalid input. Please try again.")

def get_computer_choice(difficulty):
    """
    取得電腦選擇
    """
    return DIFFICULTY[difficulty]()

def determine_result(user_choice, computer_choice):
    """
    判斷遊戲結果
    """
    result = RESULTS[user_choice][computer_choice]
    return result

def display_result(result):
    """
    顯示遊戲結果
    """
    if result == WIN:
        print(COLORS['GREEN'] + "You win!" + COLORS['END'])
    elif result == LOSE:
```

7.3 LLM 程式編寫的環境準備

在這一節，我們將學習如何使用 Python 和 OpenAI 的 LLM 模型進行程式編寫。以下步驟中的操作細則已在第 2 章介紹，這裡透過簡單的「麵條式流程」再次整理呈現，幫助初學者更輕鬆上手，減輕學習負擔。我們深知，對於初學者來說，這些資訊都散落在網際網路上，需要不斷搜尋和辨別真偽，閱讀很多模棱兩可的解釋，才能準備好程式編寫的環境。而實際上，按照這一套流程，就可以準備好程式編寫環境。

安裝 Python

1. 存取 python.org，下載適合作業系統的 Python 版本並安裝。macOS 內建 Python，無須安裝。
2. 在安裝過程中，確保勾選「Add Python to PATH」（新增 Python 到路徑）選項，這將使我們能夠從命令列中存取 Python。

安裝 OpenAI 套件

1. 開啟命令提示字元介面或終端機。
2. 使用以下命令，透過 pip 安裝 OpenAI 套件：pip install openai。
3. 如果已經安裝了 OpenAI 套件，可以使用以下命令將其更新到最新版本：pip install --upgrade openai。

安裝 Git

存取 git-scm.com，下載適合作業系統的 Git 版本並安裝。

克隆倉儲

1. 開啟命令提示字元介面或終端機。
2. 使用以下命令克隆倉儲：git clone <repository-url>。其中，<repository-url> 是倉儲的 URL。如果沒有倉儲，請前往 github.com 註冊帳號和新建一個倉儲。

進入倉儲目錄

使用以下命令進入克隆的倉儲目錄：cd python-llm（假設倉儲名稱為 python-llm）。

編輯檔案

連上 visualstudio.microsoft.com，下載 VS Code 文字編輯器，使用 VS Code 開啟倉儲中的 Python 檔案進行編輯。按 Ctrl + Shift + X 快速鍵，直接搜尋 GitHub Copilot 和 GitHub Copilot Chat 外掛程式。安裝後，授權在 VS Code 上登入 GitHub 帳號，訂閱付費資源可以根據官方的指引完成（可以切換至中文）。

儲存模型廠商提供的 API 金鑰

將 API 金鑰儲存到本機，如 openai api key.txt。例如，使用月之暗面的 Kimi API 存取 platform.moonshot.cn，進入 API Key 管理介面，找到 API 金鑰。其他模型廠商的操作大致相同，一般是進入開發者平台，然後進入個人中心的 API Key 管理介面，找到金鑰。

確保不要將包含 API 金鑰的檔案提交到程式碼倉儲中，以保護金鑰。

在 Python 腳本中使用 API 金鑰

在 Python 腳本中，使用以下程式碼讀取 API 金鑰檔案並設定 openai.api_key 和 openai.base_url。

