§7 ОБРОБКА ВИКЛЮЧЕНЬ

Помилки, виключні ситуації та виключення

Помилки у програмах

Розглянемо дуже важливий розділ сучасного програмування – обробку помилок.

Існує три типи помилок, що виникають під час виконання програми:

- **синтаксичні** – це помилки, що породжуються неправильним використанням синтаксичних конструкцій мови програмування. Наприклад, такий виклик оператора `print`

```
print("Повідомлення!" # Не вистачає дужки
```

породить таку помилку

```
  File "exception.py", line 4

    ^
SyntaxError: unexpected EOF while parsing
```

- **помилки виконання** (**runtime error**) – це помилки, що виникають у синтаксично правильних програмах під час їхнього виконання, коли інтерпретатор не може коректно розв'язати проблему, що виникла або подальше виконання програми є неможливим. Класичним прикладом помилки виконання є ділення на нуль:

```
>>> div_by_zero = 1 / 0
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ZeroDivisionError: division by zero
```

Як бачимо, у цьому випадку інтерпретатор повідомив нас про те, що у програмі відбувається ділення на нуль.

- **логічні помилки** – це помилки пов'язані з логікою програми. Інтерпретатор не зможе виявити і ідентифікувати такі помилки, оскільки з точки зору синтаксису програма написана правильно. Їх можна виявити лише за результатами роботи програми.

Існує ціла сфера діяльності – тестування програмного забезпечення (eng: QA (Quality Assurance)) – пов'язана з виявленням помилок останнього типу. Цей же параграф присвячений обробці помилок перших двох типів.

Виключні ситуації та виключення

Якщо у програмі виникає помилка першого або другого типу, то будемо казати, що відбулася **виключна ситуація**. Будь-яка виключна ситуація **генерує** (також використовується термін **породжує**) **виключення**.

Виключення (eng: **exception**) – спеціальний тип даних, що використовується для ідентифікації та детального опису помилки, що виникла у програмі.

Слід зауважити, що виключення породжуються не лише у разі виникнення виключної ситуації, а і як повідомлення про те, що відбулася певна подія. Наприклад, метод списку `pop()` генерує виключення `IndexError`, якщо список порожній.

Обробка виключень

Для обробки виключень використовується оператор **`try`**. Його загальний синтаксис такий

```
try:
    processing_code_with_potential_exception
except exception_1
    handling_exception_1
...
except exception_N
    handling_exception_N
else:
    state_else
finally:
    obligatory_code
```

Виконання цього оператора відбувається таким чином:

1. Інтерпретатор намагається виконати блок коду з потенційним виключенням (`processing_code_with_potential_exception`).
2. Якщо у випадку виконання коду породжується виключення, що належить до класу виключень `exception_1` то у точці генерування виключення відбувається безумовний перехід на блок обробки виключення `handling_exception_1`. Аналогічне стосується інших типів виключення, що перехоплюються у програмі.

3. Якщо, породжене виключення не обробляється у жодному з блоків обробки виключень, то інтерпретатор припиняє роботу програми (за допомогою типового обробника виключень)
4. Якщо виконання блоку **try** повністю відбулося без виключень, то відбувається виконання коду `state_else` блоку **else**. Якщо ж було згенеровано хоча б одне виключення цей блок не виконується.
5. Код `obligatory_code` блоку **finaly** виконується завжди після виконання попередніх кроків незалежно від того, чи було породжено виключення чи ні. Наприклад, без нього не можливо обійтися, якщо необхідно закрити відкритий (до породження виключення) файл, звільнити з'єднання з базою даних тощо.

Слід зауважити, що обов'язковими блоками оператора **try** є лише блок **try** і один блок **except** обробки виключень.

```
try:
    print(1 / 0)           # Блок з потенційним діленням на 0
except ZeroDivisionError: # Обробка виключення ділення на 0
    print("Ділення на 0")
```

або блок **try** і блок **finally**.

```
f = open("myfile.txt")    # Відкриття файлу
try:
    f.read()              # Читання з файлу
finally:
    f.close()             # Закриття файлу
```

Якщо у інструкції **except** опустити клас виключення, то такий блок буде перехоплювати усі виключення. На практиці не рекомендується використовувати такі блоки обробки виключень, оскільки є небезпека перехопити виключення, що є повідомлення системі, а не помилкою або виключення, після яких подальша робота програми не має сенсу.

Приклад 7.1. Задано послідовність дійсних невід'ємних чисел. Знайдемо максимум відношень між сусідніми членами цієї послідовності.

