Материалы занятия

Курс: Разработка Web-приложений на Python, с применением Фреймворка Django
Дисциплина: Основы программирования на Python

Тема занятия №13: Обработка исключений. Собственные исключения

Исключения (exceptions) - ещё один тип данных в python. Исключения необходимы для того, чтобы сообщать программисту об ошибках.

1. Типы исключений

У каждого исключения есть собственный тип, который определяется тем, какая ошибка его вызвала, и дает возможность по-разному реагировать на различные виды ошибок.

Рассмотрим иерархию встроенных в python исключений, хотя иногда вам могут встретиться и другие, так как программисты могут создавать собственные исключения. Данный список актуален для python 3.3, в более ранних версиях есть незначительные изменения.

BaseException - базовое исключение, от которого берут начало все остальные.

- **SystemExit** - исключение, порождаемое функцией sys.exit при выходе из программы.
- **KeyboardInterrupt** - порождается при прерывании программы пользователем (обычно сочетанием клавиш Ctrl+C).
- **GeneratorExit** - порождается при вызове метода close объекта generator.
- **Exception** - а вот тут уже заканчиваются полностью системные исключения (которые лучше не трогать) и начинаются обыкновенные, с которыми можно работать.
  - **StopIteration** - порождается встроенной функцией next, если в итераторе больше нет элементов.
  - **ArithmeticError** - арифметическая ошибка.
    - **FloatingPointError** - порождается при неудачном выполнении операции с плавающей запятой. На практике встречается нечасто.
    - **OverflowError** - возникает, когда результат арифметической операции слишком велик для представления. Не появляется при обычной работе с целыми числами (так как python поддерживает длинные числа), но может возникать в некоторых других случаях.
    - **ZeroDivisionError** - деление на ноль.
  - **AssertionError** - выражение в функции assert ложно.
  - **AttributeError** - объект не имеет данного атрибута (значения или метода).
  - **BufferError** - операция, связанная с буфером, не может быть выполнена.
  - **EOFError** - функция наткнулась на конец файла и не смогла прочитать то, что хотела.
  - **ImportError** - не удалось импортирование модуля или его атрибута.
  - **LookupError** - некорректный индекс или ключ.
    - **IndexError** - индекс не входит в диапазон элементов.
    - **KeyError** - несуществующий ключ (в словаре, множестве или другом объекте).
  - **MemoryError** - недостаточно памяти.
  - **NameError** - не найдено переменной с таким именем.

        - **UnboundLocalError** - сделана ссылка на локальную переменную в функции, но переменная не определена ранее.
    - **OSError** - ошибка, связанная с системой.
        - **BlockingIOError**
        - **ChildProcessError** - неудача при операции с дочерним процессом.
        - **ConnectionError** - базовый класс для исключений, связанных с подключениями.
            - **BrokenPipeError**
            - **ConnectionAbortedError**
            - **ConnectionRefusedError**
            - **ConnectionResetError**
        - **FileExistsError** - попытка создания файла или директории, которая уже существует.
        - **FileNotFoundError** - файл или директория не существует.
        - **InterruptedError** - системный вызов прерван входящим сигналом.
        - **IsADirectoryError** - ожидался файл, но это директория.
        - **NotADirectoryError** - ожидалась директория, но это файл.
        - **PermissionError** - не хватает прав доступа.
        - **ProcessLookupError** - указанного процесса не существует.
        - **TimeoutError** - закончилось время ожидания.
    - **ReferenceError** - попытка доступа к атрибуту со слабой ссылкой.
    - **RuntimeError** - возникает, когда исключение не попадает ни под одну из других категорий.
    - **NotImplementedError** - возникает, когда абстрактные методы класса требуют переопределения в дочерних классах.
    - **SyntaxError** - синтаксическая ошибка.
    - **IndentationError** - неправильные отступы.
    - **TabError** - смешивание в отступах табуляции и пробелов.
    - **SystemError** - внутренняя ошибка.
    - **TypeError** - операция применена к объекту несоответствующего типа.
    - **ValueError** - функция получает аргумент правильного типа, но некорректного значения.
    - **UnicodeError** - ошибка, связанная с кодированием / раскодированием unicode в строках.
        - **UnicodeEncodeError** - исключение, связанное с кодированием unicode.
        - **UnicodeDecodeError** - исключение, связанное с декодированием unicode.
        - **UnicodeTranslateError** - исключение, связанное с переводом unicode.
    - **Warning** - предупреждение.