```
# 以月之暗面的 Kimi API 為例
from openai import OpenAI
openai.api_key = " 你的金鑰 "
openai.base_url = "https://api.mo**shot.cn/v1"
```

模型廠商開發者平台會提供以 HTTP 為基礎的 API 服務的接入，並且絕大部分相容了 OpenAI SDK。

執行 Python 腳本

使用以下命令執行 Python 檔案以測試程式碼：python hello_world.py（假設檔案名稱為 hello_world.py）。

提交更改到 Git 倉儲

1. 使用以下命令將更改加入到 Git 暫存區：git add .。
2. 使用以下命令提交更改到本機倉儲：git commit -am "initial commit"。
3. 使用以下命令將更改推送到遠端倉儲：git push。

接下來，可以探索倉儲中的範例程式碼，瞭解如何使用 LLM 產生文字、進行對話、回答問題等。隨著學習的深入，我們將能夠建立自己的 Python 腳本，並利用 LLM 的強大功能建構令人興奮的應用程式。

7.4 在本機開發一個 LLM 聊天機器人

為了幫助初學者瞭解程式開發和部署上線的不同階段，我們將按照里程碑的方式介紹從基礎到企業級上線的學習路徑。如圖 7-1 所示，這些里程碑可以幫助我們整理思緒，清楚瞭解程式開發處於什麼階段。

圖 7-1

本機執行

在本機執行應用軟體（圖 7-1 左起第一個學習里程碑）是最簡單的方式。需要在本機編寫程式碼，安裝相依函式庫，執行應用軟體並進行本機存取。在本機開發和測試應用程式是重要的第一步。這一步可以幫助我們快速確認程式碼是否正常工作，而無須考慮複雜的部署問題。我們只需要具備基本的程式編寫和相依性管理技能。

Gradio 是一個開源的 Python 函式庫，旨在使機器學習和資料科學應用程式的開發和共用變得更加簡單和直觀。透過 Gradio，開發者可以快速建立互動式的使用者介面，以展示和測試機器學習模型、資料集和演算法。Gradio 允許開發者在幾行程式碼內建立友善的使用者介面。這些介面可用於輸入資料、執行模型並顯示輸出，非常適合快速原型設計和測試。Gradio 的設計簡潔，開發者無須具備前端開發的知識即可建立功能豐富的網頁應用，只需專注於 Python 程式碼，相關的前端介面會自動產生。

Kimi API 是月之暗面公司 Kimi 開放平台提供的 API 服務，主要用於自然語言處理和文字產生。呼叫 Kimi API，可以實作 Kimi 聊天對話的效果。使用 API 服務前，需要在它的主控台中建立一個 API 金鑰。我們使用預設模型，型號是 moonshot-v1-8k，它適用於產生短文本，最大上下文長度為 8,000 Tokens。

第一步：取得 API 金鑰和 API 呼叫範例程式碼。

首先，需要從模型廠商處取得開發的 API 金鑰和 API 呼叫範例程式碼。可以存取 Kimi 開放平台的文件頁面（platform.moonshot.cn/docs），查閱 API 呼叫範例程式碼。API 金鑰可以在 Kimi 開放平台的主控台頁面（platform.moonshot.cn/console/api-keys）產生和查看。

第二步，將需求和 API 呼叫範例程式碼組合為提示詞，如：

```
我是初學者，請從新建專案開始，
指導我如何在本機開發一個使用 Kimi API 和 Gradio 的聊天機器人。
Kimi API 文件：
---
from openai import OpenAI
from dotenv import load_dotenv

client = OpenAI(
    api_key = "$MOONSHOT_API_KEY",
    base_url = "https://api.mo**shot.cn/v1",
)

completion = client.chat.completions.create(
    model = "moonshot-v1-8k",
    messages = [
        {"role": "system", "content": "你是 Kimi，由 Moonshot AI 提供的人工智慧助手，你更擅長中文和英文的對話。你會為使用者提供安全、有幫助、準確的回答。同時，你會拒絕一切涉及恐怖主義、種族歧視、黃色暴力等問題的回答。Moonshot AI 為專有名詞，不可翻譯成其他
```

```
語言。"},
        {"role": "user", "content": "你好，我叫李雷，1+1 等於多少？"}
    ],
    temperature = 0.3,
)

print(completion.choices[0].message.content)
---
```

我們可以在 GitHub Copilot 的聊天介面中使用這樣的提示詞進行詢問，這一步最好在聊天介面完成。行內和多行註解的方式聚焦於在程式碼區域產生程式碼。在詢問流程方面，聊天介面可以提供追問和知識管理（由於我們與它共享相同的背景知識，因此給 GitHub Copilot 的提示詞可以不附加歷史訊息，GitHub Copilot 依然能做出更聰明且精準的回應）。透過 GitHub Copilot 的回饋，我們可以按照以下順序完成開發。

````
好的，讓我們從頭開始建立一個使用 Kimi API 和 Gradio 的聊天機器人。以下是詳細步驟：

1. 首先，我們需要在你的電腦上安裝 Python。你可以從 Python 官方網站下載並安裝。

2. 安裝 Python 後，我們需要安裝一些 Python 函式庫。開啟命令提示字元或終端機，然後輸入以下命
   令：

```bash
pip install gradio openai python-dotenv
```

3. 建立一個新的 Python 檔案，例如 `chatbot.py`。

4. 在同一目錄下建立一個名為 `.env` 的檔案，然後在其中加入你的 Kimi API 金鑰，如下所示：

```dotenv
MOONSHOT_API_KEY=your_api_key_here
```

