表單的頂端

Python 程式設計核心概念：迴圈、條件式、函式與錯誤處理

1. 錯誤和例外處理 (Errors and Exception Handling)

Python 程式碼在執行時可能遇到兩類常見錯誤：「語法錯誤 (Syntax Error)」和「例外 (Exception)」。

1.1 語法錯誤 (Syntax Error)

* 又稱「剖析錯誤 (parsing error)」。
* 通常發生在程式碼不符合 Python 語法規則時，例如缺少冒號。
* Python 直譯器會指出錯誤發生的行數，並用箭頭指向檢測到錯誤的地方，但實際需要修改的位置可能有所不同。
* **範例**：while True print('Hello world') 會導致 SyntaxError: invalid syntax，箭頭會指向 print，但實際錯誤是 print 前缺少冒號。

1.2 例外 (Exception)

* 即使語法正確，程式執行時仍可能發生錯誤，這些執行時錯誤稱為「例外」。
* 例外不一定都嚴重，可以在程式中進行處理。
* 常見的例外類型包括：
* ZeroDivisionError：除以零。
* NameError：未定義的變數或名稱。
* TypeError：操作數類型不支援（例如字串與整數相加）。
* **例外處理**：
* 使用 try...except 陳述式來捕獲和處理例外。
* try 子句內的程式碼在執行時可能引發例外。
* except 子句用於指定要處理的例外類型。如果多個 except 子句匹配，則只有第一個匹配的會被觸發。
* except 子句可以指定一個變數來綁定例外實例，其引數通常儲存在 .args 屬性中。
* **範例**：
* try:
* raise Exception('spam', 'eggs')
* except Exception as inst:
* print(type(inst)) # <class 'Exception'>
* print(inst.args) # ('spam', 'eggs')
* print(inst) # ('spam', 'eggs')
* BaseException 是所有例外的基底類別，Exception 是所有非致命例外的基底類別。
* 捕獲 Exception 可以作為通配符，但建議盡可能具體指定要處理的例外類型。
* 常見模式是列印或記錄例外，然後重新引發它，以便呼叫函式也能處理。
* **else 子句**：
* try...except 陳述式可以包含一個可選的 else 子句，它必須放在所有 except 子句之後。
* 如果 try 子句沒有引發任何例外，則 else 子句的程式碼會被執行。
* 使用 else 子句優於在 try 中添加額外的程式碼，以避免意外捕獲非 try...except 保護的例外。
* **finally 子句**：
* try...finally 陳述式定義了清理動作，無論 try 塊中是否發生例外，finally 子句都會執行。
* 即使在 try 塊中遇到 break、continue 或 return 陳述式，finally 子句也會在這些陳述式執行前先執行。
* 如果 finally 子句包含 return 陳述式，則回傳值將來自 finally 子句，而不是 try 子句。
* **範例**：
* def bool\_return():
* try:
* return True
* finally:
* return False
* print(bool\_return()) # False
* **例外鏈接 (Exception Chaining)**：允許一個例外因為另一個例外而被引發。
* **使用者自定義例外**：可以通過繼承 Exception 或其子類別來定義自己的例外類型，以提供更具體的錯誤處理。
* **範例**：定義模組的根例外類別，並讓所有模組內部的例外都繼承自它，例如 Error, InvalidDensityError, InvalidVolumeError。
* **處理多個無關的例外 (Exception Groups)**：Python 3.11 引入了 ExceptionGroup 和 except\* 語法，允許同時處理多個不相關的例外。
* except\* 可以選擇性地只處理群組中與特定類型匹配的例外。

1.3 互動式偵錯 (Debugging with pdb)

* Python 內建的 pdb 模組提供互動式偵錯功能。
* 可以透過在程式碼中呼叫 breakpoint() 內建函式來啟動偵錯器。
* 在 (Pdb) 提示符下，可以檢查變數值 (p <name>)、查看所有區域變數 (locals())、匯入模組、檢查全域狀態等，甚至修改正在執行的程式。

2. 迴圈 (Loops)

