# Лекція 28

## Обробка виключень

# Означення виключення

**Виключення** – це повідомлення інтерпретатора, які виникають у випадку виникнення помилки в програмному коді або при настанні якої-небудь події.

Якщо в коді не передбачена обробка виключення, то виконання програми переривається, і виводиться повідомлення про помилку.

```
a = (1,2,3,4,5)
a[0] = 10
```

Результат роботи:

```
  Traceback (most recent call last):
    File "C:/PYTHON/proba.py", line 2, in
<module>
      a[0] = 10
  TypeError: 'tuple' object does not support
item assignment
```

## Синтаксичні помилки

Синтаксичні помилки – це помилки в імені оператора або функції, тобто помилки в синтаксисі мови. Як правило, інтерпретатор попередить про наявність помилки, а програма не буде виконуватися зовсім.

Приклад синтаксичної помилки:

```
print("Немає закриваючих лапок!)
  File "C:/PYTHON/proba.py", line 1
    print("Немає закриваючих лапок!)
                                     ^
  SyntaxError: EOL while scanning string literal
```

# Логічні помилки

Логічні помилки – це помилки в логіці програми, які можна виявити тільки за результатами її роботи.

Завдання. Знайти перетин множин X та Y

```
X={1,2,3,4,5}
Y={4,5,6,7,8}
Z=X | Y
print("Z = ",Z)
```

Результат:

```
Z =  {1, 2, 3, 4, 5, 6, 7, 8}
```

Відповідь неправильна, оскільки в програмі існує логічна помилка: неправильно записана логічна операція

Потрібно було записати &, а записано |

## Помилки часу виконання

Помилки часу виконання – це помилки, які виникають під час роботи програми. Причиною є події, які не передбачені програмістом.

Класичним прикладом служить ділення на нуль:

**Приклад 1.**

```
def test(x, y) :
    return x / y
print("Нормальне виконання",test(4, 2) )
print("Помилка!",test(4, 0)) # Помилка
Результат виконання:
Нормальне виконання 2.0
.....
Zerodivisionerror: division by zero
```

## Виключення при настанні події

В мові Python виключення виконуються не тільки при помилці, але і як повідомлення про настання яких-небудь подій. Наприклад, метод `index()` виконує виключення `Valueerror`, якщо шуканий фрагмент не входить у рядок:

**Приклад 2.**

```
>>> "Рядок".index("текст")

Traceback (most recent call last):
  File "<input>", line 1, in <module>
Valueerror: substring not found
```

# Інструкція *try...except...else...finally*

Для обробки виключень призначена інструкція `try`.

Формат інструкції:

**try**:
<Блок, у якому перехоплюються виключення>
[**except** [<Виключення1>[ **as** <Об'єкт виключення>]]:
<Блок, виконуваний при виникненні виключення 1>
**except** [<Виключення2>[ **as** <Об'єкт виключення2>]]:
<Блок, виконуваний при виникненні виключення2>]]
[**else**:
<Блок, виконуваний, якщо виключення не виникло>]
[**finally**:
<Блок, виконуваний у будь-якому випадку>]

## Найпростіший варіант інструкції `try...except`

Інструкції, у яких перехоплюються виключення, повинні бути розташовані усередині блоку **`try`**.

У блоці **`except`** у параметрі <Виключення1> вказують клас оброблюваного виключення.

Наприклад, обробити виключення, що виникає при діленні на нуль, можна так, як показано в прикладі.

**Приклад 3.**

```
try:     # Перехоплюємо виключення
     x = 1/0     # Помилка: ділення на 0
except ZeroDivisionError:# Указуємо клас виключення
     print("Опрацювали ділення на 0")
     x = 0
print(x)
```

```
Результат виконання :
Опрацювали ділення на 0
0
```

## Інструкція `try`…`except` у загальному випадку

Якщо в блоці `try` виникло виключення, то керування передається блоку `except`.

Якщо виключення не відповідає зазначеному класу, керування передається наступному блоку `except`.

Якщо жоден блок `except` не відповідає виключенню, то виключення «спливе» до обробника більш високого рівня.

Якщо виключення в програмі взагалі ніде не обробляється, воно передається обробнику за замовчуванням, який зупиняє виконання програми й виводить стандартну інформацію про помилку.

**Обробники `try`…`except`**
**можуть бути вкладеними**

## Приклад 4. Вкладені обробники

```
try:     # Опрацьовуємо виключення
     try:  # Вкладений обробник
          x = 1 / 0  # Помилка: ділення на 0
     except NameError:
          print("Невизначений ідентифікатор")
     except IndexError:
          print("Неіснуючий індекс")
     print("Вираз після вкладеного обробника")
except ZeroDivisionError:
     print("Опрацювання ділення на 0")
     x = 0
print(x)  # виведе: 0
Результат виконання :
Опрацювання ділення на 0
0
```

## Пояснення

1.Вкладений try не спрацював.
Причина цього – у ньому не зазначене виключення **`ZeroDivisionError`**

2.Виключення «спливає» до обробника більш високого рівня.

