

|  |
| --- |
| **Автономное учреждение  профессионального образования**  **Ханты-Мансийского автономного округа – Югры**  **«СУРГУТСКИЙ ПОЛИТЕХНИЧЕСКИЙ КОЛЛЕДЖ»**  **(АУ «Сургутский политехнический колледж»)** |
|  |
| СТРУКТУРНОЕ ПОДРАЗДЕЛЕНИЕ - 4  (Энергетическое отделение) |
|  |

**Отчет**

**по практической работе**

**«Инструментальные Средства»**

Организация обработки исключений

Выполнил:

студент очной формы обучения

Браславцев Трофим Сергеевич

Сургут 2024

**Введение**

Организация обработки исключений играет важную роль в создании устойчивых к ошибкам программ. Обработка исключений помогает предотвратить сбои программы и обеспечить корректное завершение работы, даже если возникли ошибки во время выполнения. В данной работе будет рассмотрено, как правильно организовать обработку исключений в программных модулях.

**Цель работы**

Цель данной работы – изучить методы и подходы к обработке исключений, научиться применять конструкции try-catch, обрабатывать и регистрировать исключения для повышения стабильности программного обеспечения.

**Практическая часть**

**Задание 1: Базовая обработка исключений**

1. Создайте простой программный модуль, в котором возникает ошибка (например, деление на ноль).

def divide(a, b):

return a / b

result = divide(10, 0)

print(result)

2. Реализуйте обработку исключения с помощью конструкции try-catch, чтобы предотвратить аварийное завершение программы

def divide(a, b):

try:

return a / b

except ZeroDivisionError as e:

print(f"Ошибка: {e}")

result = divide(10, 0)

print(result)

3. Выведите сообщение об ошибке и продолжите выполнение программы.

def divide(a, b):

try:

return a / b

except ZeroDivisionError as e:

print(f"Ошибка: {e}")

return None

result = divide(10, 0)

if result is None:

print("Продолжаем выполнение программы...")

else:

print(result)

**Задание 2: Иерархия исключений**

1. Изучите, как работают различные типы исключений в вашем языке программирования (например, стандартные исключения и их наследники).

В Python существует обширная иерархия исключений, начиная с базового класса `Exception`. Некоторые из стандартных исключений:

- `ZeroDivisionError`: возникает при делении на ноль.

- `IndexError`: возникает при выходе за границы индекса.

- `IOError`: возникает при ошибках ввода-вывода.

2. Создайте несколько примеров обработки различных исключений (например, деление на ноль, выход за границы массива, ошибки ввода-вывода).

def example\_exceptions():

try:

# Деление на ноль

result = 10 / 0

except ZeroDivisionError as e:

print(f"Ошибка деления на ноль: {e}")

try:

# Выход за границы массива

array = [1, 2, 3]

print(array[5])

except IndexError as e:

print(f"Ошибка выхода за границы массива: {e}")

try:

# Ошибка ввода-вывода

with open("non\_existent\_file.txt", "r") as file:

content = file.read()

except IOError as e:

print(f"Ошибка ввода-вывода: {e}")

example\_exceptions()

3. Задокументируйте различия между ними и предложите стратегию обработки каждой категории.

- ZeroDivisionError:

- Различия: Возникает при попытке деления на ноль.

- Стратегия обработки: Предотвратить деление на ноль путем проверки знаменателя перед операцией или обработать исключение с выводом сообщения об ошибке.

- IndexError:

- Различия: Возникает при попытке доступа к элементу, индекс которого выходит за границы массива.

- Стратегия обработки: Проверить индекс перед доступом к элементу или обработать исключение с выводом сообщения об ошибке.

- IOError:

- Различия: Возникает при ошибках ввода-вывода, например, при попытке открыть несуществующий файл.

- Стратегия обработки: Проверить существование файла перед его открытием или обработать исключение с выводом сообщения об ошибке.

