Компьютерная математика II

Дисциплина для студентов 1-го курса специальности «Компьютерная математика и системный анализ» и профилизации "Искусственный интеллект и математическая экономика" ММФ БГУ

Тема 12. Программирование с использованием исключений

доц. Лаврова О.А., кафедра дифференциальных уравнений и системного анализа (ауд. 329)

май, 2025

Исключение -- это событие, который используется в Python для управления процессом последовательного выполнения программы.

Возможны следующие случаи использования исключений:

- возникновение ошибки при выполнении программы; Например, исключения *генерируются автоматически* при обнаружении интерпретатором ошибки в программе во время выполнения.
- возникновение особой ситуации при выполнении программы. Например, исключение `StopIteration` *генерируется автоматически* при достижении последнего элемента итерируемого объекта.

Если при выполнении последовательности действий может возникнуть исключение (ошибка, особая ситуация или др.), то легче обработать исключение, чем пытаться предусмотреть все условия, при которых исключения не будет (Easy to Ask Forgiveness than Permission, EAFP).

Исключения можно *генерировать/инициировать* в коде в процессе выполнения программы.

Исключения можно *перехватывать* и *обрабатывать* в процессе выполнения кода.

Инструментами для работы с исключениями являются следующие операторы:

- операторы `raise`, `assert` для генерации исключений
- оператор `try` для перехвата и обработки исключений

12.1 Стандартный обработчик исключений

При возникновении любого исключения Python вызывает **стандартный обработчик исключений**, если исключение не перехватывается в программе с помощью оператора `try`. Стандартный обработчик исключений выводит сообщение об ошибке и *останавливает* процесс выполнения программы.

In [9]: `sin(x)`

```
-----
NameError                                Traceback (most recent call last)
Cell In[9], line 1
----> 1 sin(x)

NameError: name 'sin' is not defined
```

Стандартное сообщение об ошибке содержит список всех строк кода, которые были активными на момент возникновения исключения, в примере

```
----> 1 sin(x)
```

а также имя сгенерированного исключения, в примере

NameError

вместе с сопутствующей информацией, в примере

```
name 'sin' is not defined`
```

Для того, чтобы исключение НЕ обрабатывалось стандартным обработчиком исключений, необходимо использовать оператор `try` для *перехвата* исключения и возможной его *обработки*.

```
In [11]: try:
          sin(x)
        except:
          print('got exception')
          print('process goes further')
```

```
got exception
process goes further
```

12.2 Классы исключений

В Python существуют *встроенные исключения*, можно также создать *пользовательские исключения*. Имена для встроенных исключений указаны во встроенной области видимости

```
In [15]: import builtins
```

```
In [17]: exc = [i for i in dir(builtins) if i.endswith("Error")]
          print(len(exc))
          print(exc[:7])
```

```
49
['ArithmeticError', 'AssertionError', 'AttributeError', 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'ChildProcessError']
```

Класс `BaseException` является корневым классом для всех исключений. Класс `BaseException` обеспечивает создание объектов исключений и стандартный вывод для всех исключений. Все классы исключений связаны в единое дерево наследования с корневым классом `BaseException`. Класс `BaseException` не рекомендовано использовать для прямого наследования пользовательскими классами.

```
In [20]: BaseException.__bases__
```

```
Out[20]: (object,)
```

```
In [22]: exc = BaseException('Arguments for constructor')
exc
```

```
Out[22]: BaseException('Arguments for constructor')
```

Класс `Exception` является подклассом класса `BaseException`

```
In [25]: Exception.__bases__
```

```
Out[25]: (BaseException,)
```

От класса `Exception` должны наследоваться все классы исключений, специфичные для разрабатываемых программ и определяемые пользователем

```
In [28]: class MyException(Exception):
...
raise MyException("Info!!!!")
```

```
-----
MyException                                Traceback (most recent call last)
Cell In[28], line 4
      1 class MyException(Exception):
      2     ...
----> 4 raise MyException("Info!!!!")

MyException: Info!!!!
```

Для обеспечения специализированного строкового представления для исключения необходимо реализовать перегрузку методов строкового представления `__repr__` и/или `__str__` для пользовательского класса исключения

```
In [31]: print(f'Before: {MyException()}=')
class MyException(Exception):
    def __repr__(self):
        return "It's a user-defined exception"

print(f'After: {MyException()}=')
```

```
Before: MyException()=MyException()
After: MyException()=It's a user-defined exception
```