迴圈是程式設計中的核心概念，用於重複執行一段程式碼，避免重複性工作，提高程式碼的簡潔性和可維護性。

2.1 while 迴圈

* while 迴圈是一種條件控制迴圈，只要指定的布林條件為 True，就會重複執行其程式碼區塊。
* 在每次迭代開始時檢查條件。如果條件一開始就為 False，迴圈體可能完全不執行。
* **基本語法**：
* while condition:
* # statements
* **應用**：當執行次數不確定，而是依賴於某個條件何時變為 False 時使用。
* **範例**：持續接收使用者輸入直到輸入特定值 ('0') 才停止。
* name = ''
* while name != '0':
* name = input('請輸入姓名:[輸入0,停止輸入]')
* print(f"Hello! {name}")
* **無限迴圈 (Infinite Loop)**：如果迴圈條件永遠為 True，則會產生無限迴圈，導致程式卡住。需要確保條件最終會變為 False，或使用 break 語句跳出。
* **與 else 子句結合**：while 迴圈也可以選擇性地搭配 else 子句。如果迴圈是正常結束（即條件變為 False 而非遇到 break），則 else 塊會執行。
* 然而，**建議避免使用 else 塊搭配迴圈**，因為其行為不直觀且容易混淆。

2.2 for 迴圈

* for 迴圈是一種計數控制迴圈，用於迭代集合（如列表、字串、元組、字典等）中的每個元素。
* **基本語法**：
* for variable in collection:
* # statements
* **應用**：
* **迭代列表元素**：
* students = ['Hermione', 'Harry', 'Ron']
* for student in students:
* print(student)
* **搭配 range() 函式**：range() 函式用於生成一個數字序列，常用於指定迴圈執行次數。
* range(stop)：從 0 到 stop-1。
* range(start, stop)：從 start 到 stop-1。
* range(start, stop, step)：從 start 到 stop-1，步長為 step。
* **範例**：
* for i in range(5):
* print(i) # 輸出 0, 1, 2, 3, 4
* **迭代字串**：
* text = "Python"
* for char in text:
* print(char)
* **巢狀迴圈 (Nested Loops)**：一個迴圈內包含另一個迴圈，常用於處理二維數據結構（如矩陣或網格）。
* **範例**：印出九九乘法表，外層迴圈控制被乘數，內層迴圈控制乘數。
* for i in range(1, 10):
* for j in range(1, 10):
* print(f'{i} \* {j} = {i \* j}', end=' ')
* print()
* **搭配 len() 函式**：當需要基於數字索引迭代列表時，可以將 len() 函式與 range() 結合使用。
* **範例**：
* students = ['Hermione', 'Harry', 'Ron']
* for i in range(len(students)):
* print(f"{i + 1}. {students[i]}")

2.3 迴圈控制陳述式 (break 和 continue)

* **break**：強制立即終止當前迴圈的執行，程式流程會跳到迴圈後的第一行程式碼。
* **範例**：在 while True 迴圈中，當滿足特定條件時使用 break 跳出。
* while True:
* name = input('請輸入姓名:[輸入0,停止輸入]')
* if name == '0':
* break
* print(f"Hello! {name}")
* **continue**：跳過當前迴圈迭代中剩餘的程式碼，並進入下一次迭代。

3. 條件式 (Conditionals)

條件式允許程式根據特定條件的真假來執行不同的程式碼區塊，實現流程控制的「選擇」。

3.1 if-then-(else) 陳述式

* 基於布林條件（True 或 False）來決定執行哪段程式碼。
* **基本形式**：
* if condition:
* # code to execute if condition is True
* **if...else**：當條件為 True 時執行一段代碼，否則執行另一段。
* if a > 0:
* print("yes")
* else:
* print("no")
* **if...elif...else**：用於處理多個互斥條件。elif (else if) 允許檢查多個條件序列。
* **範例**：根據 BMI 值判斷體重狀態。
* if bmi < 18.5:
* state = "體重過輕"
* elif bmi < 24:
* state = "非常標準"
* # ... and so on
* else:
* state = "過重"
* **巢狀條件式**：條件陳述式可以嵌套在其他條件陳述式內部。

4. 函式 (Functions)

函式是組織程式碼的基本單位，它是一個可重複使用的小型程式碼區塊，執行特定的任務。

4.1 函式的定義與呼叫

* **定義函式**：使用 def 關鍵字。
* **語法**：def 函式名稱(參數1, 參數2, ...):
* **參數 (parameters)**：函式定義中聲明的值，它們是函式內部使用的佔位符。
* **呼叫函式**：使用函式名稱後跟括號，並傳入對應的引數。
* **語法**：函式名稱(引數1, 引數2)
* **引數 (arguments)**：呼叫函式時實際傳遞給參數的值。
* **回傳值 (return value)**：函式可以選擇性地使用 return 關鍵字回傳一個或多個值。如果沒有 return 語句，函式會隱式回傳 None。
* **範例**：
* def add\_numbers(a, b):
* return a + b
* result = add\_numbers(5, 3) # result will be 8