Розв'язок. Очевидно, що якщо деякий член послідовності буде дорівнювати нулю, то в момент знаходження відношення буде генеруватися виключення «ділення на нуль». Очевидно, що просто ігнорувати такі члени послідовності не можна, оскільки результат роботи програми у такому разі буде не об'єктивним.

```
# Введення послідовності
N = int(input("N = "))
```

```
l = []
for i in range(N):
    l.append(float(input("l[%d] = " % i)))
# Знаходження максимуму відношення
try:
    max_ratio = 0
    for i in range(1, len(l)):
        ratio = l[i] / l[i-1]
        if ratio > max_ratio:
            max_ratio = ratio
except ZeroDivisionError:
    print("У послідовності є нульові елементи")
else:
    print("Максимум відношень = %f" % max_ratio)
```

Звернемо увагу на те, що у нашій програмі оператор **try**, у випадку, якщо породжується виключення «ділення на нуль», передає керування блоку **except** тим самим перериваючи роботу циклу **for**, подібно до того, як це здійснює оператор **break**.

Приклад 7.2. Обчислимо суму послідовності цілих чисел, що вводяться користувачем.

Розв'язок. Для введення цілих або дійсних чисел з клавіатури ми користувалися інструкцією `input`, з подальшим перетворенням введеного результату до цілого чи дійсного типу за допомогою інструкцій `int` або `float` відповідно. При цьому, раніше ми вважали, що користувач дисциплінований і вводить з клавіатури лише коректні дані. Проте, якби користувач ввів рядок, який не можливо було б перетворити до відповідного типу, породжувалося виключення `ValueError`.

Напишемо програму, що буде контролювати правильність введення даних. Завершувати підрахунок будемо у випадку, коли користувач введе з клавіатури слово “стоп”.

```
print("Введіть 'стоп' для отримання результату")
suma = 0
while True:
    x = input("Введіть число: ")
    if x == "стоп":
        break  # Завершення підрахунку
    try:
        x = int(x) # Перетворюємо рядок у число
    except ValueError:
        print("Потрібно задати ціле число!")
    else:
```

```
        suma += x
print("Сума = ", suma)
```

Процес роботи програми буде виглядати так

```
Введіть 'стоп' для отримання результату
Введіть число: 1
Введіть число: 2
Введіть число: щось
Потрібно задати ціле число!
Введіть число: 3
Введіть число: стоп
Сума =  6
```

Стандартні класи виключень

Набір вбудованих виключень у мові Python організований у вигляді ієрархії, що наведена нижче.

```
BaseException
      SystemExit
      KeyboardInterrupt
      GeneratorExit
      Exception
            ArithmeticError
                  FloatingPointError, OverflowError,
                  ZeroDivisionError
            AssertionError
            AttributeError
            BufferError
            EOFError
            ImportError
            LookupError
                  IndexError, KeyError
            MemoryError
            NameError
                  UnboundLocalError
            OSError
                  ConnectionError, FileExistsError,
                  FileNotFoundError, InterruptedError,
                  IsADirectoryError, NotADirectoryError,
                  PermissionError, ProcessLookupError,
                  TimeoutError
            ReferenceError
```

```
            RuntimeError
            NotImplementedError
            StopIteration
            SyntaxError
            SystemError
            TypeError
            ValueError
            Warning
```

Якщо читач вже знайомий з парадигмою об'єктно-орієнтованого програмування, то вищенаведена ієрархія може йому нагадати ієрархію класів успадкованих від базового класу `BaseException`. Саме таким чином в Python і організована робота з виключеннями – кожний тип виключення це клас успадкований від іншого типу виключення.

Перевага використання ієрархії для обробки виключень полягає у тому, що є можливість задавання базового класу виключень для перехоплення всіх дочірніх виключень. Так наприклад, у попередніх прикладах для перехоплення виключення «ділення на нуль» використовувалося виключення `ZeroDivisionError`. Якщо ж замість нього вказати базове виключення `ArithmeticError`, то будуть перехоплюватися всі типи виключень, що є нащадками до цього класу: `FloatingPointError`, `OverflowError`, `ZeroDivisionError`.

Нижче наведена таблиця та опис виключень, які будуть зустрічатися у цьому курсі. З описом інших виключень можете ознайомитися за допомогою додаткової літератури, наприклад офіційної документації [1].

Таблиця 7.1. Вбудовані виключення.

Виключення	Опис
`BaseException`	Базовий клас виключень.
`Exception`	Основний клас системних виключень. Всі власні виключення повинні успадковуватися від цього класу.
`ArithmeticError`	Арифметична помилка
`AssertionError`	Помилка твердження – породжується, якщо умова твердження є хибною.
`EOFError`	Породжується функцією зчитування з файлу, якщо досягнуто кінця файлу.
`ImportError`	Помилка імпорту модуля чи пакету
`LookupError`	Помилка індексу або ключа при зверненні до елементів колекції.
`NameError`	Не знайдено змінної з таким ім'ям.

