exceptions: дерево наслідування

Згадаймо що виключення - це класи. А раз це класи то все що ми говорили про наслідування - працює і тут. Вбудовані виключення мають деяку ієрархію наслідування. На цьому етапі немає ніякої потреби запам’ятовувати її, треба розуміти деякі базові принципи побудови цього дерева наслідування:

- всі винятки є похідними класами від **BaseException**
- вбудовані винятки за допомогою наслідування згруповані відповідно до призначення і застосування
- використовувати вбудовані винятки необхідно саме за тим призначенням, ке описано в документації (приклади: TypeError, ValueError)

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├──
FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
. . .
```

Зверніть увагу як від класу **BaseException**, який описує загальну поведінку будь-якого виключення у **Python**, створено спеціалізований клас **Exception** - від якого створюються всі вбудовані винятки, які не походять від системи.
Далі для виключень, які можуть виникати при математичних операціях створено клас **ArithmeticError**, вже від якого створені конретні винятки (наприклад - займий нам **ZeroDivisionError**).

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ExceptionGroup [BaseExceptionGroup]
      ├── ImportError
      │    └── ModuleNotFoundError
      ├── LookupError
      │    ├── IndexError
      │    └── KeyError
      ├── MemoryError
      ├── NameError
      │    └── UnboundLocalError
      ├── OSError
      │    ├── BlockingIOError
      │    ├── ChildProcessError
      │    ├── ConnectionError
      │    │    ├── BrokenPipeError
      │    │    ├── ConnectionAbortedError
      │    │    ├── ConnectionRefusedError
      │    │    └── ConnectionResetError
      │    ├── FileExistsError
      │    ├── FileNotFoundError
      │    ├── InterruptedError
      │    ├── IsADirectoryError
      │    ├── NotADirectoryError
      │    ├── PermissionError
      │    ├── ProcessLookupError
      │    └── TimeoutError
      ├── ReferenceError
      ├── RuntimeError
      │    ├── NotImplementedError
      │    └── RecursionError
      ├── StopAsyncIteration
      ├── StopIteration
      ├── SyntaxError
      │    └── IndentationError
      │         └── TabError
      ├── SystemError
      ├── TypeError
      ├── ValueError
      │    └── UnicodeError
      │         ├── UnicodeDecodeError
      │         ├── UnicodeEncodeError
      │         └── UnicodeTranslateError
      └── Warning
           ├── BytesWarning
           ├── DeprecationWarning
           ├── EncodingWarning
           ├── FutureWarning
           ├── ImportWarning
           ├── PendingDeprecationWarning
           ├── ResourceWarning
           ├── RuntimeWarning
           ├── SyntaxWarning
           ├── UnicodeWarning
           └── UserWarning
```

exceptions: наслідування і перехоплення

Якщо ми згадаємо що похідний клас це спеціалізована версія базового класу (тобто - екземпляр похідного класу є також екземпляром базового), то стає зрозумілим чому такий код вважається потенційно неякісним:

```
try:
    # some code
    ...
except:
    # some exception handling
    ...
```

Такий код буде перехоплювати всі виключення в блоці “try: …”. Обробка, яка описана в блоці “except: …” як правило орієнтована на певні виключення і врахувати всі можливі ситуації - не може. Тому такий код вважається поганою практикою.

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├──
FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
   . . .
```

Варіант коду який наведено справа вже кращий, але все ж вважається не дуже добрим. Тому що будуть перехоплені всі виключення, які наслідувані від **Exception** - а їх доволі багато.
Прийнятним варіантом може бути перехоплення якоїсь невеликої групи - якщо Ви добре розумієте для чого це Вам. Наприклад, якби замість Exception ми б вказали **ArithmeticError** - ми б звузили перехоплення до виключень **FloatingPointError**, **OwerflowError**, **ZeroDivisionError**.

```
try:
    # some code
    ...
except Exception:
    # some exception handling
    ...
```

Хорошою практикою є максимально чітка вказівка в розділі except конкретного виключення, яке необхідно перехопити.

exceptions: створення власних виключень

Так як виключення це клас, то нічого нам не заважає створити свій клас виключення, який наслідує Exception (так рекомендовано документацією Python)

```
class MyException:
    pass
```

```
try:
    # some code
    ...
    raise MyException(“My
message”)
except MyException as exc:
    # some exception handling
    ...
```

Тепер ми можемо генерувати свій власний виняток в необхідних нам ситуаціях (наприклад - як сигнал про певну подію або як сигнал про специфічну помилку в бізнес-логіці або у якомусь реалізуємому нами специфічному протоколі).
При цьому ми точно будемо знати що джерелом генерації цього виключення може бути виключно наш код.
При використанні вбудованих виключень для обслуговування нашої логіки такої впевненості бути не може, бо генерація вбудованих виключень можлива на рівні ядра Python у відповідних ситуаціях.