Python-8
Исключения и обработка ошибок (try-except).

История <Exception>

Идея использования механизма исключений в программировании имеет долгую историю и была развита в контексте различных языков программирования.

Одним из первых языков, где были введены исключения, является язык программирования **ALGOL** 68, созданный в 1968 году. В **ALGOL** 68 впервые был представлен механизм исключений для обработки ошибок и исключительных ситуаций.

Впоследствии многие современные языки программирования включили в свои конструкции механизмы обработки исключений. Например, в языке **Ada** (1980) исключения использовались для обработки ошибок. Язык **C++** (1983) также ввел ключевые слова try, catch, и throw для обработки исключений. **Java** (1995) также известна своим механизмом исключений.

Обработка исключений стала важным компонентом программирования, обеспечивая структурированное и надежное управление ошибками в программах.

Зачем нужны исключения?

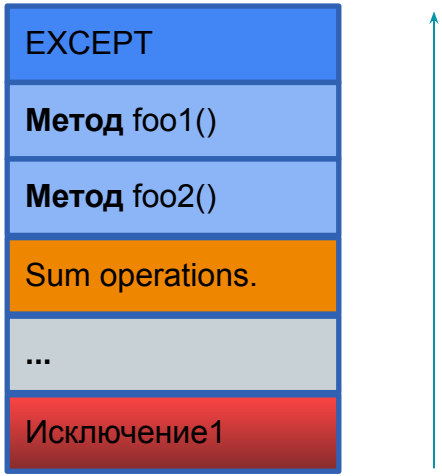
Программы редко могут быть идеальными, и ошибки могут возникнуть по разным причинам: некорректные данные, проблемы с ресурсами, сетевые сбои и т.д. **Исключения** предоставляют механизм для обработки и управления такими ситуациями.

Использование исключений позволяет программистам избежать неожиданных сбоев программы и предоставляет контроль над потенциально проблемными участками кода.

Как работают исключения в Python?

В Python, когда возникает какая-либо производственная, техническая, обычная, лексическая или любая др. ошибка, создается так называемый специальный объект-исключение. Этот объект затем "поднимается" по стеку вызовов, пока не найдется блок кода, который его обработает. И именно так и работают исключения в языке программирования Python.

(Исключение передается по стеку, начиная с текущей функции и двигаясь вверх по уровням вызовов.)



Обработка исключений. TRY-EXCEPT.

В Python существует единая структура обработки исключений - try-except.

Блок TRY: это такой блок кода, где может возникнуть исключение.

Блок Except: блок обработки исключения.

Дополнительные блоки.
Блок ELSE: блок выполняется, если исключение не возникло.
Блок FINALLY: блок выполняется всегда, независимо от того, было исключение или нет.

Пример. Простой TRY-EXCEPT.

```
try:
    # блок кода, где может возникнуть исключение
    result = 10 / 0  # пример деления на ноль
except ZeroDivisionError as e:
    # блок обработки исключения
    print(f"Ошибка: {e}")
```

Пример с ELSE/FINALLY

```
try:
    # блок кода, где может возникнуть исключение
    result = 10 / 0  # пример деления на ноль
except ZeroDivisionError as e:
    # блок обработки исключения
    print(f"Ошибка: {e}")
else:
    # блок выполняется, если исключение не возникло
    print("Операция выполнена успешно.")
finally:
    # блок выполняется всегда, независимо от того, было исключение или нет
    print("Завершение операции.")
```

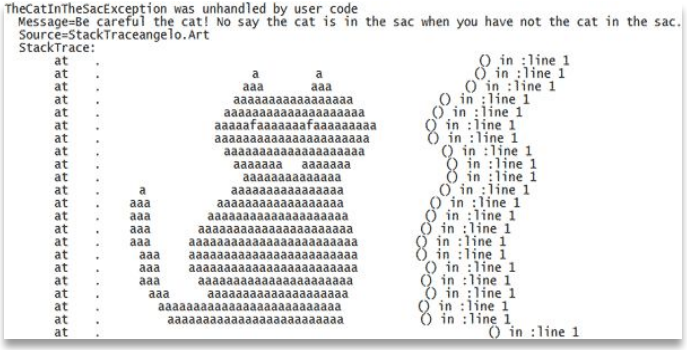
Пример если вызов функции.

```python
def foo():
    result = 10 / 0
    return result

try:
    result = foo()
except ZeroDivisionError as e:
    print(f"Ошибка: {e}")
```

Типы/Виды исключений.

- **ZeroDivisionError**: Возникает при делении на ноль.
- **FileNotFoundError**: Возникает, когда программа пытается открыть файл, который не существует.
- **TypeError**: Возникает, когда операция применяется к объекту несоответствующего типа.
- **ValueError**: Возникает, когда встроенная операция получает аргумент правильного типа, но с некорректным значением.



Синтаксис написания в блоке EXCEPT.

```
except TypeError as error:
except ValueError as e:
```

Стратегии обработки исключений.

- **Предварительная проверка (Pre-check)**: Проверьте данные перед выполнением операций, чтобы избежать исключений.
- Использование **нескольких блоков except** для различных типов исключений.
- Обработка и логирование исключений с использованием **try-except** конструкции.
- Блок **else** для кода, который выполняется, если исключение не возникло.
- Блок **finally** для кода, который выполняется всегда, независимо от того, было ли исключение или нет.

Внутреняя обработка исключения в функции.

```python
def divide_numbers(a, b):
    try:
        result = a / b
    except ZeroDivisionError as e:
        print(f"Ошибка деления на ноль: {e}")
    except TypeError as e:
        print(f"Ошибка типов: {e}")
    else:
        print(f"Результат деления: {result}")
    finally:
        print("Операция завершена.")

divide_numbers(10, 2)
divide_numbers(10, 0)
divide_numbers("10", 2)
```

Оператор RAISE.

Оператор **raise** в Python используется для явного выбрасывания исключения в коде. Это может быть полезно, когда вы хотите создать исключение в определенных условиях или с определенными данными.

Синтаксис: raise тип_ошибки(сообщение_об_ошибки)

Пример:
raise ValueError(*"Это сообщение об ошибке"*)

```
raise ValueError  # Выброс исключения без сообщения
```

Пример RAISE

```python
def foo(a, b):
    if b == 0:
        raise ZeroDivisionError("деление на ноль")
    return a / b

try:
    res1 = foo(4, 0)
    res2 = foo(4, 2)
    print(res1, res2)
except ZeroDivisionError as e:
    print(f"Ошибка: {e}")
    # Дополнительные действия при делении на ноль, если необходимо
```

Создание пользовательских исключений

Вы можете определить свой собственный класс исключения, унаследованный от базового класса <Exception>.

Пример #1 (без методов):

```
class CustomError(Exception):
    def __init__(self, message):
        super().__init__(message)

try:
    raise CustomError("Сообщение об ошибке")
except CustomError as e:
    print(f"Поймано пользовательское исключение: {e}")
```

Пример #2 с методами:

```
class CustomError(Exception):
    def __init__(self, message, code):
        super().__init__(message)
        self.code = code

    def get_error_code(self):
        return self.code

try:
    raise CustomError("Сообщение об ошибке", 500)
except CustomError as e:
    print(f"Поймано пользовательское исключение: {e}")
    print(f"Код ошибки: {e.get_error_code()}")
```