2. Перехват исключений

Для обработки исключений в Python используют конструкцию «try … except». В общем случае для построения этой конструкции необходимо:

- открыть блок «try», введя соответствующую инструкцию и двоеточие;
- указать набор инструкций, в результате работы которых может возникнуть исключение;
- выделить набор инструкций отступом;
- открыть блок «except», введя соответствующую инструкцию и двоеточие;
- указать набор инструкций, которые нужно выполнить в случае возникновения исключения;
- выделить набор инструкций отступом.

Рассмотри пример, на склад доставили 10 ящиков, также на складе есть бочки. Программа учета хранимых на складе предметов просит пользователя указать, сколько и какие именно предметы были доставлены:

```
amount = int(input("Enter the amount of received  items: "))
items_type = input("Specify the type of received  items: ")
```

Просьба указать тип полученных элементов отображается только после введения и числа, пользователи, которые не знакомы с программой, могут допустить ошибку:

```
Enter the amount of received items: 10 boxes
```

Часть введенной строки «boxes» нельзя привести к целому числу, вследствие чего будет вызвано исключение «ValueError».

```
Enter the amount of received items: 10 boxes
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    amount = int(input("Enter the amount of received  items: "))
ValueError: invalid literal for int() with base 10: '10 boxes'
```

Для обработки этого исключения применим конструкцию «try … except»:

```
try:
 amount = int(input("Enter the amount of received  items: "))
 items_type = input("Specify the type of received  items: ")
except:
 print("Amount should be an integer")
```

Результат программы:

```
Enter the amount of received  items: 10 boxes
Amount should be an integer
```

Ещё две инструкции, относящиеся к нашей проблеме, это finally и else. Finally выполняет блок инструкций в любом случае, было ли исключение, или нет (применима, когда нужно непременно что-то сделать, к примеру, закрыть файл). Инструкция else выполняется в том случае, если исключения не было.

```
try:
 amount = int(input("Enter the amount of received  items: "))
 items_type = input("Specify the type of received  items: ")
 parts_number = int(input("Enter the number of  parts: "))
 parts_amount = amount / parts_number
 print("Supply of", amount, items_type, "saved")
 print("Each of", parts_number, "parts consists of",
 parts_amount, items_type)
except ValueError:
 print("Amount should be an integer")
```

Блок «finally» используется в том случае, когда нам необходимо гарантировать окончание работы программы.

Обработка исключений

Введение

Исключения — один из двух основных типов ошибок в программировании. В отличие от синтаксических ошибок, которые возникают во время написания, исключения могут появиться во время выполнения программы. Примером такого различия может служить автомобиль: он может быть неисправен, что сделает путешествие на нем невозможным (подобно синтаксическим ошибкам), кроме того, водитель может не справиться с управлением, что приведет к ДТП уже во время поездки (подобно исключениям). В программировании же исключения приводят не к ДТП, а к полному прекращению или к неверному выполнению программ. Для избегания прекращения работы или получения дополнительной информации об ошибке используют конструкции обработки исключений.

Типы исключений

У каждого исключения есть собственный тип, который определяется тем, какая ошибка его вызвала, и дает возможность по-разному реагировать на различные виды ошибок.

Рассмотрим некоторые типы исключений, которые могут возникнуть во время прохождения этого курса:

BaseException — базовый тип, из которого происходят все остальные, в том числе системные:

Exception — базовый тип для «стандартных» и пользовательских исключений:

ArithmeticError — арифметическая ошибка:

OverflowError — возникает, когда результат арифметической операции слишком велик для представления;

На сегодняшний день практически невозможно получить OverflowError используя стандартную библиотеку Python, т.к. целые числа имеют динамическую длину и скорее вызовут MemoryError, а числа с плавающей запятой при пересечении граничных значений заменяются на 0.0, либо inf. Однако, эта ошибка может быть получена при работе со сторонними библиотеками или модулями, написанными на языках программирования с жесткой типизацией (числовые типы имеют четкие границы, выход за которые — ошибка).

ZeroDivisionError — деление на ноль.

ImportError — импортировать модуль или его атрибут не удалось;

LookupError — некорректный индекс или ключ:

IndexError — индекс не входит в диапазон элементов;

KeyError — несуществующий ключ (например, в словаре).

NameError — не найдено переменной с указанным именем;

RuntimeError — возникает, когда исключение не попадает ни под одну из других категорий;

SyntaxError — синтаксическая ошибка:

IndentationError — неправильные отступы:

TabError — смешивание в отступах табуляции и пробелов.

TypeError — операция применена к объекту несоответствующего типа;

ValueError — функция получает аргумент правильного типа, но некорректного значения.

Список предусмотренных самим языком программирования («встроенных») типов исключений гораздо больше и приведен в документации к языку. Кроме того, пользователи сами могут определять новые типы исключений в своих программах и библиотеках.