**Задание 3: Генерация пользовательских исключений**

1. Создайте пользовательский класс исключений для обработки специфических ситуаций в вашем программном проекте.

class CustomError(Exception):

def \_\_init\_\_(self, message):

self.message = message

super().\_\_init\_\_(self.message)

def custom\_exception\_example():

try:

raise CustomError("Это пользовательское исключение")

except CustomError as e:

print(f"Поймано пользовательское исключение: {e}")

custom\_exception\_example()

2. Напишите пример кода, который генерирует и обрабатывает это исключение.

class CustomError(Exception):

def \_\_init\_\_(self, message):

self.message = message

super().\_\_init\_\_(self.message)

def custom\_exception\_example():

try:

raise CustomError("Это пользовательское исключение")

except CustomError as e:

print(f"Поймано пользовательское исключение: {e}")

custom\_exception\_example()

```

3. Обоснуйте необходимость создания пользовательского исключения.

Создание пользовательских исключений необходимо для:

- Улучшения читаемости кода: Пользовательские исключения делают код более понятным, так как они явно указывают на специфические ошибки, характерные для конкретного проекта.

- Более тонкой обработки ошибок: Пользовательские исключения позволяют обрабатывать ошибки более детально, чем стандартные исключения, что может улучшить стабильность и надежность программы.

- Легкости поддержки: Пользовательские исключения упрощают поддержку кода, так как разработчики могут легко идентифицировать и исправлять специфические ошибки.

**Задание 4: Логирование исключений**

1. Реализуйте механизм логирования исключений в файл или в базу данных для их дальнейшего анализа.

import logging

#Настройка логирования

logging.basicConfig(filename='errors.log', level=logging.ERROR)

def log\_exception(e):

logging.error(f"Ошибка: {e}")

def example\_with\_logging():

try:

result = 10 / 0

except ZeroDivisionError as e:

log\_exception(e)

print("Произошла ошибка, информация записана в лог.")

example\_with\_logging()

2. Проверьте корректность логирования и убедитесь, что каждая ошибка записывается с указанием времени, типа ошибки и контекста.

import logging

from datetime import datetime

# Настройка логирования

logging.basicConfig(filename='errors.log', level=logging.ERROR, format='%(asctime)s - %(levelname)s - %(message)s')

def log\_exception(e):

logging.error(f"Ошибка: {e}")

def example\_with\_logging():

try:

result = 10 / 0

except ZeroDivisionError as e:

log\_exception(e)

print("Произошла ошибка, информация записана в лог.")

example\_with\_logging()

3. Задокументируйте подходы к анализу и устранению исключений на основе логов.

- Анализ логов:

- Время возникновения: Логирование времени позволяет определить, когда произошла ошибка.

- Тип ошибки: Тип ошибки помогает идентифицировать ее природу.

- Контекст: Дополнительная информация о контексте (например, параметры функции) помогает воспроизвести ошибку и понять ее причину.

- Устранение исключений:

- Исправление кода: Исправьте ошибку в коде, если она воспроизводима.

- Дополнительная проверка: Добавьте дополнительные проверки в код, чтобы предотвратить возникновение подобных ошибок в будущем.

- Улучшение обработки исключений: Улучшите обработку исключений, чтобы программа могла корректно обрабатывать подобные ситуации.

**Задание 5: Обработка исключений в многопоточности**

1. Если ваш проект использует многопоточность, реализуйте обработку исключений в многопоточной среде.

import threading

def worker():

try:

result = 10 / 0

except ZeroDivisionError as e:

print(f"Ошибка в потоке: {e}")

threads = []

for i in range(5):

t = threading.Thread(target=worker)

threads.append(t)

t.start()

for t in threads:

t.join()

2. Убедитесь, что исключения в одном потоке не приводят к завершению программы или других потоков.

import threading

def worker():

try:

result = 10 / 0

except ZeroDivisionError as e:

print(f"Ошибка в потоке: {e}")

threads = []

for i in range(5):

t = threading.Thread(target=worker)

threads.append(t)

t.start()

for t in threads:

t.join()

print("Программа продолжает выполнение после ошибки в потоке.")

3. Используйте синхронизацию для корректной работы с исключениями в многопоточных приложениях.

import threading

import logging

# Настройка логирования

logging.basicConfig(filename='errors.log', level=logging.ERROR, format='%(asctime)s - %(levelname)s - %(message)s')

def log\_exception(e):

logging.error(f"Ошибка: {e}")

def worker():

try:

result = 10 / 0

except ZeroDivisionError as e:

log\_exception(e)

print(f"Ошибка в потоке: {e}")

threads = []

for i in range(5):

t = threading.Thread(target=worker)

threads.append(t)

t.start()

for t in threads:

t.join()

print("Программа продолжает выполнение после ошибки в потоке.")

**Вывод**

В данной работе были рассмотрены методы и подходы к обработке исключений в программных модулях. Были изучены базовые конструкции try-catch, иерархия исключений, генерация пользовательских исключений, логирование исключений, а также обработка исключений в многопоточной среде. Правильная организация обработки исключений повышает стабильность и надежность программного обеспечения, предотвращая аварийные завершения и обеспечивая корректное завершение работы даже в случае возникновения ошибок.