Exception Handling With Recovery

ACCU 2006/Python UK, Oxford

Michael Hudson mwh@python.net
Heinrich-Heine-Univeristät Düsseldorf

What is an Exception, anyway?

- A surprisingly hard question – like the quote about pornographic films, you know them when you see them*
- There is not going to be one obviously correct answer, so what do you do?
- I asked Google

*Justice Potter Stewart, Jacobellis v. Ohio, 878 U.S. 184 (1964)

What is an Exception, anyway?

Typing ‘define: "exceptional situation"’ into Google found the Dylan Reference Manual* and told me that an exceptional situation is:

> *A situation that is not conceptually part of the normal execution of the program*

which works for me.

* http://www.gwydiondylan.org/books/drm/

Examples

- Common exceptional situations:
  - divide-by-zero
  - trying to open a non-existent file
  - many programming errors, like trying to access an invalid field of an object

What do I mean by "with Recovery"?

- The main way code we write today copes with situations it doesn't understand is to give up in some sense: call abort(), raise an exception
- This isn't the only possibility – could also ask someone/something else what to do
- (Demo)

So I've defined my terms…

- Around 18 months ago, I started to think about what it would take to add a facility like this to Python
- Came up with some fearsomely complicated patches to CPython
- Proposed a technical talk to describe them at EuroPython, to be held in June 2005
- Then thought and read a bit more...

Going Soft

- What I realized was that there are actually no *technical* reasons that we can't have exception handling with recovery in Python today
- My technical talk rapidly became a "softer," more software engineering-y talk
- So the main point of this talk is to convey some ideas and provoke discussion

Recovering from the Unexpected

- The standard way to react to an exceptional situation in Python is to raise an Exception
  - i.e. transfer the control to another part of the program after unwinding the stack
- To enable recovery we want to transfer the control to another part of the program *without* unwinding the stack
  - we can do this by just calling a function!

Recovering from the Unexpected

So instead of writing this:

```
raise ZeroDivisionError()
```

we should write this:

```
r = handle(ZeroDivisionError())
```

(and do something useful with `r`, of course)

Implementing `handle()`

- One implementation would be just this:

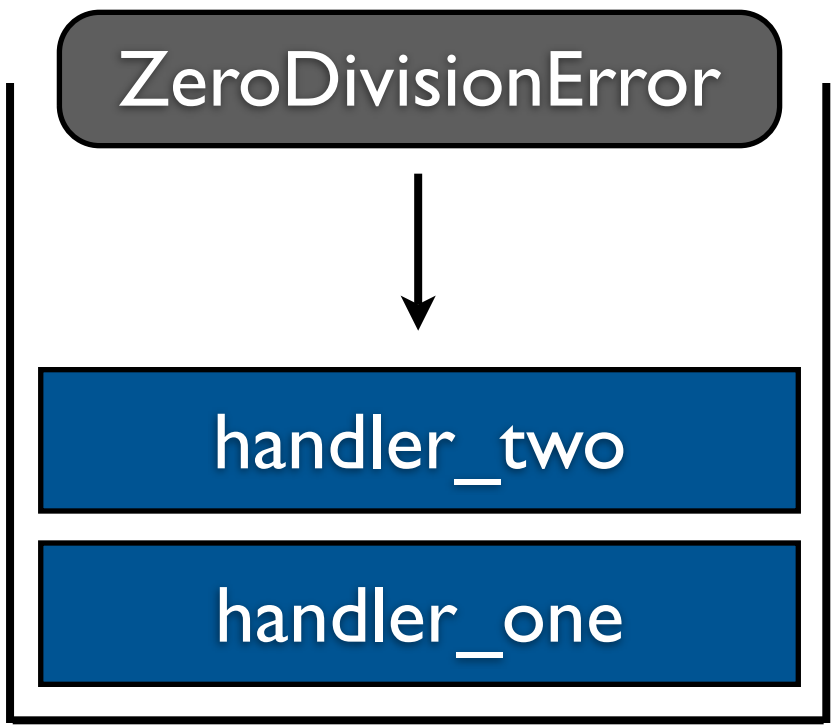
```
def handle(exc):
    raise exc
```

- This wouldn't get us very far – just the same as CPython today, in fact

Implementing `handle()`

- It is likely that we'll want to handle exceptions differently in different parts of the program, like `try:...except:...` blocks today
- Want to maintain a stack of handlers, and allow them to consider exceptions inside out