Класс `LookupError` является встроенным подклассом класса `Exception` и суперклассом для всех ошибок индексирования: `IndexError`, `KeyError` и др.

```
In [34]: print(LookupError.__bases__)
[Class.__bases__ for Class in (IndexError, KeyError)]
(<class 'Exception'>,)

```

```
Out[34]: [(LookupError,), (LookupError,)]
```

Класс `ArithmeticError` является встроенным подклассом класса `Exception` и суперклассом для всех ошибок, связанных с числами: `OverflowError`, `ZeroDivisionError`, `FloatingPointError`

```
In [37]: print(ArithmeticError.__bases__)
[Class.__bases__ for Class in (OverflowError, ZeroDivisionError, FloatingPointEr
(<class 'Exception'>,)

```

```
Out[37]: [(ArithmeticError,), (ArithmeticError,), (ArithmeticError,)]
```

До Python 2.6 исключения были реализованы на основе строк.

Начиная с Python 2.6, объект исключения -- это всегда экземпляр класса, унаследованного от класса `BaseException`. В каждый момент времени активным может быть только один объект исключения. После перехвата исключений объект исключения перестает быть активным и удаляется.

Иерархию классов встроенных исключений можно посмотреть по ссылке:

<https://docs.python.org/3/library/exceptions.html#exception-hierarchy>

12.3 Оператор try

Перехват исключений, их обработка и восстановление процесса выполнения программы осуществляется с помощью составного оператора `try/except/else/finally`.

Совмещение конструкций `except` и `finally` в одном вызове оператора `try` возможно с версии Python 2.5.

Синтаксический шаблон составного оператора `try`

```
try:
    вложенный блок (основной код)
except ...: #необязательная конструкция; может быть несколько
    вложенный блок (обработка)
except ...:
    вложенный блок (обработка)
else:      #необязательная конструкция
    вложенный блок (postprocessing)
finally:   #необязательная конструкция
    вложенный блок (postprocessing)
```

Вложенный блок оператора `try` содержит код программы, который необходимо выполнить и для которого необходимо обработать исключения, возникающие при

выполнении этого кода.

Например, вызов функции большого размера можно поместить внутрь оператора `try` для контроля за всеми исключениями, возникающими внутри функции.

Рекомендовано, чтобы вложенный блок оператора `try` не содержал много строк кода.

- Перед началом выполнения оператора `try` интерпретатор Python запоминает *текущий контекст* программы, чтобы интерпретатор мог возвратиться к нему, если возникнет исключение.
- Сначала выполняется код внутри вложенного блока оператора `try`. Если при выполнении кода внутри вложенного блока оператора `try` возникает исключение, то интерпретатор завершает выполнение всех функций, которые оставались активными после входа в оператор `try`.
- Затем выполнение программы возобновляется на вложенном блоке конструкции `except`, дающей совпадение с возникшим исключением.
- Затем выполнение переходит во вложенный блок конструкции `finally`.
- Если исключение было обработано конструкцией `except`, то далее выполняется оператор или выражение, расположенное ПОСЛЕ оператора `try`.
- Если обработки исключения не было, то вызывается стандартный обработчик исключений.

```
In [50]: try:
         "str"[3]
         print("This operator is skipped")
     except:
         print("Block except")
     else:
         print("Block else is executed when no exceptions!")
     finally:
         print("Block finally is executed always!")
     print("Operator after 'try'")
```

```
Block except
Block finally is executed always!
Operator after 'try'
```

- Если при выполнении кода внутри вложенного блока оператора `try` НЕ возникает исключение, то интерпретатор выполняет весь код вложенного блока оператора `try`.
- Затем выполнение программы переходит во вложенный блок конструкции `else`.
- Затем выполнение переходит во вложенный блок конструкции `finally`.
- Далее выполняется оператор или выражение, расположенное ПОСЛЕ оператора `try`.

```
In [53]: try:
    "str"[1]
    print("Block try")
except:
    print("Block except")
else:
    print("Block else is executed when no exceptions!")
finally:
    print("Block finally is executed always!")
print("Operator after 'try'")
```

```
Block try
Block else is executed when no exceptions!
Block finally is executed always!
Operator after 'try'
```

Исключения, сгенерированные при выполнении вложенного блока оператора `try` сопоставляются с исключениями, перечисленными в конструкциях `except`.

Если при выполнении кода вложенного блока оператора `try` возникает исключение, то выполнение программы переходит в *соответствующую* конструкцию `except`, которая содержит код для *обработки* исключения.

- Конструкция `except name:` перехватывает исключение с именем `name`.
- Конструкция `except (name1, name2, ..., nameN)^:` перехватывает исключения с именами `name1`, `name2`, ... `nameN`.
- Конструкция `except:` соответствует всем (или всем остальным) исключениям, которые не перечислялись в конструкциях `except`, расположенных выше. Конструкция `except:` затрудняет обнаружение ошибок в коде и должна использоваться редко и аккуратно.

При возникновении исключения конструкции `except` проверяются сверху вниз и слева направо и выполняется код для первого совпадения.

```
In [59]: try:
    #1/0
    s
    #(1,2)[2]
except NameError:
    print("got exception NameError")
except (ValueError, ArithmeticError):
    print("got exceptions ValueError or ArithmeticError")
except:
    print("got unknown exception")
```

```
got exception NameError
```

Указание суперкласса в обработчике исключений приводит к перехвату экземпляров данного класса и экземпляров всех его подклассов при движении вниз по дереву наследования.

Поэтому **обработчики специфических исключений всегда должны предшествовать обработчикам более общих исключений.** Несоблюдение этого

правила приведет к тому, что обработчики более специфических исключений никогда не будут выполнены.

Для доступа к атрибутам и методам *объекта исключения* в заголовке конструкции `except` необходимо указать конструкцию `as`, за которой указывается переменная для хранения ссылки на объект исключения

```
In [64]: try:
          1/0
          #s
          #(1,2)[2]
        except NameError as e:
          print(e)
        except (ValueError, ArithmeticError) as e:
          print(e)
        except:
          print("got unknown exception")
```

division by zero

Конструкция `finally` содержит код для выполнения ПОСЛЕ выполнения блока оператора `try` (**код финализации, обработчик очистки**), независимо от того, возникает исключение при выполнении кода тела оператора `try` или нет.

```
In [67]: try:
          1/0
        except:
          print("infinity")
        finally:
          print("final actions")
```

infinity

final actions

```
In [69]: try:
          1/1
        except:
          print("infinity")
        finally:
          print("final actions")
```

final actions

При отсутствии обработчика для возникшего исключения конструкция `finally` не останавливает исключение

```
In [72]: try:
          abs(-5)
        except ValueError:
          print("infinity")
        finally:
          print("final actions")
```

final actions

```

-----
NameError                                Traceback (most recent call last)
Cell In[72], line 2
      1 try:
----> 2     abs(-5)
      3 except ValueError:
      4     print("infinity")

NameError: name 'abs' is not defined

```

Конструкция `finally` чаще всего содержит код по очистке, который должен происходить всегда при выходе из оператора `try`, например, закрытие файлов или разрыв соединения с сервером.

Не все встроенные исключения могут быть перехвачены. Например, исключение `KeyboardError`, которое генерируется после нажатия `Ctrl-C`, перехвачено быть не может.

Оператор `try` часто используется для реализации функционирования *серверов*, а также для *тестирования*.

12.4 Операторы `raise` и `assert`

При возникновении ошибки в коде Python *автоматически* генерируются встроенные исключения.

Операторы `raise` и `assert` используются для генерации/инициирования исключений по необходимости.

Генерировать можно как встроенные исключения, так и пользовательские исключения, определенные с помощью классов.

```
In [81]: raise NameError("Комментарий от разработчика")
```

```

-----
NameError                                Traceback (most recent call last)
Cell In[81], line 1
----> 1 raise NameError("Комментарий от разработчика")

NameError: Комментарий от разработчика

```

Результатом выполнения оператора `raise` является генерация исключения с последующим вызовом *стандартного обработчика исключений*.

Синтаксически после оператора `raise` может быть указан экземпляр класса или объект класса

```
In [85]: try:
          raise MyException()
        except MyException:
          print("got exception")
```

```
got exception
```

При выполнении оператора `raise` для объекта класса экземпляр класса будет создаваться *автоматически*

```
In [88]: try:
         raise MyException
        except MyException:
            print("got exception")
```

got exception

```
In [90]: try:
         raise MyException()
        except MyException:
            print("got exception")
```

got exception

Оператор `assert` представляет собой синтаксическое сокращение для распространенной схемы применения оператора `raise` для генерации встроенного исключения `AssertionError` и может считаться *условным* оператором `raise`

```
In [93]: a = 1.
         test = isinstance(a,int)
         info = f'int is expected but got {a}'
```

```
In [95]: if not test:
         raise AssertionError(info)
```

```
-----
AssertionError                                Traceback (most recent call last)
Cell In[95], line 2
      1 if not test:
----> 2     raise AssertionError(info)

AssertionError: int is expected but got 1.0
```

Аналогичная реализация с оператором `assert`

```
In [98]: assert test, info
```

```
-----
AssertionError                                Traceback (most recent call last)
Cell In[98], line 1
----> 1 assert test, info

AssertionError: int is expected but got 1.0
```

Если выражение `test` возвращает `False`, то генерируется исключение `AssertionError` и необязательное выражение `info` передается в качестве аргумента при создании исключения `AssertionError`.

Перехватывать программные ошибки с помощью оператора `assert` не нужно, так как Python генерирует исключения автоматически.

In [102...

```
x = 0
assert x!=0, "division by zero"
1/x
```

```
-----
AssertionError                                Traceback (most recent call last)
Cell In[102], line 2
      1 x = 0
----> 2 assert x!=0, "division by zero"
      3 1/x

AssertionError: division by zero
```

In [104...

```
1/x
```

```
-----
ZeroDivisionError                            Traceback (most recent call last)
Cell In[104], line 1
----> 1 1/x

ZeroDivisionError: division by zero
```

Операторы `assert` могут быть удалены из байт-кода программы в случае применения флага `-O` в командной строке при запуске программы.

12.5 Примеры использования пользовательских исключений

Пример 1. Оператор `break` прерывает выполнение только одного цикла, в котором он вызывается. С помощью генерации и перехвата *пользовательского исключения* можно прервать выполнение нескольких вложенных циклов.

In [109...

```
class ExitLoop(Exception):
    def __str__(self):
        return 'break from all loops'

try:
    while True:
        while True:
            for i in range(5):
                if i>3:
                    raise ExitLoop
except ExitLoop as e:
    print(e)
```

break from all loops

Пример 2. Если пользовательская функция, реализующая, например, поиск, кроме некоторого результата поиска должна возвращать состояние, обозначающее успех или неудачу поиска, то реализовать это можно с помощью *пользовательского исключения*.

In [112...

```
class Failure(Exception): ...

def search():
```

```
if ...: # успех
    return ...
else:   # неудача
    raise Failure

try:
    search()
except Failure:
    ... # неудача
else:
    ... # успех
```

12.6 Диспетчер контекста

Составной оператор `with/as` введен в Python в версии 2.6 для работы с объектами, которые поддерживают управление контекстом.

Оператор `with` предназначен для выполнения действиями *при входе* в контекст (инициализация контекста) и *при выходе* из контекста (финализация контекста), например, устанавливать и закрывать соединение с базой данных, закрывать файлы при выходе из контекста или временно изменять настройки системы после входа в контекст).

Синтаксический шаблон составного оператора `with` :

`with` выражение `as` переменная:

 вложенный блок

Выражение в заголовке оператора `with` должно возвращать объект, который поддерживает **протокол управления контекстом**. Сам объект называется **менеджер контекста**. Оператор `with` не является универсальным и применим только к объектам, которые поддерживают *протокол управления контекстом*. Такие объекты должны иметь методы `__enter__` и `__exit__` для выполнения действий в момент входа в контекст и выхода из контекста, соответственно.

Некоторые объекты встроенных типов поддерживают *протокол управления контекстами*.

Например, **файловые объекты** поддерживают протокол управления контекстами. После выполнения вложенного блока оператора `with` для файлового объекта *автоматически* закрывается файл, не зависимо от того, генерировалось или нет исключение во вложенном блоке оператора `with` .

```
In [121... with open('tmp.txt','r') as file:
    for line in file:
        print(line)
```

the first line

the second line

the last line

```
In [123... file
```

```
Out[123... <_io.TextIOWrapper name='tmp.txt' mode='r' encoding='cp1251'>
```

```
In [125... [hasattr(file, attr) for attr in ("__enter__", "__exit__") ]
```

```
Out[125... [True, True]
```

Хотя файловые объекты могут автоматически удаляться при сборке мусора, не всегда легко узнать, когда это произойдет. Оператор `with` для файлового объекта гарантирует закрытие файла.

Альтернативная реализация оператора `with` с помощью `try/finally` требует больше кода, но является более универсальной и не ограничена объектами, поддерживающими *протокол управления контекстами*.

```
In [129... try:
    file = open('tmp.txt', 'r')
    for line in file:
        print(line)
finally:
    if file: file.close()
```

the first line

the second line

the last line