Python 程式設計：異常處理

目錄

1. 異常 (Exception)

在程式執行期間，可能會發生各種錯誤，導致程式中斷。這類在「執行期間 (runtime)」發生的錯誤，在 Python 中被稱為異常 (Exception)。與之相對的是「語法錯誤 (Syntax Error)」，指的是程式碼不符合 Python 的語法規範，這種錯誤在程式執行前就會被解析器發現。

1.1 範例：一個未處理錯誤的例子

當異常發生但程式碼中沒有任何應對機制時，程式會立即停止執行，並在控制台印出一份稱為「追蹤訊息 (Traceback)」的錯誤報告。

```
def division(x, y):
    return x / y

division(10, 2)
division(5, 0)   # 此行將觸發異常
Traceback (most recent call last):
#   File "D:\github\python-tutor\test.py", line 5, in <module>
#     print(division(5, 0))   # 此行將觸發異常
#           ^^^^^^^^^^^^^
#   File "D:\github\python-tutor\test.py", line 2, in division
#     return x / y
```

```
#            ~~^~~
# ZeroDivisionError: division by zero
# (base) PS D:\github\python-tutor>
```

2. try...except 結構

為了捕捉並處理異常，我們使用 try...except 這個語法結構。它的核心思想是：

- try 區塊：將你預期可能發生異常的程式碼包裹起來。
- except 區塊：如果 try 區塊中的程式碼確實發生了異常，程式會立即跳到 except 區塊來執行。如果 try 區塊沒有發生異常，則 except 區塊會被完全跳過。

```
def division(x, y):
    try: # 嘗試執行可能會出錯的程式碼
        return x / y
    except ZeroDivisionError: # 如果發生了 ZeroDivisionError，執行此處
        print("除數不可為 0")
        # return None  # 可以選擇回傳一個特殊值，表示計算失敗

division(10, 2)
division(5, 0)
```

3. try...except...else

有時我們希望在 try 區塊沒有發生任何異常的情況下，執行某些特定的程式碼。這時就可以使用 else 區塊。

else 區塊：僅在 try 區塊成功執行完畢（未觸發任何異常）時才會執行。

```
def division(x, y):
    try:
        ans = x / y
    except ZeroDivisionError:
        print("異常發生：除數為 0")
        return # 發生異常時明確返回 None
    else: # 只有在 try 區塊沒有異常時，才會執行這裡
        print(f"計算結果: {ans}")
        return
division(10, 2)
division(5, 0)
```

4. 處理多個異常

一個 try 區塊內可能發生多種類型的異常。我們有幾種策略來應對。

4.1 策略一：使用多個 except 區塊

```
def division(x, y):
    try:
```

```
        return x / y
    except ZeroDivisionError:
        print("異常發生：除數為 0")
    except TypeError:
        print("異常發生：使用了不正確的資料型態進行運算")

division(10, 2)
division(5, 0) # 觸發的 ZeroDivisionError 被第一個 except 捕捉。
division('a', 'b') # 觸發的 TypeError 被第二個 except 捕捉。
division(6, 3)
```

4.2 策略二：在單一 except 區塊中處理多個異常

如果多種異常的處理邏輯完全相同，你可以將它們放在一個元組 (tuple) 中。

```
def division(x, y):
    try:
        return x / y
    except (ZeroDivisionError, TypeError): # 兩種錯誤都用同樣的方式處理
        print("異常發生：除數為 0 或使用了不正確的資料型態")
```

4.3 策略三：捕捉通用異常

Exception 是一個基礎的異常類別，大多數內建的執行期異常都繼承自它。因此，except Exception 可以捕捉幾乎所有類型的異常。

```
def division(x, y):
    try:
        return x / y
    except Exception: # 捕捉所有繼承自 Exception 的異常
        print("通用錯誤處理：發生了某種異常")
```

5. 取得異常的內建訊息

有時候，我們不僅想知道發生了哪一種類型的錯誤，還想獲得 Python 提供的具體錯誤訊息。我們可以使用 as 關鍵字將異常實例儲存到一個變數中。

```
def division(x, y):
    try:
        return x / y
    except (ZeroDivisionError, TypeError) as e: # # 將異常物件賦值給變數 e
        print(f"異常發生，內建訊息: {e}")

division(5, 0)
division('a', 'b')
```