Перехват исключений

Для обработки исключений в Python используют конструкцию «try … except». В общем случае для построения этой конструкции необходимо:

■ открыть блок «try», введя соответствующую инструкцию и двоеточие;

■ указать набор инструкций, в результате работы которых может возникнуть исключение;

■ выделить набор инструкций отступом;

■ открыть блок «except», введя соответствующую инструкцию и двоеточие;

■ указать набор инструкций, которые нужно выполнить в случае возникновения исключения;

■ выделить набор инструкций отступом.

Применение данной конструкции рассмотрим на следующем примере: на склад доставили 10 ящиков, также на складе есть бочки. Программа учета хранимых на складе предметов просит пользователя указать, сколько и какие именно предметы были доставлены:

```
amount = int(input("Enter the amount of received
                items: "))
items_type = input("Specify the type of received                items: ")
```

Т. к. просьба указать тип полученных элементов отображается только после введения их числа, пользователи, которые не знакомы с программой, могут допустить ошибку:

Enter the amount of received items: 10 boxes

Часть введенной строки «boxes» нельзя привести к целому числу, вследствие чего будет вызвано исключение «ValueError».

Пример

Для обработки этого исключения применим конструкцию «try … except»:

```
try:
  amount = int(input("Enter the amount of received

                items: "))

 items_type = input("Specify the type of received

                items: ")

except:
  print("Amount should be an integer")
```

Результат работы нашего кода в консоли:

Enter the amount of received items: 10 boxes

Amount should be an integer

Так, если пользователь введет строку, которую нельзя привести к целому числу, выполнение блока «try» будет прервано (все последующие инструкции блока выполнения будут пропущены), и выполнение перейдет к блоку «except». Допустим, программа учета разделяет полученные предметы на несколько равных частей, чтобы их можно было расположить на разных уровнях склада. Добавим в блок «try» следующие инструкции:

parts_number = int(input("Enter the number of parts: ")) parts = amount / parts_number

Если пользователь в качестве «parts_number» укажет «0», произойдет ошибка, которая соответствует типу исключения «ZeroDivisionError» (деление на ноль невозможно). Однако сообщение об ошибке, показанное пользователю, останется прежним. Для того чтобы при разных ошибках выполнялись разные инструкции, необходимо создать несколько блоков «except» и указать типы исключений, при которых они будут выполняться:

```
try:
  amount = int(input("Enter the amount of received

                items: "))

  items_type = input("Specify the type of received
                items: ")

  parts_number = int(input("Enter the number of
                    parts: "))
  parts_amount = amount / parts_number

  print("Supply of", amount, items_type, "saved")

  print("Each of", parts_number, "parts consists of",

      parts_amount, items_type)

except ValueError:

  print("Amount should be an integer")

except ZeroDivisionError:
  print("You cannot divide the delivery into 0 parts")
```

Результат работы нашего кода в консоли:

Enter the amount of received items: 10

Specify the type of received items: boxes

Enter the number of parts: 0

You cannot divide the delivery into 0 parts

Кроме того, конструкцией «try … except» предусмотрен блок «finally», инструкции которого будут выполнены вне зависимости от того, возникнет исключение или нет:

```
try:
```

```
    amount = int(input("Enter the amount of received

                    items: "))

    items_type = input("Specify the type of received

                    items: ")

    parts_number = int(input("Enter the number of

                    parts: "))

    parts_amount = amount / parts_number

    print("Supply of", amount, items_type, "saved")

    print("Each of", parts_number, "parts consists of",

          parts_amount, items_type)

  except ValueError:
    print("Amount should be an integer")

  except ZeroDivisionError:
    print("You cannot divide the delivery into 0 parts")
  finally:
    print("The program has finished")
```

Блок «finally» используется в том случае, когда нам необходимо гарантировать окончание работы программы. Например, мы можем быть подключены к удаленному серверу по сети, работать с файлом или графическим интерфейсом пользователя (GUI). Во всех этих случаях мы должны освободить используемые программой ресурсы до того, как программа завершиться вне зависимости от того, успешно она выполнилась или нет. Такие действия (освобождение ресурсов) выполняются в блоке «finally». Рассмотрим общий случай применения на примере работы с файлом:

```
try:
  f = open("test.txt",'w')

  # perform file operations

finally:
  f.close()
```

Особенности работы с try … except

Вызов исключений

В Python мы можем не только перехватывать исключения, но и напрямую вызывать их. Для этого необходимо использовать ключевое слово «raise». С помощью «raise» мы можем указать, к какому типу будет принадлежать вызванное исключение:

```
try:
    apples = int(input("Enter the amount of apples
                    you have: "))
```

```
    if apples < 0:

        raise Exception

    print("You have", apples, "apples")

except Exception:
  print("You can’t have -10 apples")
```

Результат работы нашего кода в консоли:

```
Enter the amount of apples you have: -10
You can’t have -10 apples
```

Вызов исключений может потребоваться в тех случаях, когда логика программы требует дополнительных условий. Например, в приведенном выше примере для программы не важно, будет введено 10 или -10, т.к. это целые числа, но мы с вами знаем что для обозначения количества предметов используются числа натуральные (1, 2, 3…) и ноль, следовательно, яблок никак не может быть -10. Порядок размещения блоков exept

Одной из главных особенностей работы с «try … except» является правильное расположение блоков «except». В предыдущем примере с «ValueError» и «ZeroDivisionError» расположение не играло роли, поскольку ни один из них не является подтипом другого. Из таблицы типов исключений, приведенной в соответствующем разделе, видно, что «ZeroDivisionError» является подтипом «ArithmeticError», а «ValueError» — подтипом «Exception». Полную структуру иерархии встроенных типов исключений можно посмотреть в документации к языку. Рассмотрим влияние расположения блоков на работу программы на примере пары «ValueError» и «Exception»:

```
try:
  raise Exception

except Exception:

  print("Hmm... Something went wrong")

except ValueError:
  print("Improper value was obtained")
```

Результат работы нашего кода в консоли:

```
Hmm... Something went wrong
```

В данном примере внутри блока try возникает исключение Exception, которое обрабатывается соответствующим блоком except. Изменим тип вызываемого исключения:

```
try:
  raise ValueError

except Exception:

  print("Hmm... Something went wrong")

except ValueError:
  print("Improper value was obtained")
```

Результат работы нашего кода в консоли:

Hmm... Something went wrong

«ValueError» является подтипом «Exception», возникшее исключение будет обработано первым блоком, а второй, предназначенный для обработки исключений такого типа, выполнен не будет. Для того, чтобы исправить это, необходимо изменить порядок следования блоков except. Общие типы исключений должны быть расположены ниже частных:

```
try:
  raise ValueError

except ValueError:

  print("Improper value was obtained")

except Exception:
  print("Hmm... Something went wrong")
```

Результат работы нашего кода в консоли:

Improper value was obtained

Комбинирование общего и конкретных блоков exept

Мы так же можем комбинировать использование блоков «exept» для конкретных типов исключений и общий блок «exept». В таком случае общий блок должен быть размещен последним, он выполниться если тип полученного исключения не совпадает ни с одним из типов блоков «exept»:

```
try:
  f = open("Some_file.txt")

except ZeroDivisionError:
  print("You cannot divide the delivery into 0

parts")

except:
  print("Hmm... Something went wrong")
```

Результат работы нашего кода в консоли:

Hmm... Something went wrong

В этом примере мы пытаемся открыть файл «Some_file. txt», который не был создан ранее, что приводит к исключению типа «FileNotFoundError», однако для этого типа не предусмотрен отдельный блок «exept» и оно обрабатывается общим блоком.

Блок exept для нескольких типов исключений

Python так же позволяет один блок «exept» для разных типов исключений. Для этого их необходимо разместить через запятую внутри круглых скобок. Для рассмотрения этой особенности немного видоизменим один из предыдущих примеров:

```
try:
  amount = int(input("Enter the amount of received

                items: "))

  items_type = input("Specify the type of received
```

```
                items: ")

    parts_number = int(input("Enter the number of parts:"))

    parts_amount = amount / parts_number

    print("Supply of", amount, items_type, "saved")

    print("Each of", parts_number, "parts consists of",

        parts_amount, items_type)

except (ValueError, ZeroDivisionError):
    print("Improper value was obtained")
```

В силу того, что ошибки являются объектами соответствующих типов (подробнее об определении объекта будет рассказано в следующих модулях), их можно присвоить переменным с помощью ключевого слова «as»:

```
try:
    x = 1/0


except Exception as ex:
    print(ex)
```

В этом примере объект ошибки присваивается переменной «ex». Результатом выполнения данного кода будет вывод дополнительной информации об ошибке, хранящейся в переменной. Используя ключевое слово «raise», можно вручную указать, какое сообщение будет храниться в объекте исключения:

```
raise ValueError("You made a mistake entering a value")
```

Кроме того, используя переменную, в которой хранится объект исключения, можно получить точную информацию о типе исключения:

```
try:
    raise ValueError

except BaseException as ex:
    print(type(ex).__name__)
```

Приведенный код демонстрирует, что при выполнении инструкций блока «except», на консоль будет выведена строка «ValueError», которая представляет тип возникшего исключения.