5. 現在，我們可以開始編寫 Python 程式碼了。在 `chatbot.py` 檔案中，複製並貼上以下程式
   碼：

```import gradio as gr
from openai import OpenAI
from dotenv import load_dotenv
import os

load_dotenv()
````
````

8.3 利用 GitHub Copilot 輔助開發單元測試

單元測試是軟體開發過程中一個重要但常被忽視的環節。在傳統開發中，編寫單元測試是一項耗時且容易出錯的任務。程式開發人員需要手工編寫大量的測試程式碼，設計各種輸入資料來確認函式的行為是否符合預期，不僅工作量大，而且很容易漏掉一些極端案例，影響測試的全面性。

軟體開發是一門複雜的技術，人們往往要應對兩個困難。第一個困難是人類記憶力和注意力有限。在複雜的商業邏輯下，開發者特別容易忽視單元測試（單元測試單獨進行且有大量重複工作，導致人們特別容易忽視它）。第二個困難是由於大意，對很多步驟沒有進行例行檢查，而很多人會說：「我明明記得。」這兩個困難的最直接表現就是故意跳過單元測試，直到程式真的發生了錯誤，才回頭進行單元測試以排除錯誤。

Python 內建的 unittest 模組為初學者提供了完備的單元測試支援，使編寫和執行測試變得簡單且直覺。然而，即便有了這樣的測試模組，編寫高品質的單元測試仍然需要大量的時間和經驗。

如何才能在學習 Python 的同時掌握編寫優秀單元測試的技巧呢？

人工智慧技術的發展為這個問題提供了一個創新的解決方案。像 GitHub Copilot 這樣的 AI 程式編寫助手，透過學習大量優質的測試程式碼，總結出了編寫測試的「策略」。它可以根據函式的定義，自動產生相關的單元測試程式碼。就像我們透過模仿優秀作文和範文來學習寫作技巧一樣，GitHub Copilot 提供了大量優質的單元測試範例供我們參考和模仿。

初學者可以先讓 GitHub Copilot 為自己的程式碼產生單元測試，然後仔細閱讀和瞭解這些測試程式碼。GitHub Copilot 產生的測試程式碼通常包括各種極端情況和例外輸入的檢查，這些都是手寫測試程式碼時容易遺漏的。透過學習這些程式碼，我們可以掌握測試案例設計的要點。

但 GitHub Copilot 畢竟只是一個 AI，它的建議並不總是完美的。我們不應該完全依賴它，而要學會批判性思考：這個測試有沒有遺漏，是否還有別的極端情況需要考慮？我們要以 GitHub Copilot 為起點，不斷思考如何改進和超越它提供的方案。

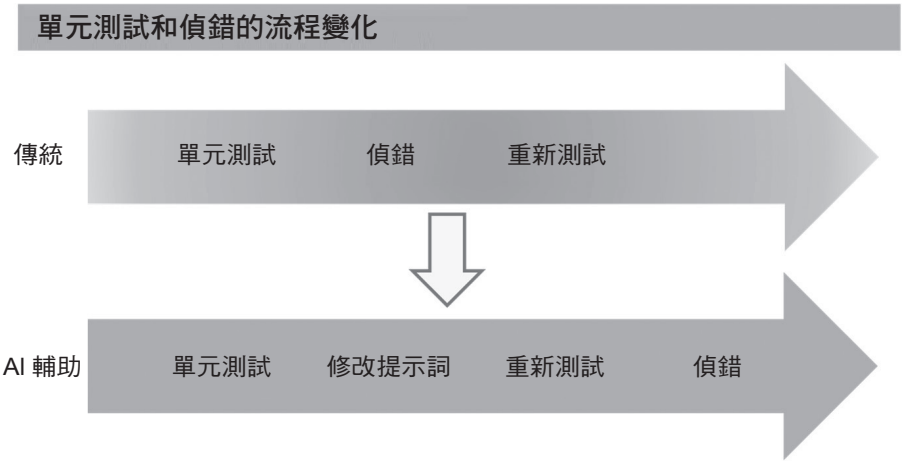

圖 8-2

AI 輔助流程具體說明如下。

1. 單元測試：透過單元測試發現程式碼中的錯誤。
2. 修改提示詞：根據錯誤資訊調整提示詞，讓 AI 重新產生程式碼。
3. 重新測試：對新產生的程式碼進行測試，確認是否修復了之前的錯誤。
4. 必要時偵錯：如果錯誤依舊存在，則再進行偵錯，以深入分析和修復。

透過這種方式，我們可以更有效地利用 AI 的學習和自我最佳化能力，提升開發效率和程式碼品質。這種方式的優勢在於充分利用 GitHub Copilot 產生程式碼的特性，避免陷入傳統偵錯的複雜流程。從提示詞入手，透過調整提示詞來改進 GitHub Copilot 產生的程式碼，可以大幅減少手動偵錯花費的時間和精力，提高整體開發效率。

舉個例子，我們使用 GitHub Copilot 產生了一段程式碼，用於計算兩個數的最大公約數，但在測試中發現，當輸入負數時，程式會回報錯誤。傳統的偵錯思路是直接開啟程式碼，找到出錯的地方，然後修改程式碼邏輯。而在新的 AI 輔助開發流程下，我們首先要修改給 GitHub Copilot 的提示，告訴它程式碼需要支援負數輸入。例如，將提示詞改為「請產生一段計算兩個整數最大公約數的程式碼，要求支援負整數的輸入」，然後讓 GitHub Copilot 重新產生程式碼，再進行測試。如果重新產生的程式碼通過了測試，就不必再進行偵錯了。