3.Після обробки виключення керування передається інструкції, розташованій відразу після обробника: **`print(x)`**.

4. Інструкція
`print("Вираз після вкладеного обробника")`
виконана не буде.

## Кілька виключень

В інструкції **except** можна вказати відразу кілька виключень, перелічивши класи через кому усередині круглих дужок.

**Приклад 5.**

```
try:
    x = 1 / 0
except (NameError, IndexError,
ZeroDivisionError):
    # Обробка відразу декількох виключень
    x = 0
print(x) # Виведе: 0

Результат виконання : 0
```

## Інформація про оброблене виключення

Одержати інформацію про оброблюване виключення можна через другий параметр в інструкції **except**.

**Приклад 6.**

```
try:
    x = 1/0
except (NameError, IndexError,
ZeroDivisionError) as err:
    print(err.__class__.__name__)
    print(err)
```

Результат виконання:

```
ZeroDivisionError
division by zero
```

## Функція `exc_info()`

### Одержання інформації про виключення

Функція знаходиться в модулі **`sys`** та повертає кортеж із трьох елементів:

1. тип виключення,
2. значення
3. об'єкт з трасувальною інформацією.

Перетворити ці значення в зручний для читання вигляд дозволяє модуль `traceback`.

Приклад ви користання функції **`exc_info()`** і модуля **`traceback`** наведений у прикладі.

## Приклад 7.

```
import sys, traceback
try:
    x = 1 / 0
except ZeroDivisionError:
    Type, Value, Trace = sys.exc_info()
    print ("Type: ", Type)
    print("Value:", Value)
    print("Trace:", Trace)

print("\n", "print exception()".center(40, "-"))
traceback.print_exception(Type, Value, Trace, limit=5, file=sys.stdout)
print("\n", "print_tb()".center(40, "-" ))
traceback.print_tb(Trace, limit=1, file=sys.stdout)
print("\n", "format_exception()". center ( 40, "-") )
print(traceback.format_exception(Type, Value, Trace, limit=5))
print("\n", "format_exception_only()".center(40,
"-"))
print(traceback.format_exception_only(Type,
Value))
```

```
Результат виконання прикладу:
Type:  <class 'ZeroDivisionError'>
Value: division by zero
Trace: <traceback object at 0x002F7CD8>

 -----------print exception()-----------
Traceback (most recent call last):
  File "C:/PYTHON/first.py", line 3, in
<module>
    x = 1 / 0
ZeroDivisionError: division by zero

 ---------------print_tb()--------------
  File "C:/PYTHON/first.py", line 3, in
<module>
    x = 1 / 0
```

```
----------format_exception()-----------
['Traceback (most recent call last):\n',
'  File "C:/PYTHON/first.py", line 3, in
<module>\n                x   =   1   /   0\n',
'ZeroDivisionError: division by zero\n']

--------format_exception_only()--------
['ZeroDivisionError:     division     by
zero\n']
```

Якщо в інструкції `except` не зазначений клас виключення, то такий блок перехоплює всі виключення. На практиці слід уникати порожніх інструкцій `except`, тому можна перехопити виключення, яке є лише сигналом системі, а не помилкою.

Приклад порожньої інструкції `except`

**Приклад 8.**

```
try:
    x = 1/0  # Помилка ділення на 0
except:  # Обробка всіх виключень
    x = 0
print(x)
```

```
Результат виконання: 0
```

Блок `else` – інструкції усередині цього блоку будуть виконані тільки при відсутності помилок.

Блоко `finally` – виконуються завершальні дії незалежно від того, чи виникло виключення, чи ні.

**Приклад 9. Застосування `else` і `finally`**

```
a=int(input("Введіть дільник: "))
try:
    x = 10 / a    # Помилка ділення на 0
except ZeroDivisionError:
    print ( "Ділення на 0")
else:
    print("Блок else")
finally:
    print("Блок finally")
```

```
Результат виконання при відсутності виключення:
Блок else
Блок finally
-------------------------
Ділення на 0
Блок finally
```

## Інструкція try…finally
(без блоку except)

Інструкції усередині блоку **`finally`** будуть виконані, але виключення не буде оброблено.

Воно продовжить «спливання» до обробника більш високого рівня.

Якщо обробник користувача відсутній, то керування передається обробнику за замовчуванням, який перериває виконання програми й виводить повідомлення про помилку.