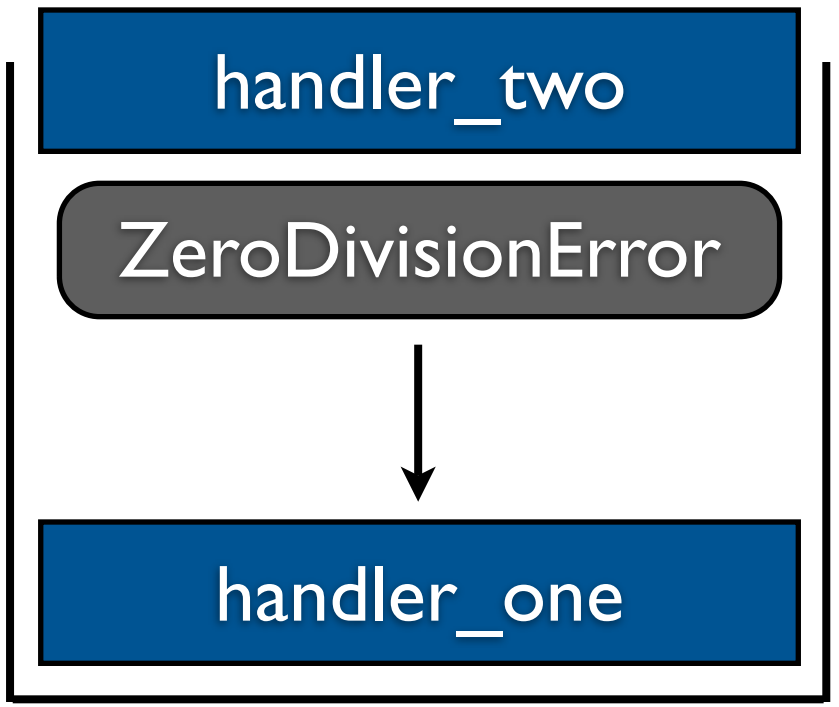
Implementing `handle()`



handler_two gets the first chance to consider the exception

the stack of handlers

Implementing `handle()`



if it declines, move
on to handler_one

the stack of handlers

Implementing `handle()`

```
NOT_HANDLED = object() # just a marker
_handlers = []

def handle(exc):
    for h in reversed(_handlers):
        r = h(exc)
        if r is not NOT_HANDLED:
            return r
    raise exc
```

Implementing a handler

A handler gets an exception it can inspect, for example:

```
def no_stop_on_divide(exc):
    if isinstance(exc, ZeroDivisionError):
        return "infinity"
    else:
        return NOT_HANDLED
```

Making `handle()` useful

- We need a nice way of pushing and popping handlers
- ```
  _handlers.append(a_handler)
  try:
      ...
  finally:
      _handlers.pop()
  ```
  works, but is a bit verbose
  ```

Making `handle()` useful

- A new feature in Python 2.5 (the 'with' statement, PEP 343) lets us write this as:
- ```
  with handler(a_handler):
      ...
  ```
  (for a suitable definition of `handler`, of course)
- (Demo)
  ```

Restarts

- In the code presented so far, the handler doesn't have many options; essentially it can return a value or re-raise the exception
- Common Lisp has the concept of a *restart*, a named place to resume execution from
- It is then the responsibility of the handler to choose among restarts

Restarts in Python

- It is possible to implement this idea in a way which lets one write code like this:

```
def restartable_divide(a, b):
    with restart(“take infinity”) \
            as take_infinity:
        try:
            return divide(1, 0)
        except take_infinity:
            return “infinity”
```