Подытожим все полученные знания и совместим их в одном примере:

```
try:
    apples = int(input("Enter the amount of apples
                you have: "))

    if apples < 0:
        raise Exception("You can't have -10 apples")
    parts_number = int(input("Enter the number of parts:"))
    parts_amount = apples / parts_number
```

```
    print("You have " + str(apples) + " apples \n")
    print("Each of " + str(parts_number) +
          " parts consists of " +
          str(parts_amount) + " apples")
except (ZeroDivisionError, ValueError):
    print("Improper value was obtained")
except Exception as ex:
    print(ex)
except:
    print("Hmm... Something went wrong")
finally:
    print("The program has finished")
```

Подробнее об исключениях в ООП

Объектная природа исключений в Python делает их очень гибким инструментом, отвечающим особым потребностям и даже тем, о которых вы еще не знаете.

Прежде чем мы погрузимся в объектно-ориентированные тонкости исключений, мы хотели бы показать вам некоторые синтаксические и семантические аспекты того, как Python обрабатывает блок try-except, поскольку в предыдущих модулях мы обсудили не все.

Первая особенность, о которой мы хотим сейчас поговорить, — это дополнительная ветка, которую можно поместить внутри (или, скорее, непосредственно позади) блока try-except. Это та часть кода, которая начинается с else, как в примере ниже.

```
def reciprocal(n):
    try:
        n = 1 / n
    except ZeroDivisionError:
        print("Division failed")
        return None
    else:
        print("Everything went fine")
        return n
```

```
print(reciprocal(2))
print(reciprocal(0))
```

Код, помеченный таким образом, выполняется, если (и только в этом случае) внутри блока try: не было сгенерировано ни одной ошибки. Можно сказать, что после блока try: может быть выполнена ровно одна ветка: либо та, которая начинается с except (не забывайте, что их может быть две и больше), либо та, которая начинается с else.

Примечание: *блок else: должен идти после последнего блока except.*

Код в примере выводит следующий результат:

```
Everything went fine
0.5

Division failed
None
```

Блок try-except можно расширить еще одним способом — добавив часть с ключевым словом finally (это должна быть последняя ветка кода, который занимается обработкой исключений).

Примечание: *эти два варианта (else и finally) никак не зависят друг от друга и могут сосуществовать или генерироваться независимо друг от друга.*

Блок finally всегда выполняется (он завершает выполнение блока try-except, о чем и говорит его имя) вне зависимости от того, что произошло ранее, даже при возникновении исключения и независимо от того, было ли оно обработано.

Посмотрите на код ниже.

```
def reciprocal(n):
    try:

        n = 1 / n

    except ZeroDivisionError:

        print("Division failed")

        n = None

    else:

        print("Everything went fine")

    finally:

        print("It's time to say goodbye")

        return n


print(reciprocal(2))
print(reciprocal(0))
```

Результат вывода следующий:

```
Everything went fine
It's time to say good bye

0.5

Division failed

It's time to say good bye
None
```

Исключения — это классы

В предыдущих примерах мы вполне были довольны тем, что обнаруживали определенные типы исключений и реагировали на них соответствующим образом. А сейчас мы хотим пойти дальше и заглянуть внутрь самого исключения.

Вы, наверное, даже не удивились, когда узнали из названия выше, что исключения являются классами. Кроме того, когда возникает исключение, создается объект класса, который проходит все уровни выполнения программы в поисках ветки except, которая готова разобраться с этим исключением.

Такой объект несет некоторую полезную информацию, которая может помочь вам точно определить все нюансы рассматриваемой ситуации. Для достижения этой цели в Python есть специальный вариант описания исключения, который приведен ниже.

```
try:
    i = int("Hello!")

except Exception as e:

    print(e)
    print(e.__str__())
```

Как видите, оператор except расширен и содержит дополнительную фразу, начинающуюся с ключевого слова as, за которым следует идентификатор. Идентификатор перехватывает объект исключения, чтобы вы могли проанализировать, откуда он взялся, и сделать правильные выводы. **Примечание:** *область действия идентификатора распространяется только на блок except и не более.*

В примере представлен очень простой способ использования полученного объекта — вам просто нужно вывести его (как видно в примере, результат создается методом __str__() этого объекта) и содержит краткую информацию с описанием причины.

Такое же сообщение будет выведено, если в коде нет подходящего блока except, а Python будет вынужден обрабатывать его в одиночку.

Все встроенные исключения в Python образуют иерархию классов. Но вы всегда можете расширить ее, если посчитаете нужным.

Посмотрите на код ниже.

```
def printExcTree(thisclass, nest = 0):
    if nest > 1:

        print("    |" * (nest - 1), end="")
```

```
        if nest > 0:
            print("   +---", end="")

        print(thisclass.__name__)

        for subclass in
    thisclass.__subclasses__():
            printExcTree(subclass, nest + 1)

    printExcTree(BaseException)
```

Эта программа выводит все готовые классы исключений в виде дерева.

Потому что дерево — прекрасный пример рекурсивной структуры данных, а рекурсия кажется лучшим инструментом для обхода дерева. Функция printExcTree() принимает два аргумента:

- точка внутри дерева, с которой мы начинаем обходить дерево;
- уровень вложенности (мы будем использовать его для построения упрощенного представления ветвей дерева).

Давайте начнем с корня дерева — корнем классов исключений является класс BaseException (это суперкласс для всех других исключений).

Для каждого из обнаруженных классов выполните один и тот же набор операций:

- выведите его имя, которое нужно взять из свойства __name__;
- выполните обход в списке подклассов, который находится в методе __subclasses __ (), и рекурсивно вызовите функцию printExcTree(), увеличивая, таким образом, уровень вложенности.

Обратите *внимание на то, как мы показали ветки и вилки. Результат вывода никак не сортируется, так что вы можете попытаться отсортировать его самостоятельно, если вам интересно проверить свои силы. Кроме того, некоторые ветки представлены с небольшими неточностями. Это тоже можете исправить при желании.*

Вот как это выглядит:

```
    BaseException
    +---Exception

    | +---TypeError

    | +---StopAsyncIteration

    | +---StopIteration

    | +---ImportError

    | | +---ModuleNotFoundError

    | | +---ZipImportError
```

```
    | +---OSError

    | | +---ConnectionError
    | | | +---BrokenPipeError
```

```
    | | | +---ConnectionAbortedError
    | | | +---ConnectionRefusedError
    | | | +---ConnectionResetError
    | | +---BlockingIOError
    | | +---ChildProcessError
    | | +---FileExistsError
    | | +---FileNotFoundError
    | | +---IsADirectoryError
    | | +---NotADirectoryError
    | | +---InterruptedError
    | | +---PermissionError
    | | +---ProcessLookupError
    | | +---TimeoutError
    | | +---UnsupportedOperation
    | | +---herror
    | | +---gaierror
    | | +---timeout
    | | +---Error
    | | | +---SameFileError
    | | +---SpecialFileError
    | | +---ExecError
    | | +---ReadError
    | +---EOFError
    | +---RuntimeError
    | | +---RecursionError
    | | +---NotImplementedError
    | | +---_DeadlockError
    | | +---BrokenBarrierError
    | +---NameError
    | | +---UnboundLocalError
    | +---AttributeError
    | +---SyntaxError
    | | +---IndentationError
    | | | +---TabError
    | +---LookupError
    | | +---IndexError
```

```
    | | +---KeyError
    | | +---CodecRegistryError
    | +---ValueError
    | | +---UnicodeError
    | | | +---UnicodeEncodeError
    | | | +---UnicodeDecodeError
    | | | +---UnicodeTranslateError
    | | +---UnsupportedOperation
    | +---AssertionError
```

```
    | +---ArithmeticError
    | | +---FloatingPointError
    | | +---OverflowError
    | | +---ZeroDivisionError
    | +---SystemError
    | | +---CodecRegistryError
    | +---ReferenceError
    | +---BufferError
    | +---MemoryError
    | +---Warning
    | | +---UserWarning
    | | +---DeprecationWarning
    | | +---PendingDeprecationWarning
    | | +---SyntaxWarning
    | | +---RuntimeWarning
    | | +---FutureWarning
    | | +---ImportWarning
    | | +---UnicodeWarning
    | | +---BytesWarning
    | | +---ResourceWarning
    | +---error
    | +---Verbose
    | +---Error
    | +---TokenError
    | +---StopTokenizing
    | +---Empty
    | +---Full
```

```
    | +---_OptionError
    | +---TclError
    | +---SubprocessError
    | | +---CalledProcessError
    | | +---TimeoutExpired
    | +---Error
    | | +---NoSectionError
    | | +---DuplicateSectionError
    | | +---DuplicateOptionError
    | | +---NoOptionError
    | | +---InterpolationError
    | | | +---InterpolationMissingOptionError
    | | | +---InterpolationSyntaxError
    | | | +---InterpolationDepthError
    | | +---ParsingError
    | | | +---MissingSectionHeaderError
    | +---InvalidConfigType
    | +---InvalidConfigSet
    | +---InvalidFgBg
    | +---InvalidTheme
    | +---EndOfBlock
    | +---BdbQuit
    | +---error
    | +---_Stop
    | +---PickleError
```

```
    | | +---PicklingError
    | | +---UnpicklingError
    | +---_GiveupOnSendfile
    | +---error
    | +---LZMAError
    | +---RegistryError
    | +---ErrorDuringImport
    +---GeneratorExit
    +---SystemExit
    +---KeyboardInterrupt
```

Подробная анатомия исключений

Давайте подробнее рассмотрим объект исключения, поскольку здесь есть действительно интересные тонкости (мы вернемся к этой проблеме, когда будем рассмотрим базовые методы ввода-вывода в Python, поскольку их подсистема исключений немного расширяет эти объекты).