6. 手動丟出異常：raise

除了處理系統自動觸發的異常，我們也可以在程式中根據特定邏輯條件，主動觸發一個異常。這使用 raise 關鍵字來完成。

```
def check_password(pwd):
    if len(pwd) < 5:
        # 如果密碼長度不足，主動拋出一個異常
        raise Exception("密碼長度不足")
    if len(pwd) > 8:
        raise Exception("密碼長度太長")
    print("密碼長度正確")

try:
    check_password("aaa")  # 長度不足
except Exception as e:
    print(f"錯誤: {e}")

try:
    check_password("aaabbbccc") # 長度太長
except Exception as e:
    print(f"錯誤: {e}")

try:
    check_password("goodpwd") # 長度正確
except Exception as e:
    print(f"錯誤: {e}")
```

7. finally

在異常處理的語法中，finally 區塊扮演著一個特殊的角色：無論 try 區塊是否發生異常，finally 區塊中的程式碼都保證會被執行。它的主要用途是資源清理，例如：

- 關閉一個已開啟的檔案。
- 釋放一個網路連線。
- 還原某個臨時設定。

```
def division(x, y):
    try:
        return x / y
    except:
        print("異常發生")
    finally: # 無論是否發生異常，此處程式碼都會執行
        print("階段任務完成")
```

```
division(10, 2)
print("-" * 20)
division(5, 0)
```

8. 程式斷言 assert

在編寫程式時，除了會導致程式崩潰的「執行期異常」外，還有一種更隱蔽的錯誤，稱為「邏輯錯誤」。這種類型的錯誤不會讓程式中斷，但會產生不正確或不符合預期的結果。

8.1 範例：銀行帳戶的例子

```
class Banks():
    title = 'Taipei Bank'

    def __init__(self, uname, money):
        self.name = uname
        self.balance = money

    def save_money(self, money):
        self.balance += money
        print("存款", money, "完成")

    def withdraw_money(self, money):
        self.balance -= money
        print("提款", money, "完成")

    def get_balance(self):
        print(self.name, "目前餘額:", self.balance)


# --- 程式執行 ---
hungbank = Banks('Hung', 100)
hungbank.get_balance()
hungbank.save_money(500)       # 存款 500
hungbank.get_balance()
hungbank.withdraw_money(700)  # 提款 700 => 程式沒有報錯，順利地執行完畢，但最終餘額變成了 -100。
hungbank.get_balance()
```

從銀行業務的邏輯來看，餘額不應該是負數，而且也不應該能提出比帳戶餘額更多的錢。這就是一個典型的邏輯錯誤。程式本身沒有崩潰，但其行為與我們的設計初衷相違背。因此，我們可以在在程式中設定斷言(assert)。

assert <條件 (condition)>, <錯誤訊息 (error message)>

* <條件>：一個布林表達式 (boolean expression)，其結果為 **True** 或 **False**
* <錯誤訊息>：一個可選的字串。當條件為 **False** 時，這個字串會作為 **AssertionError** 的一部分顯示出來。

```
# ch15_20.py
class Banks():
    title = 'Taipei Bank'

    def __init__(self, uname, money):
        self.name = uname
        self.balance = money

    def save_money(self, money):
        # 斷言：存款金額必須大於 0
        assert money > 0, "存款金額必須為正數"
        self.balance += money
        print("存款", money, "完成")

    def withdraw_money(self, money):
        # 斷言：提款金額必須小於或等於目前餘額
        assert money <= self.balance, "存款餘額不足，無法提款"
        self.balance -= money
        print("提款", money, "完成")

    def get_balance(self):
        print(self.name.title(), "目前餘額:", self.balance)

# --- 程式執行 ---
hungbank = Banks('Hung', 100)
hungbank.get_balance()
hungbank.save_money(400)      # 存款 400 (教材此處為 500，為與圖片結果一致改為 400)
hungbank.get_balance()
hungbank.withdraw_money(700)  # 試圖提款 700
hungbank.get_balance()
```

9. assert vs. try...except

理解何時使用 assert 和何時使用 try...except 是至關重要的。

特性	assert (斷言)	try...except (異常處理)
目的	**捕捉開發時的程式邏輯錯誤 (Bugs)**。用於檢查那些「理論上絕不應該發生」的情況。	**處理執行期間可預期的錯誤**。例如使用者輸入錯誤、檔案不存在、網路中斷等。

對象	主要面向**程式開發者**。斷言失敗意味著程式碼有問題，需要修復。	主要面向**程式使用者或外部環境**。捕捉異常是為了讓程式更健壯，能優雅地處理外部問題。
可否停用	**可以**。在生產環境中可使用 -O 選項全域停用以提升效能。	**不可以**。異常處理是程式核心邏輯的一部分，不能被停用。
範例	檢查函式內部狀態、驗證傳入的參數是否符合內部邏輯。assert money > 0	處理使用者輸入、讀取檔案、進行網路請求。except FileNotFoundError: