Лекция 10

Обработка ошибок

Исключения

Исключения, терминология

- Исключение это событие, происшедшее во время исполнения программы
- Терминология:
  - программа *возбуждает* исключение (exception is triggered)
  - исключение *перехватывается* (catch exception)
  - исключение (ошибка) *обрабатывается* (handle exception)
- Инструкции для работы с исключениями
  - try/except - перехват исключения
  - try/finally - обработка без перехвата
  - raise - возбуждение исключения
  - assert - возбуждение исключения для отладки
  - with/as - менеджер контекста
- В большинстве случаев исключение возникает в результате ошибки исполнения программы

Традиционная обработка ошибок

- Каждая функция в которой может возникнуть ошибка возвращает статус, обозначающий нормальное или ошибочное завершение
- После каждого вызова функции проверяется возвращаемый статус и в случае ошибки выполняются действия по ее нейтрализации

```
/* Пример на языке C */
fd = open(filename);
if(fd < 0)
  /* File open error */
  return -1;
n = read(fd, buf, 2)
if(n < 2)
  /* File read error or file too short */
  return -2;
if(!(buf[0] == 'M' && buf[1] == 'Z'))
  /* Invalid file type */
  return -3;
```

Исключения, инструкция try

- Инструкция try вводит блок кода в котором может произойти исключение
- При возникновении исключения исполнение блока try прекращается и происходит *распространение исключения* (движение вверх по вложенным блокам try и / или по стеку вызовов)
- После перехвата исключения исполнение программы продолжается с места следующего за прерванным блоком кода
- Исключение может произойти на любом уровне вызова функций, то есть обрабатывается любое исключение, возникшее во время исполнения блока

```
try:                  # => try похож на if, условие - "нет ошибки"
  some_action()       # => Здесь может возникнуть исключение
except ExceptionName:
  process_exception() # => Обработка исключения в блоке except
action_after_exception()
```

Модель обработки исключения

```
error_code = None

while True: # вводит блок аналогичный try:
  error_code = some_action()
  if error_code:
    break
  error_code = other_action()
  if error_code:
    break
  break
if error_code: # вводит блок аналогичный except:
  process_exception(error_code)
```

- Все участки кода, где может возникнуть ошибка должны быть оформлены как функции
- Все функции должны возвращать код ошибки
- Если функция вернула ошибку, и вызов функции не был сделан внутри блока проверки, текущая функция должна немедленно вернуть полученный код ошибки инструкцией return

Несколько блоков except

- После блока try может следовать несколько блоков except
- При возникновении исключения будет исполнен соответствующий ему except блок

```
try:
  2.0 / 0.0
except ZeroDivisionError:
  print('Division by zero')
except IndexError:
  print('Invalid index')
else:
  print('No errors')
# => Division by zero
```

- Если исключения не было, исполняется блок *else*
- Блок else является дополнением к блоку *except*, а не к блоку try. Без блока except нельзя использовать блок else.

Синтаксис строки except

- Строка except имеет вид:

  `except [ описание_исключения [ as переменная ] ] :`

- *описание_исключения* это:
  - класс исключения
  - кортеж содержащий классы исключения; *использование скобок при записи кортежа обязательно*
- Исключение характеризуется объектом исключения. Если класс объекта исключения совпадает классом, указанным в строке except или *является его подклассом* исключение считается перехваченным, а блок следующий за строкой except становится его обработчиком.
- Если в строке except присутствует конструкция as *переменная*, объект исключения помещается в переменную
- Блоки except подобны блокам if / elif, проверка условий происходит сверху вниз, слева направо

Порядок блоков except

- Условия в блоках except должны идти в порядке от более специальных к более общим
- Пример: исключение ZeroDivisionError следует указать раньше, чем ArithmeticError, так как ArithmeticError перехватывает все арифметические ошибки:
  - FloatingPointError
  - OverflowError
  - ZeroDivisionError

  иначе программа никогда не дойдет до проверки исключения ZeroDivisionError

Преимущества исключений

- Текст программы становится короче и понятнее
- Появляется возможность защитится от ошибок возникающих в сторонних библиотеках
- Гибкость в обработке ошибок, можно:
  - не обрабатывать ошибки вообще,
  - обрабатывать все ошибки единообразно,
  - отдельные виды ошибок обрабатывать индивидуально
- Если исключение не было обработано в программе, интерпретатор вызывает обработчик по умолчанию
  - Обработчик по умолчанию прекращает выполнение программы и выводит подробное сообщение об ошибке включая стек вызовов с именами файлов и номерами строк
  - Имя класса исключения в сообщении об ошибке служит подсказкой для написания кода обработчика

Пример исключения

- Выход индекса за пределы списка

```
a = [0, 1, 2]

x = a[7] # Аварийное завершение программы.
         # В распечатке стека вызовов будет
         # виден класс исключения IndexError.

try:
  x = a[7]
  print('After assignment: x =', x)
except IndexError:
  print('ERROR:', 'Invalid index')

print('Code after try block')

# => ERROR: Invalid index
# => Code after try block
```

Исключения как инструмент программирования

- Исключение может вызываться фрагментом программы как результат работы, например "объект не найден" для процедуры поиска
- Парные функции (исключение или статус), пример:
  - index (возбуждает исключение)
  - find (возвращает статус)

```
s = 'abcd'
token = 'cz'
try:
  n = s.index(token)
  print('Token', token, 'found at index', n)
except ValueError:
  print('Token', token, 'not found, ValueError exception')
n = s.find(token)
if n < 0:
  print('Token', token, 'not found: find() return', n)
```

Итерация по коллекции

- Техника итерации основана на использовании исключений
- При переборе коллекции функция next() возвращает очередной элемент коллекции

- Если элементы коллекции закончились функция next() возбуждает исключение *StopIteration*

```
a = [0, 1, 2]
i = iter(a)
print(next(i)) # => 0
print(next(i)) # => 1
print(next(i)) # => 2
print(next(i)) # => исключение StopIteration
```

- Альтернатива исключению - второй параметр функции next(). Это значение будет возвращено при завершении перебора

```
print(next(i, None)) # => None
```

Функция-генератор

- Функция-генератор создает итерируемый объект по своим свойствам аналогичный коллекции
- В точке возврата из функции-генератора возбуждается исключение StopIteration

```
def gen():
  a = [0, 1, 2]
  for e in a:
    yield e

i = iter(gen())
print(next(i)) # => 0
print(next(i)) # => 1
print(next(i)) # => 2
print(next(i)) # => исключение StopIteration
```

- Внутренняя реализация инструкции for использует итератор. Исключение StopIteration перехватывается и код его обработки обеспечивает выход из цикла.

Модель цикла for

- Пример цикла for

```
a = [0, 1, 2]

for e in a:
  print(e)
else:
  print('All elements printed')
```

- Имитация цикла for из предыдущего примера:

```
a = [0, 1, 2]
i = iter(a)

while True:
  try:
    e = next(i)
    print(e)
  except StopIteration:
    print('All elements printed')
    break
```

Инструкция raise

- Инструкция *raise* возбуждает исключение, синтаксис:

```
raise exception_object [ from other_exception_object ]
```

- Используя raise программист имитирует возникновение ошибки
- С инструкцией raise можно использовать как объект класса исключения так и сам класс, Во втором случае объект класса исключения создается автоматически.

```
def divide_x_by_n(x, n):
  if abs(n < 0.1):
    raise ZeroDivisionError()
  else:
    return x / n

def divide_x_by_n(x, n):
  if abs(n < 0.1):
    raise ZeroDivisionError
  else:
    return x / n
```

Классы исключений

- Класс исключения это класс *BaseException* или класс наследующий от него. Класс *Exception* один из его потомков.
- Программист может создавать собственные классы исключений наследуя от класса Exception

```
class DeviceConnectionFailed(Exception):
  pass

def connect():
  raise DeviceConnectionFailed

try:
  connect()
  print('After connect')
except DeviceConnectionFailed:
  print('ERROR:', 'DeviceConnectionFailed')
print('Code after try block')

# => ERROR: DeviceConnectionFailed
# => Code after try block
```

Аргументы исключения

- Метод __init__() класса BaseException воспринимает любое количество аргументов, которые сохраняются в виде кортежа в атрибуте `args`
- Дочерние классы наследуют метод __init__(), если он не был переопределен
- При преобразовании объекта класса исключения в строку (вывод на печать), в нее включаются все аргументы из атрибута `args`

```
class DeviceConnectionFailed(Exception):
  pass
def connect3():
  raise DeviceConnectionFailed('Brorken,', 'distance', 12, 'miles')
try:
  connect3()
except DeviceConnectionFailed as e:
  print('ERROR:', ' '.join(map(str, e.args)))
# => ERROR: Brorken, distance 12 miles
```

Использование инструкции raise

- Имитация ошибки в объекте делая его поведение подобным поведению объекта встроенных классов
  - Пример: виртуальная коллекция, задан индекс или ключ для которого в коллекции нет элемента
- Сообщение о специфических ошибках, сопровождающееся информацией помогающей решить проблему
- Прием программирования, позволяющий быстро прекратить выполнение участка программы с возвратом управления через несколько вложенный циклов или несколько уровней вызова функций

Блок finally

- Исполнение блока finally происходит всегда, независимо от того произошло исключение в блоке try или нет

```
def divide_2_by(n):
  try:
    x = 2.0 / n
    print('A: x =', x)
  finally:
    print('B: Finally division by', n)
  print('C: After division by', n)

try:
  divide_2_by(2.0)
except ZeroDivisionError:
  print('D: Division by zero occured')
# => A: x = 1.0  B: Finally division by 2.0  C: After division by 2.0

try:
  divide_2_by(0.0)
except ZeroDivisionError:
  print('D: Division by zero occured')
# => B: Finally division by 0.0  D: Division by zero occured
```

Полный синтаксис инструкции try

- После блока try должен следовать как минимум один дополнительный блок. Это может быть либо блок except, либо блок finally.
- Одна инструкция try может содержать все дополнительные блоки, но:
  - блоки except записываются сразу после try
  - блок except без параметров должен следовать после всех блоков except с параметрами
  - блок else записывается после блоков except, без блока except использовать else нельзя
  - блок finally записывается последним

Пример инструкции try со всеми возможными блоками

```
try:
  pass
except IndexError:
  pass
except (ZeroDivisionError, OverflowError):
  pass
except:
  pass
else:
  pass
finally:
  pass
```

Исключение внутри обработчика исключения

- Инструкция try обрабатывает первое исключение возникшее в блоке try
- Исключения возникшие в блоках except, else или finally не находятся в текущем блоке try и поэтому этой инструкцией try не обрабатываются
- Такие исключения распространяются и обрабатываются обычным образом, например во внешнем блоке try заключающем в себе текущий блок try, или обработчиком внутри интерпретатора
- Только одно исключение может быть активно в определенный момент времени. После возбуждения исключения исполнение программы приостанавливается и начинается процесс распространения исключения вплоть до его перехвата.
- Внутри обработчика исключения возможен вызов raise без параметра. При этом обработка исключения прекращается и оно распространяется дальше.

Конструкция raise from

- При возбуждении исключения внутри обработчика исключения можно сохранить информацию об этом "внешнем" исключении и передать ее дальше

```
class DeviceConnectionFailed(Exception):
       pass
try:
  try:
    'Connection reset by peer'.index('connected')
    print('Connection established')
  except ValueError as ve:
    print('Internal except', ve.__class__)
    raise DeviceConnectionFailed from ve
except Exception as be:
  print('External except', be.__class__)
  print('      caused by', be.__cause__.__class__)

# => Internal except <class 'ValueError'>
# => External except <class '__main__.DeviceConnectionFailed'>
# =>       caused by <class 'ValueError'>
```

Инструкция assert

- Инструкция assert это условная форма инструкции raise используемая при отладке
- Конструкция

```
if __debug__:
  if not condition:
    raise AssertionError(data)
```

  может быть коротко записана как

```
assert condition, data
```

  Параметр data необязательный

- При обычном запуске интерпретатора : `__debug__ == True`
- При запуске интерпретатора с ключом `-O` : `__debug__ == False`
- *Изменить значение __debug__ программно нельзя*

Инструкция assert, пример

```
def divide_2_by(n):
  assert n != 0, n
  return 2.0 / n

try:
  divide_2_by(2)
  divide_2_by(0)
except AssertionError as ex:
  print('divide_2_by(', ex, ')')

# => divide_2_by( 0 )
```

- Инструкция assert это правило, которое не должно нарушаться
- Нарушение правила свидетельствует об ошибке программиста
- Исключение AssertionError, как правило, не перехватывается. Программа аварийно завершается и распечатка стека вызовов указывает на место обнаружения проблемы

Менеджер контекста with

- Конструкция with/as аналогична конструкции try/finally
- Конструкция with/as работает с объектами, поддерживающими протокол менеджера контекста

- Менеджер контекста гарантируют исполнение встроенного в объект кода завершения после окончания блока with, даже если внутри блока with произошло исключение

```
with open('sample.txt') as f:
  f.write('abc') # => Вызывает ошибку: файл открыт только для чтения

with open('sample.txt') as f:
  2.0 / 0.0 # => with срабатывает на любую ошибку

with open('sample.txt') as f:
  for line in f:
    print(line, end='')
```

- Во всех трех случаях файл будет закрыт

Протокол менеджера контекста

- Объект-аргумент менеджера контекста должен иметь методы __enter__() и __exit__()
- Метод __enter__() вызывается перед входом в блок with. Если присутствует опциональная конструкция *as var*, то переменной *var* будет присвоено значение, возвращенное методом __enter__()
- Метод __exit__() вызывается и в случае возникновения исключения и в случае нормального завершения блока with
- Если произошло исключение и метод __exit__() вернул False, исключение возбуждается повторно и распространяется вверх по вложенным блокам try и / или по стеку вызовов
- В одной строке with может быть несколько объектов-менеджеров контекста:

```
with open('test_a.txt', 'w') as fa, open('test_b.txt', 'w') as fb:
  fa.write('This is file a\n')
  fb.write('This is file b\n')
```

Реализация менеджера контекста

```
class RemoteDevice:
  def __enter__(self):
    self.device_handler = self.connect_remote_device()
    return self.device_handler

  def __exit__(self, type_, value, traceback):
    self.finalize_data_processing()
    self.disconnect_remote_device()
    return True # comment this line to propogate exception

  def connect_remote_device(self):
    h = open_device()
    init_device(h)
    return h

with RemoteDevice() as dev:
  while True:
    data = dev.read()
    if not data:
      break
    process_data_block(data)
```

Декоратор @contextmanager

- В модуле contextlib определен декоратор contextmanager облегчающий создание менеджеров контекста

```
from contextlib import contextmanager

@contextmanager
def remote_device():
  device_handler = connect_remote_device()
  try:
    yield device_handler # yield должен сработать только один раз
  finally:
    finalize_data_processing()
    disconnect_remote_device(device_handler)
    return True # comment this line to propogate exception

with remote_device() as dev:
  while True:
    data = read(dev)
    if not data:
      break
    process_data_block(data)
```

Классы Exception и BaseException

- Класс *BaseException* является вершиной иерархии классов исключений
- Класс *Exception* наследует от класса BaseException
- При создании собственных классов исключений в качестве родительского класса следует использовать класс Exception
- Кроме Exception от класса BaseException наследуют классы исключений, которые не рекомендуется перехватывать. Это исключения `SystemExit` и `KeyboardInterrupt` связанные с завершением программы, а также исключение `GeneratorExit`, возбуждаемое при завершении генератора коллекции.
- Строка `except BaseException as var:` перехватит любое исключение, переменная `var` позволит произвести его анализ
- Функция sys.exc_info() позволяет получить класс и экземпляр исключения, которое обрабатывается в настоящий момент. Это полезно при использовании except без параметров.

try/except/else/finally, заключение

- try: вводит блок в котором ожидается исключение
- except *name*: вводит блок, который исполняется при возникновении исключения с именем *name*
- except *name* as *var*: позволяет в переменной *var* получить объект исключения для дальнейшего анализа
- except: вводит блок, который исполняется при возникновении исключения, которое не было обработано никаким другим блоком except
- else: вводит блок, который исполняется только если исключения не было
- finally: вводит блок который исполняется всегда
- Два часто встречающихся вида использования инструкции try:
  - try/except - перехват и обработка исключения
  - try/finally - освобождение ресурсов и передача исключения дальше