4.2 函式的優點

* **程式碼重複利用性 (Reusability)**：避免重複撰寫相同的程式碼，提高效率。
* **程式碼易讀性 (Readability)**：將複雜的任務分解為更小的、有意義的單元，使程式碼結構清晰。
* **程式碼易除錯性 (Debuggability)**：當問題發生時，更容易隔離和修復特定函式中的錯誤。
* **程式碼一致性 (Consistency)**：確保相同邏輯在不同地方被以相同方式應用。
* **程式碼模組化 (Modularity)**：將程式分解為獨立且分工明確的模組，每個函式負責特定的任務。

4.3 函式參數的類型

* **預設值參數**：在函式定義時為參數指定預設值。如果呼叫時不傳遞該參數，則使用預設值。
* **範例**：
* def fun(a, b=2, c=3):
* print(f"a={a}, b={b}, c={c}")
* fun(1) # a=1, b=2, c=3
* fun(1, 22, 33) # a=1, b=22, c=33
* **可變位置引數 (\*args)**：允許函式接受任意數量的位置引數，它們會被收集到一個元組中。
* **語法**：在參數前加上 \*。
* **範例**：
* def calculate\_sum(\*args):
* return sum(args)
* result = calculate\_sum(1, 2, 3, 4, 5) # result is 15
* **可變關鍵字引數 (\*\*kwargs)**：允許函式接受任意數量的關鍵字引數，它們會被收集到一個字典中。
* **語法**：在參數前加上 \*\*。
* **範例**：
* def example\_function(\*\*kwargs):
* for key, value in kwargs.items():
* print(f"{key}: {value}")
* example\_function(name="John", age=25) # name: John, age: 25
* \*args 和 \*\*kwargs 可以同時使用，\*args 必須在 \*\*kwargs 之前。

4.4 變數作用域 (Scope): 全域變數 vs. 區域變數

* **區域變數 (local variables)**：在函式內部定義的變數，只能在該函式內部訪問。函式執行結束後，區域變數就會被銷毀。
* **全域變數 (global variables)**：在函式外部（頂層）定義的變數，可以在程式的任何地方訪問。
* **在函式中修改全域變數**：如果想在函式內部修改全域變數的值，需要使用 global 關鍵字明確聲明該變數是全域的。
* **範例**：
* x1 = 10
* def myfun():
* x2 = 10 // 3
* global x1 # Declare x1 as global
* x1 = x2 + 10
* print(x1, x2)
* myfun() # Prints (13, 3)
* print(x1) # Prints 13 (global x1 has been modified)
* 不使用 global 關鍵字而直接在函式內對同名變數賦值，會創建一個新的區域變數，不會影響外部的全域變數。

4.5 匿名函式 (lambda functions)

* lambda 函式是「只有一行」的函式，通常用於處理小型、簡單的運算任務。
* 它不需要定義名稱，因此稱為「匿名函式」。
* **語法**：lambda 參數1, 參數2, ... : 運算式
* **特性**：
* 不需要 def 關鍵字。
* 只能包含一個運算式，不能包含多行陳述式或程式碼區塊。
* 自動回傳運算式的結果，不需要 return 關鍵字。
* **範例**：
* **基本使用**：
* add = lambda x, y: x + y
* print(add(1, 2)) # 3
* print(add("abc", "de")) # abcde
* **搭配 for 迴圈** (通常以列表推導式形式出現)：
* y = lambda n: [i for i in range(n)]
* print(y(5)) # [0, 1, 2, 3, 4]
* **搭配 if 判斷式** (作為條件運算子)：
* x = lambda n: True if n < 10 else False
* print(x(5)) # True
* print(x(15)) # False
* **與高階函式結合**：常用於 map(), filter(), sorted() 等函式，作為這些函式的 key 或轉換函數。
* a = [1, 2, 3, 4, 5]
* b = map(lambda x: x \* x, a)
* print(list(b)) # [1, 4, 9, 16, 25]

4.6 遞迴函式 (Recursive Functions)

* 遞迴函式是呼叫自身的函式。
* 常用於解決可以分解為更小、相同結構的子問題的問題，直到達到一個基本情況 (base case) 為止。
* **優點**：在某些情況下，遞迴可以使程式碼更簡潔、更接近問題的數學定義。
* **注意事項**：
* **必須有終止條件 (base case)**：否則會導致無限遞迴，最終引發 RecursionError。
* **效率問題**：在 Python 中，遞迴通常不如迭代（迴圈）效率高，因為每次函式呼叫都會增加呼叫堆疊 (call stack) 的開銷。Python 的遞迴深度也有預設限制。
* **資源消耗**：過深的遞迴可能導致堆疊溢出 (stack overflow)。
* **建議**：除非問題的性質特別適合遞迴且沒有明顯的迭代解法，否則通常建議優先使用 while 或 for 迴圈進行迭代。

