```
1       x = int(input("Enter number"))
2       y = int(input("Enter divider"))
3       if y == 0:
4           print "Divider can't be zero"
5       else:
6           print "{}/{} == {}".format(x, y, x / y)
```

```
1       def get_params():
2           x = int(input("Enter number"))
3           y = int(input("Enter divider"))
4           return x, y
5
6       def get_result(x, y):
7           if y == 0:
8               return None
9           else:
10              return x / y
11
12      def main():
13          x, y = get_params()
14          res = get_result()
15          if res is None:
16              print "Wrong data"
17          else:
18              print "res =", res
```

Модуляризация программы приводит к разнесению точки, где можно обнаружить ошибку и точки, где ее можно обработать. Задача - найти общий метод доставки информации о ошибке от точки обнаружения до точки обработки.

70е (а также 2013 - haskell, fp scala, Go & Co)

- Для каждой функции выделить специальное значение - признак ошибки;
- Проверять результат каждой функции, обрабатывать ошибку или передавать ее дальше, если обработка в текущей точке невозможна;
- Как вариант функция всегда возвращает код ошибки, а настоящий результат идет по ссылке (POSIX/MSAPI).

```
1      def do_some_work(name, vals):
2          if not isinstance(name, basestring):
3              return None
4          #......
5          return res
6
7      def f2():
8          res = do_some_work("1231", [1, 2])
9          if res is None:
10             return None
11         .....
```

Проблемы

- Нужно помнить какое значение возвращает функция при ошибке (MSAPI);
- Нужно хранить дополнительно информацию о ошибке;
- Информация о контексте ошибки ограничена.

Решение

- Возвращать из каждой функции тройку (is_ok, traceback_or_None, result_or_error);
- Превратить return res в return (True, None, result);
- Превратить return err в return (False, [curr_func_name], err_description);
- В обработчиках ошибок нужно фильтровать ошибки по типу;
- Проверять на выходе из каждой функции результат;
- Если не проверить ошибку, то проблема возникнет в непредсказуемом месте кода.

```
    def do_some_work(name, vals):
        if not isinstance(name, basestring):
            return (False, ["do_some_work"], "name should be a s
        #......
        return (True, None, res)

    def f2():
        is_ok, stack, res = do_some_work("1231", [1, 2])
        if not is_ok:
            return (False, stack + ["f2"], res)
        .....
```

Уже лучше, но:

- Загрязняет код;
- Для каждого вызова отдельная строка и свой if;
- Такие ошибки часто не проверяются (printf);
-
- Вспомагательный код очень простой - его генерацию можно переложить на компилятор.

Именно это и делают современные языки

Исключения

Исключения

- Исключение – это событие, после которого дальнейшее продолжение работы в данной точке не возможно. По итогу такого события генерируется объект-исключение, и исполнение передается соответствующему обработчику ошибок;
- Пример – деление на 0, выбрасывается ошибка ZeroDivisionError;
- Исключения помогают упростить код, убрав из него множество проверок и значительно облегчить восстановление программы после сбоя;
- Типы всех исключений наследуют Exception (кроме KeyboardInterrupt, GeneratorExit, SystemExit);
- Чаще всего принимают строку как параметр.

Исключения

```
1       try:
2           block1
3       except tp2 as var2:
4           block2
5       except (tp3, tp4) as var3:
6           block3
7       else:
8           block5
9       finally:
10          block4
```

Исключения

```
1       try:
2           raise tp2("xxx")  # <<<<
3       except tp2 as var2:
4           block2            # <<<<
5       except (tp3, tp4) as var3:
6           block3
7       else:
8           block5
9       finally:
10          block4            # <<<<
```

Исключения

```
1       try:
2           pass                # <<<
3       except tp2 as var2:
4           block2
5       except (tp3, tp4) as var3:
6           block3
7       else:
8           block5              # <<<
9       finally:
10          block4              # <<<
```

```
1       def f1(t, d, x, y):
2           if t - d  == 0:
3               return None
4           else:
5               t1 = ((x + y) / (t - d))
6               if t1 == 0:
7                   return None
8               else:
9                   return 1 / ((x + y) / (t - d))
10
11      def f2(t, d, x, y):
12          try:
13              return 1 / ((x + y) / (t - d))
14          except ZeroDivisionError:
15              return None
```

Исключения. raise

- raise ExceptionType (....) порождает исключение
- ExceptionType должно наследовать Exception
- raise без параметров разрешено только в блоке except. При этом повторно выбрасывается текущее исключение

```
1       try:
2           func()
3       except Exception:
4           print "func cause exception"
5           raise
```

Стандартные исключения

Исключения. traceback

В обработчике исключения sys.exc_info() возвращает тройку (Тип исключения, Объект исключения, Состояние Стека)

```
1       try:
2           raise ValueError("ddd")
3       except Exception as x:
4           tb = sys.exc_info()[2]
5
6       print tb.tb_frame # <frame at 0x....>
7       print tb.tb_frame.f_lineno # 4
8       print tb.tb_frame.f_code.co_name # '<module>'
9       print tb.tb_frame.f_code.co_filename
10          # '<ipython-input-7-492d537cf800>'
11      print tb.tb_next # <frame at 0x....> or None
12      del tb
```

Гарантии безопасности исключений

```
1       class TimedCallbackStack(object):
2           def __init__(self, tf):
3               self.cb_list = []
4               self.tf = tf
5               self.sum_time = 0.0
6
7           def pop_and_exec(self):
8               func = self.cb_list.pop()
9               t = self.tf()
10              res = func()
11              self.sum_time += self.tf() - t
12              return res
```

Другие проблемы исключений

- Выполнение функции может прерваться в любой точке - нужно работать со всеми ресурсами через try/finally или, лучше через with.
- try/finally/with делают из линейного кода вложенный
- Для не локальных объектов все печально

- Все равно нужно помнить какие исключения порождает конкретная функция (хотя все не так плохо, как с "исключениями" из 70х)

```
def x1(a, b):
    if 0 == b:
        return (False,
                ["x1"],
                ZeroDiv)

def x2(a, b):
    print a,b
    ok, res, stack = x1(a, b)
    if not ok:
        return (False,
                stack + \
                ["x2"],
                res)

ok, stack, res = x1(a, b)
if not ok and res is ZeroDiv:
    #process
```

```
def x1(a, b):
    if 0 == b:
        raise ZeroDivisionError()


def x2(a, b):
    print a,b
    return x1(a, b)


try:
    res = x1(a, b)
except ZeroDivisionError:
    #process
```