**Приклад 10.**

```
try:
    X= int(input("Введіть число:"))
finally: print("Блок finally")
```

Результат виконання:

```
Введіть число: Ж
Traceback (most recent call last):
  File "C:/PYTHON/proba.py", line 2, in
<module>
    X= int(input("Введіть число:"))
ValueError: invalid literal for int()
with base 10: ' Ж'
Блок finally
```

# Ввід рядка замість числа

**Приклад 11.**

```
print("Введіть слово 'stop' для отримання результату")
suma = 0
while True:
    x = input("Введіть число: ")
    if x == "stop":
        break # Вихід з циклу
    try:
        x = int(x) # Перетворюємо рядок на число
    except ValueError:
        print("Необхідно ввести ціле число!")
    else:
        suma += x
print("Сума чисел дорівнює:", suma)
```

Процес введення значень і одержання результату має такий вигляд (значення, введені користувачем, виділені напівжирним шрифтом):

```
Введіть слово 'stop' для отримання результату
Введіть число: 10
Введіть число: str
Необхідно ввести ціле число!
Введіть число: -5
Введіть число:
Необхідно ввести ціле число!
Введіть число: stop
Сума чисел дорівнює: 5
```

# Інструкція *with...as*

*Інструкція призначена для обгортання блоку інструкцій менеджером контексту*

Формат інструкції:

```
with <Вираз1>[ as <Змінна>][,...,
     <Выразn> [as <Змінна>]]:
     <Блок перехопення виключення>
```

Послідовність дій:

1.Обчислюється вираз < Вираз1>.

2. Виконується метод класу__enter__

3. Якщо конструкція **with** включає слово **as**, то значення, яке повертає метод __enter__, записується в змінну.

4. Виконується <Блок перехопення виключення>

5. Викликається метод класу __exit__. У цей метод передаються параметри виключення, якщо воно сталося, або у всіх аргументах значення None, якщо виключення не було.

# Метод __exit__

Формат методу:

```
__exit__(self, <Тип виключення>,
<Значення>, <Об'єкт traceback>)
```

Значення, доступні через останні три параметри, еквівалентні значенням, що повертаються функцією **exc_info ()** з модуля **sys**.

Якщо виключення оброблене, метод повинен повернути значення **True**, а якщо ні, то – **False**.

Якщо метод повертає **False**, то виключення передається вищому обробнику.

Розглянемо послідовність виконання протоколу на прикладі.

## Приклад 12. Протокол менеджерів контексту

```
class MyClass:
    def __enter__(self):
        print("Викликано метод __enter_()")
        return self
    def __exit__ (self, Type, Value, Trace):
        print ("Викликано метод _exit_() ")
        if Type is None:# Якщо виключення не виникло
            print("Виключення не виникло")
        else: # Якщо виникло виключення
            print("Value =", Value)
            return False
print("Послідовність при відсутності виключення:")
with MyClass()as obj:
    print("Блок всередині with")
```

```
Послідовність при відсутності виключення:
Викликано метод __enter_()
Блок всередині with
Викликано метод _exit_()
Виключення не виникло
```

```
class MyClass:
    def __enter__(self):
        print("Викликано метод __enter_()")
        return self
    def __exit__ (self, Type, Value, Trace):
        print ("Викликано метод _exit_() ")
        if Type is None:# Якщо виключення не виникло
            print("Виключення не виникло")
        else: # Якщо виникло виключення
            print("Value =", Value)
            return True
print("\nПослідовність за наявності виключення:")
with MyClass() as obj:
    print("Блок всередині with")
    raise TypeError("Виключення TypeError")
Послідовність за наявності виключення:
Викликано метод __enter_()
Блок всередині with
Викликано метод _exit_()
Value = Виключення TypeError
```

# Підтримка `with...as` за замовчуванням

Деякі вбудовані об'єкти підтримують протокол за замовчуванням – наприклад, файли.

Якщо в інструкції **`with`** зазначена функція **`open(),`** то після виконання інструкцій усередині блоку файл автоматично буде закритий. Приклад використання інструкції **`with:`**

**Приклад 13.**

```
with open ("test. txt", "a", encoding="utf-8") as f:
    f.write("Рядок\n") #записуємо рядок у кінець файлу
```

У цьому прикладі файл **`test.txt`** відкривається на дозапис у кінець файлу.

Після виконання функції **`open()`** змінній **`f`** буде присвоєно посилання на об'єкт файлу.

За допомогою цієї змінної ми можемо працювати з файлом всередині тіла інструкції **`with.`**

Після виходу із блоку незалежно від наявності виключення файл буде закритий.

## Виключення користувача

Для виконання виключень користувача призначено дві інструкції: **`raise`** і **`assert`**.

Інструкція `raise` виконує задане виключення. Вона має кілька варіантів формату:

**`raise`**<Екземпляр класу> (або вбудоване виключ.)

**`raise`** <Назва класу>

**`raise`** <Екземп. або назва класу> from <Об'єкт виключ>

**`raise`**

У першому варіанті формату інструкції `raise` вказується екземпляр класу в якому виникло виключення.

При створенні екземпляра можна передати дані конструктору класу.

Ці дані будуть доступні через другий параметр в інструкції **`except`**.

## raise(екземпляр класу користувача)

**Приклад 14.**

```
class MyError(Exception):
    def __init__(self, value):
        self.msg = value
    def __str__(self):
        return self.msg
# Опрацювання виключення користувача
try:
   err= MyError("Опис виключення1")
   raise err
except MyError as err:
    print(err) #Викликаємо метод __str__ ()
    print(err.msg) #Доступ до атрибуту класу
```

Опис виключення1
Опис виключення1

## Клас `Exception`

Клас **`Exception`** містить усі необхідні методи для виводу повідомлення про помилку. Тому в більшості випадків достатньо створити порожній клас, який успадковує клас **`Exception`**:

**Приклад 15.**

```
class MyError(Exception): pass
try:
    raise MyError("Опис виключення")
except MyError as err:
    print (err)
```

Результат:

```
Опис виключення
```

**raise** <Назва класу>

У другому варіанті формату інструкції **raise** у першому параметрі задають об'єкт класу, а не екземпляр.

**Приклад 16.**

```
try:
    raise ValueError # Еквівалентно:
raise ValueError()
except ValueError:
    print("Повідомлення про помилку")
```

**Приклад 17.** Приклад виконання вбудованого виключення `ValueError`:

```
>>> raise ValueError ("Опис виключення")

Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: Опис виключення
```

Приклад обробки цього виключення:

```
try:
    raise ValueError("Опис виключення")
except ValueError as msg:
    print(msg) # Виведе: Опис виключення
```

Результат виконання:

```
Опис виключення
```

**raise** <Екземпляр або назва класу> from <Об'єкт>

У третьому варіанті формату інструкції `raise` у першому параметрі задають екземпляр класу або просто назва класу, а в другому параметрі вказують об'єкт виключення.

При обробці вкладених виключень ці дані використовуються для виводу інформації не тільки про останнє виключення, але й про перше виключення.

В наступному прикладі ми одержали інформацію не тільки по виключенню `ValueError`, але й по виключенню `ZeroDivisionError`. При відсутності інструкції `from` інформація зберігається неявним чином.

**Приклад 18.**

```
try:
    x = 1 / 0
except Exception as err:
    raise ValueError() from err
```

Результат виконання:

```
Traceback (most recent call last):
  File " C:/PYTHON/first.py ", line 2, in
<module>
    x = 1 / 0
ZeroDivisionError: division by zero

The above exception was the direct cause of
the following exception:

Traceback (most recent call last):
  File " C:/PYTHON/first.py ", line 4, in
<module>
    raise ValueError() from err
ValueError
```

```
try:                (інструкція from відсутня)
    x = 1 / 0
except Exception as err:
    raise ValueError()
Результат виконання:
Traceback (most recent call last):
  File "C:/PYTHON/first.py", line 2, in
<module>
    x = 1 / 0
ZeroDivisionError: division by zero

During handling of the above exception,
another exception occurred:

Traceback (most recent call last):
  File "C:/PYTHON/first.py", line 4, in
<module>
    raise ValueError()
ValueError
```

## raise

Інструкція **raise** дозволяє повторно виконати останнє виключення й зазвичай застосовується в коді, наступному за інструкцією **except.**

**Приклад 19.**

```
class MyError(Exception): pass
try:
    raise MyError("Повідомлення про помилку")
except MyError as err:
    print (err)
    raise
```

Результат виконання:

```
Повідомлення про помилку
Traceback (most recent call last):
  File "C:/PYTHON/first.py", line 3, in
<module>
    raise MyError ("Повідомлення про
помилку")
__main__. MyError: Повідомлення про помилку
```

## Інструкція `assert`

Інструкція `assert` виконує виключення `AssertionError`, якщо логічний вираз повертає значення `False`. Інструкція має наступний формат:

```
assert <Логічний вираз> [, <Дані>)
```

Інструкція `assert` еквівалентна наступному коду:

**Приклад 20.**

```
if __debug__:
    if not <Логічний вираз>:
        raise AssertionError (<Дані>)
```

Якщо при запуску програми використовується прапор – `0`, то змінна `__debug__` буде мати неправильне значення. У такий спосіб можна вилучити всі інструкції `assert` з байт-коду.

Приклад використання інструкції `assert`:

**Приклад 21.**

```
try:
    x = -3
    assert x >= 0, "Повідомлення про помилку"
except AssertionError as err:
    print(err)
```

Результат

Повідомлення про помилку