Виключення	Опис
`OSError`	Клас виключень, пов'язаних з операційною системою.
`RuntimeError`	Породжується, коли виключення не потрапляє у жодну з категорій.
`StopIteration`	Породжується вбудованою функцією `next`, якщо в колекції вже перебрані послідовно всі елементи.
`SyntaxError`	Синтаксична помилка
`SystemError`	Внутрішня системна помилка
`TypeError`	Помилка типу – породжується якщо операція застосовується до об'єкта невідповідного типу
`ValueError`	Помилка значення – породжується якщо інструкція отримує аргумент правильного типу, але некоректного значення.

Створення екземпляру виключення

Щоб у програмі проаналізувати виключення детальніше, у блоці **`except`** вказують не лише тип виключення, але і створюють екземпляр цього виключення, що містить детальну інформацію про виключення. Для цього, у блоці **`except`** після типу виключення вказують ім'я екземпляру виключення

```
except exception as e:
```

тут e – ім'я екземпляру виключення.

Як було зауважено раніше, типи виключень це класи. Тоді екземпляри виключень це об'єкти, що належать до відповідних класів. Оскільки передбачається, що наразі читач ще не знайомий з об'єктно-орієнтованим програмуванням, то не будемо перевантажувати читача правилами роботи з екземплярами виключень, як з об'єктами класів. Єдине що важливо буде для нас, так це те, що екземпляр виключення можна вивести на екран з допомогою інструкції `print`.

```
try:
    print(1 / 0)
except ZeroDivisionError as e:    # Створюємо екземпляр e
    print(e)    # Виводимо екземпляр виключення e на екран
```

Результатом виконання цього блоку коду буде виведення на екран повідомлення

```
division by zero
```

Перевірка твердження

Щоб перевірити стан програми (наприклад для того, щоб переконатися, що подальше її виконання має сенс) використовують інструкцію **assert**. Вона має такий синтаксис

```
assert condition, message
```

де `condition` – умова (логічний вираз), `message` – необов'язковий параметр, що є даними, що передаються на обробку при виникненні виключної ситуації.

Інструкція **assert** генерує виключення типу `AssertionError`, якщо логічний вираз `condition` є хибним. Наприклад,

```
x = 0
try:
    assert x != 0
except AssertionError:
    print("Ділити на 0 не можна")
else:
    print(1 / x)
```

Ініціювання виключення

Програміст може самостійно ініціювати виключення будь-якого типу. Основне призначення цього механізму – це застосування механізмів обробки виключних ситуацій для створених користувацьких виключень.

Синтаксис ініціювання виключення такий

```
raise exception
```

тут `exception` – тип виключення (або екземпляр класу виключення).

При виконанні цієї інструкції Python діє аналогічно тому, як він діє при штатному виникненні виключення цього типу. Наприклад, у наведеному нижче блоці коду, програма ініціює виключення `ZeroDivisionError`, якщо змінна `x` (тобто другий операнд у операції ділення) дорівнює нулю.

```
x = 0
try:
    if x == 0:
        raise ZeroDivisionError     # Ініціювання виключення
except ZeroDivisionError: # Обробка виключення ділення на 0
    print("Ділення на 0")
else:          # Якщо виключення не відбулося виводимо 1 / x
```

```
    print(1 / x)
```

Менеджер контексту

Менеджери контексту дозволяють захопити та звільняти ресурси програми строго за необхідності, не залежно від того, що відбулося з програмою у ході її виконання.

Менеджер контексту реалізується за допомогою оператора **with**.

```
with expression as target:
    do_something
```

тут `expression` операція, що захоплює ресурси (наприклад відкриває файл, блокує ресурси тощо), `target` – об'єкт, що створюється в результаті операції `expression`.

Розглянемо правило за яким використовується менеджер контексту: нехай маємо дві пов'язаних операції, що мають бути виконані у парі (наприклад, відкриття і закриття файлу) і є код, що потрібно розмітити між ними (наприклад, обробка файлу)

```
захопити_ресурси f
обробити_дані f
звільнити_ресурси f
```

Тоді цей блок можна замінити оператором менеджера контексту

```
with захопити_ресурси as f:
    обробити_дані f
```

Отже, така операція з застосуванням менеджера контексту еквівалентна такому блоку з використанням оператора **try**:

```
захопити_ресурси f
try:
    обробити_дані f
finally:
    звільнити_ресурси f
```

і фактично є синтаксичною конструкцією, що дозволяє скоротити код програми і гарантує безпечне використання даних.