Класс BaseException вводит свойство args. Это кортеж, предназначенный для сбора всех аргументов, переданных конструктору класса. Если конструкция была вызвана без каких-либо аргументов, то этот конструктор пустой. Или содержит только один элемент, если конструктор получил один аргумент (аргумент self здесь не учитывается) и так далее.

Мы подготовили простую функцию, которая элегантно выведет свойство args. Посмотрите на функцию ниже.

```
def printargs(args):
    lng = len(args)

    if lng == 0:

        print("")

    elif lng == 1:

        print(args[0])

    else:

        print(str(args))



try:

    raise Exception

except Exception as e:

    print(e, e.__str__(), sep=' : ' ,end='
: ')

    printargs(e.args)



try:
```

```
        raise Exception("my exception")


    except Exception as e:
        print(e, e.__str__(), sep=' : ', end='
    : ')
        printargs(e.args)



    try:
        raise Exception("my", "exception")



    except Exception as e:
        print(e, e.__str__(), sep=' : ', end='
    : ')
        printargs(e.args)
```

Мы использовали функцию для печати содержимого свойства args в трех разных случаях, где исключение класса Exception вызывается тремя разными способами. Мы также вывели сам объект и результат вызова __str__().

Первый случай выглядит совершенно обычно — есть только имя Exception, которое следует за ключевым словом raise. Это означает, что объект этого класса был создан самым обычным способом.

Второй и третий случаи могут показаться немного странными на первый взгляд, но на самом деле здесь нет ничего необычного — это всего лишь вызовы конструктора. Во втором выражении raise конструктор вызывается с одним аргументом, а в третьем — с двумя.

Как видите, выходные данные программы демонстрируют это через соответствующее содержимое свойства args:

```
    : :
    my exception : my exception : my exception

    ('my', 'exception') : ('my', 'exception')
    :
                            ('my', 'exception')
```

Как создать собственное исключение

Иерархия исключений не является ни закрытой, ни законченной, и вы всегда можете расширить ее, если необходимо создать свой мир, заполненный вашими исключениями.

Это может пригодиться при создании сложного модуля, который обнаруживает ошибки и вызывает исключения, и вам нужно, чтобы эти исключения отличались от стандартных исключений в Python.

Для этого нужно задать новые исключения в качестве подклассов, наследующих стандартные классы.

Примечание: *если нужно создать исключение, которое будет использоваться как специальный случай любого встроенного исключения, то его нужно наследовать только из этого исключения. Если вам нужно построить собственную иерархию, и вы не хотите,*

чтобы она была тесно связана с деревом исключений Python, выведите его из любого верхнего класса исключений, например, Exception.

Представьте, что вы создали совершенно новую арифметику, которая подчиняется вашим собственным законам и теоремам. Понятно, что деление тоже изменится и должно будет вести себя не так, как обычное деление. Также ясно, что это новое деление должно вызвать свое собственное исключение, отличное от встроенного ZeroDivisionError. Но разумно также предположить, что в некоторых обстоятельствах вы (или пользователь) можете рассматривать все деления на ноль одинаково.

Подобные требования можно выполнить представленным ниже способом. Посмотрите на код ниже.

```
class
MyZeroDivisionError(ZeroDivisionError):
    pass

def doTheDivision(mine):
    if mine:
        raise    MyZeroDivisionError("some
worse news")
    else:
        raise ZeroDivisionError("some bad
news")

for mode in [False, True]:
    try:
        doTheDivision(mode)
    except ZeroDivisionError:
        print('Division by zero')

for mode in [False, True]:
    try:
        doTheDivision(mode)
    except MyZeroDivisionError:
        print('My division by zero')
    except ZeroDivisionError:
        print('Original division by zero')
```

И давайте проанализируем его:

- Мы определили наше собственное исключение MyZero DivisionError, полученное из встроенного ZeroDivisionError. Как видите, мы решили не добавлять новые компоненты в класс.

В зависимости от того, что вам нужно, исключение этого класса можно рассматривать как обычное ZeroDivisionError или как особый случай.

- Функция doTheDivision() генерирует либо MyZeroDivisionError, либо ZeroDivisionError, в зависимости от значения аргумента.

Функция вызывается в общей сложности четыре раза, в то время как первые два вызова обрабатываются с использованием только одного (более общего) блока except, а два последние с двумя разными блоками, которые способны различать исключения (не забывайте: порядок блоков имеет принципиальное значение!).

Если вы строите совершенно новую вселенную, наполненную совершенно новыми существами, которые не имеют ничего общего с привычными нам, то тут не обойтись без собственной структуры исключений.

Например, если вы работаете с большой системой моделирования, которая предназначена для моделирования деятельности пиццерии, скорее всего, придется сформировать отдельную иерархию исключений.

Вы можете начать с определения общего исключения в качестве нового базового класса для любого другого специального исключения. Например, вот так:

```
class PizzaError(Exception):
def __init__(self, pizza, message):

Exception.__init__(message)
self.pizza = pizza
```

Примечание: *мы будем собирать более конкретную информацию, чем обычное исключение Exception, поэтому наш конструктор будет принимать два аргумента:*

- первый устанавливает пиццу в качестве субъекта процесса,
- а второй содержит более или менее точное описание проблемы.

Как видите, мы передаем второй параметр конструктору суперкласса и сохраняем первый в нашем собственном свойстве.

Более конкретная проблема (например, избыток сыра) может потребовать и более конкретного исключения. Можно вывести новый класс из готового класса PizzaError, как мы сделали здесь:

```
class TooMuchCheeseError(PizzaError):
def    __init__(self,    pizza,    cheese,
message):

PizzaError._init__(self, pizza, message)
self.cheese = cheese
```

Исключению TooMuchCheeseError нужно больше информации, чем обычному PizzaError, поэтому мы добавляем его в конструктор. Затем сохраняем имя cheese для дальнейшей обработки.

Посмотрите на код ниже.

```
class PizzaError(Exception):
    def __init__(self, pizza, message):

        Exception.__init__(self, message)

        self.pizza = pizza



class TooMuchCheeseError(PizzaError):

    def   __init__(self,   pizza,   cheese,
message):

        PizzaError.__init__(self,   pizza,
message)
        self.cheese = cheese
```

```
def makePizza(pizza, cheese):
    if pizza not in ['margherita',
'capricciosa',
                     'calzone']:
        raise PizzaError(pizza,
                         "no such pizza on
the menu")

    if cheese > 100:
        raise TooMuchCheeseError(pizza,
cheese,
                                 "too
much cheese")

    print("Pizza ready!")

for (pz, ch) in [('calzone', 0),
('margherita', 110),
                 ('mafia', 20)]:
    try:
        makePizza(pz, ch)
    except TooMuchCheeseError as tmce:
        print(tmce, ':', tmce.cheese)
    except PizzaError as pe:
        print(pe, ':', pe.pizza)
```

Мы объединили два ранее созданных исключения и использовали их для работы с небольшим фрагментом кода.

Один из них генерируется внутри функции makePizza(), если обнаруживается любая из этих двух ошибок: неправильный запрос на пиццу или запрос на слишком большое количество сыра. **Примечание:**

- *если удалить ветку, которая начинается с except TooMuchCheeseError, это приведет к тому, что все исключения будут классифицированы как PizzaError;*
- *если удалить ветку, которая начинается с except PizzaError, то ошибки TooMuchCheeseError останутся необработанными и приведут к завершению программы.*

Предыдущее решение элегантное и эффективное, но имеет один важный недостаток. Из-за несколько ленивого способа объявления конструкторов, новые исключения нельзя использовать как есть, без полного списка необходимых аргументов.

Мы устраним этот недостаток, установив значения по умолчанию для всех параметров конструктора. Посмотрите:

```
class PizzaError(Exception):
    def __init__(self, pizza='uknown',
message=''):
        Exception.__init__(self, message)
        self.pizza = pizza


class TooMuchCheeseError(PizzaError):
    def __init__(self, pizza='uknown',
cheese='>100',
                 message=''):
```

```
        PizzaError.__init__(self,  pizza,
message)
        self.cheese = cheese

def makePizza(pizza, cheese):
    if  pizza  not  in  ['margherita',
'capricciosa',
                     'calzone']:
        raise PizzaError
    if cheese > 100:
        raise TooMuchCheeseError
    print("Pizza ready!")

for  (pz,  ch)  in  [('calzone',  0),
('margherita', 110),
                 ('mafia', 20)]:
    try:
        makePizza(pz, ch)

    except TooMuchCheeseError as tmce:
        print(tmce, ':', tmce.cheese)
    except PizzaError as pe:
        print(pe, ':', pe.pizza)
```

Теперь, если условия позволяют, можно использовать только имена классов.