Restarts in Python

- In a system with restarts, it makes sense for the default handler to ask the user (developer) to ask which restart to choose
- Demo
- Also possible to allow restarting with a value, with the value supplied either interactively or programmatically

Design Considerations

- In the previous few slides, I've shown a very simple facility for handling exceptions with recovery in (pure!) Python
- It's probably too simple to be useful, but it has another problem: nothing uses it
  - Basically all code written today 'handles' exceptional situations by raising

Design Considerations

- It is technically possible to make it possible to resume from all exceptions in CPython but the code raising the exception does not expect control flow to come back after a `raise` statement
- This leads the way to my next point…

A Slogan

Exception Handling Is Primarily a Protocol Matter

(from Kent Pitman's paper"Condition Handling in the Lisp Language Family", 2001 – although he uses the term "condition" where I've been using the term "exception")

Protocol?

- Another word with a fuzzy definition...
- From the OED this time (sense 5(c)):

  any code of conventional or proper conduct; formally correct behaviour.

- Also means a way for parties that do not completely understand each other to communicate

Protocol

- In software terms, “like an interface but a bit more so”
- Protocol in this sense includes things like “call method1() before method2()”, “don’t call func() if stdin isn’t attached to a terminal” and “how the function behaves in exceptional situations”

The Need For Protocols

- The need for protocols arises when pieces of code from more than one source need to interoperate
- This is why examples in conference talks *always* seem a bit forced: in simple situations, and in particular if you wrote all the code involved yourself, there is no need to follow any particular protocol on, for example, handling exceptional situations

Python's Exception Handling Protocol

- So to rephrase an earlier point, almost all Python code in existence uses the 'raise or return' exception handling protocol
- Certainly, none of it uses the protocol outlined earlier in the talk!

Trying to Change the World

- I could release the code I showed earlier as a small library
- In all likelihood, no-one would use it
  - As outlined above, there would be little benefit to a single package from using my library
  - The advantage (if any) comes at integration time, and that's a harder sell

Trying to Change the World

- An even more ridiculous situation would be for there to be multiple competing protocols for exception handling with recovery
- There should probably only be one such protocol in any given program

So… what?

- One position I could take is to campaign for the introduction of a facility like the one I described earlier to Python, in other words to extend Python's default exception handling protocol
- I'm not sure I want to do this, but let's have a look at the arguments

Arguments for

- Should lead to more robust, more flexible software
- If it's going to be done, it should be done centrally – the protocol thing
  - Also Python's own handling of exception situations (NameError?) should participate
- One less reason for Common Lispers to be smug :-)

Arguments for

- Should allow for a generalized approach for the parts of standard Python that already have a complex approach to handling exceptional situations, for example:
  - unicode error callbacks (PEP 293)
  - warnings
  - `shutil.rmtree`
- A little CPython hacking would make it more useful

Arguments against

- It's a change, and change always has cost
- Adds complexity
- Writing a library that sets up appropriate and useful restarts is hard, time-consuming and in practice people don't seem to do it
  - Not very XP!

Arguments against

- The lack of a ‘fix and continue’ facility reduces the usefulness of the feature
- Will probably trigger a ~~huge flamewar~~ lively discussion on python-dev about a comfortable syntax

In The Mean Time…

- Notice that there are no technical limitations preventing the adoption of restartable exceptions
- Be aware that in an exceptional situation that there are more options than "pretend it didn't happen" and "give up"
  - For example, in a batch script you could ask the user interactively if some data is missing (I actually do this now!)

Credits

- I was heavily inspired by two papers by Kent Pitman:

  "Condition Handling in the Lisp Language Family"

  "Exceptional Situations in Lisp"

- Both are available on the web at

http://www.nhplace.com/kent/Papers/index.html

Credits

- I also bent the ear of various people in various IRC channels, particularly #lisp on freenode and #smalltalk on irc.parcplace.net

Questions?

Thanks for listening

Slides and example code will appear shortly at

http://python.net/crew/mwh/accu2006/