5. 實用內建函式和概念

5.1 enumerate() 函式

* enumerate() 是 Python 的內建函式，用於在遍歷可迭代物件時，同時獲取元素的「索引」和「值」。
* 它返回一個「enumerate 物件」，在迭代時產生 (index, value) 元組。
* **語法**：enumerate(iterable, start=0)iterable：要迭代的可迭代物件 (例如列表、元組、字串、字典鍵)。
* start (可選)：索引起始值，預設為 0。
* **優點**：
* **簡潔性**：避免手動維護索引變數，使程式碼更清晰。
* **可讀性**：直接提供索引和值，使迭代過程更直觀。
* **減少錯誤**：降低手動管理索引可能引入的錯誤。
* **範例**：
* **遍歷列表**：
* fruits = ['apple', 'banana', 'cherry']
* for index, fruit in enumerate(fruits):
* print(f"{index}: {fruit}")
* # 輸出: 0: apple, 1: banana, 2: cherry
* **指定索引起始值**：
* for index, fruit in enumerate(fruits, start=1):
* print(f"{index}: {fruit}")
* # 輸出: 1: apple, 2: banana, 3: cherry
* **查找列表中某個值的所有索引**：
* numbers = [3, 5, 2, 3, 7, 3, 9]
* target = 3
* indices = []
* for index, num in enumerate(numbers):
* if num == target:
* indices.append(index)
* print(f"數字 {target} 出現在索引位置：{indices}") # 數字 3 出現在索引位置：[0, 3, 5]
* **注意事項**：在迭代過程中修改原始的可迭代物件可能導致意外行為。如需修改，建議迭代副本或使用索引進行修改。

5.2 列表 (Lists)

* 列表是 Python 中一種可變、有序的序列，可以儲存多個不同類型的元素。
* 元素可以透過索引（從 0 開始）訪問。支援負數索引，表示從列表末尾開始計算。
* **切片 (Slicing)**：用於獲取列表的一部分。
* a[:]：複製整個列表。
* a[:stop]：從開頭到 stop-1。
* a[start:]：從 start 到結尾。
* a[start:stop]：從 start 到 stop-1。
* 負數索引用於相對末尾的偏移：a[:-1]（除最後一個元素外所有元素）、a[-3:]（最後三個元素）。

5.3 字典 (Dictionaries)

* 字典是 Python 中一種鍵值對 (key-value pair) 的無序集合，類似於現實世界的字典。
* 每個鍵 (key) 必須是唯一的，並且用於快速檢索其對應的值 (value)。
* **定義**：使用 {} 大括號定義。
* **範例**：
* student = {
* "name": "Hermione",
* "house": "Gryffindor",
* "patronus": "Otter"
* }
* **訪問元素**：使用鍵來訪問值，例如 student["name"]。
* 字典可以嵌套，例如列表中的字典。
* **None 關鍵字**：表示值的缺失或空值。例如，如果某個學生沒有守護神，可以設定其 patronus 為 None。

6. 程式設計思維與最佳實踐

* **抽象化 (Abstraction)**：函式允許我們將複雜的邏輯包裝起來，只需關注函式「做什麼」，而不必關心其「如何做」的內部細節。這有助於簡化思考和設計更大型的程式。
* **範例**：print\_column(height) 函式抽象了打印一列磚塊的行為。
* **重構 (Refactoring)**：函式允許在不改變其外部行為的情況下，修改其內部實現細節。這對於性能優化、錯誤修復或代碼改進非常有用。
* **Python 版本**：Python 3 是當前主流版本，通常以 python3 命令運行。
* **Pythonic Thinking**：避免使用不直觀的語法，例如迴圈後的 else 塊。
* **命名慣例**：給變數和函式取有意義的名稱，提高程式碼的可讀性。例如，迭代列表中的學生時，將迭代變數命名為 student 而不是 \_ 或 i，除非你不需要使用該變數的值。
* **註解 (Comments)**：添加註解以解釋程式碼的意圖，尤其是在複雜的邏輯或巢狀結構中。

7. 問題解決流程 (Problem Solving)

* 將複雜問題分解為更小的、可管理的子問題。
* 例如，在 CS50P 的 Mario 問題中，將打印二維方塊的問題分解為打印行和打印列的子問題，然後再組合它們。
* 使用迴圈和條件式來實現動態和重複的行為。

表單的底部

NotebookLM 提供的資訊未必正確，請查證回覆內容。