Error Handling in the Real World

Miguel Grinberg
@miguelgrinberg

About Me

- Software Engineer at SDVI Corporation
- Author of the O'Reilly book "Flask Web Development"
- Author of several Python open source packages
- Blogger at http://blog.miguelgrinberg.com
- Gave several PyCon talks and tutorials on Flask and REST APIs
    - Preparing an Advanced Flask tutorial for PyCon 2016!

What is There to Say About Error Handling?

- What everybody talks about...
  - The two approaches to error handling

- What nobody talks much about...
  - How to report errors so that others can handle them
  - How to write maintainable error handling code
  - How to avoid error handling code from complicating application logic
  - Error handling during development vs. production

High-Level Error Handling Approaches

LBYL

Look Before You Leap

```
if can_i_do_x():
    do_x()
else:
    # handle error
```

- Not always clear what needs to be “looked” before the “leap”.
- A race condition can occur between the “looking” and the “leaping”.
- While this is common practice in other languages, it is not considered Pythonic.

High-Level Error Handling Approaches

LBYL

Look Before You Leap

```
if not can_i_do_x():
    do_x()
else:
    # handle error
```

EAFP

Easier to Ask Forgiveness than Permission

```
try:
    do_x()
except SomeError:
    # handle error
```

- Not always clear what errors need to be handled.
- try/except blocks lead to code that is harder to read.

Problems Neither Approach Addresses

- How to report errors
- Production Needs
  - All errors should be handled so that the application never crashes.
  - Backtraces should be logged and never shown to users.
- Development Needs
  - Errors should not be handled, so that the developer sees them.
  - Backtraces should be shown.

High-Level Error Handling Approaches

LBYL

Look Before You Leap

```
if can_i_do_x():
    do_x()
else:
    # handle error
```

EAFP

Easier to Ask Forgiveness than Permission

```
try:
    do_x()
except SomeError:
    # handle error
```

YOLO!

```
do_x()
```

No-Nonsense Guide To Error Handling

Recoverable and Unrecoverable Errors

- Can you recover from an error?
  - If the function can continue in spite of the error, then obviously yes.
  - If the function needs to stop, then the error is unrecoverable.
- An unrecoverable error can become recoverable at a higher scope level.

Handling Recoverable Errors (I)

- If your function generates a recoverable error, then recover it and keep going.

```
def add_song(song):
    # ...
    if song.year is None:
        song.year = 'Unknown'
    # ...
```

Handling Recoverable Errors (II)

- If your function calls another function that triggers a recoverable error, then use EAFP to catch the error, recover it and keep going.

```
def export(filename):
    try:
        os.remove(filename)
    except OSError:
        pass
    # ...
```

Handling Unrecoverable Errors (I)

- If your function generates an unrecoverable error, raise an exception to alert the higher scope levels.

```
def validate_customer(customer):
    if customer.name is None or customer.name == '':
        raise ValidationError('Customer has no name')
    # ...
```

Handling Unrecoverable Errors (II)

- If your function calls another function that triggers an unrecoverable error, just call the function and let it error if it wants to. YOLO!

```
def save_customer_to_db(customer):
    validate_customer(customer)
    write_customer(customer)
```

Unrecoverable Errors: Wrong approach

```
@app.route('/songs/<id>', methods=['PUT'])
def update_song(id):
    # ...
    try:
        db.session.add(song)
        db.session.commit()
    except SQLAlchemyError:
        logger.error('failed to update song %s, %s', song.name, e)
        try:
            db.session.rollback()
        except SQLAlchemyError as e:
            logger.error('error rolling back failed create song, %s', e)
        return 'Internal Service Error', 500
    return '', 204
```

Unrecoverable Errors: Right approach

```
@app.route('/songs/<id>', methods=['PUT'])
def new_song(id):
    # ...
    db.session.add(song)
    db.session.commit()
    return '', 204
```

- The error will bubble up until it gets to Flask, which can recover it.
- Flask will write the error's stack trace to the log (or start the debugger)
- ... then it will do cleanup (including a db session rollback).
- ... then it will send a code 500 error to the client.

Sounds Great... But What's the Trick?

```
def save_customer_to_db(customer):
    try:
        validate_customer(customer)
        write_customer(customer)
    except (ValidationError, IOError) as e:
        # what do I do with e here?

if __name__ == '__main__':
    args = parser.parse_args()
    save_customer_to_db(args.customer)
```

```
def save_customer_to_db(customer):
    validate_customer(customer)
    write_customer(customer)

if __name__ == '__main__':
    args = parser.parse_args()
    try:
        save_customer_to_db(args.customer)
    except (ValidationError, IOError) as e:
        logger.exception('Error: ' + str(e))
    except Exception as e:
        logger.exception(
            'Unexpected error: ' + str(e))
```

Handling errors with “merry”

- Install with `pip install merry`

```
from errors import merry

@merry._try
def save_customer_to_db(customer):
    validate_customer(customer)
    write_customer(customer)

if __name__ == '__main__':
    args = parser.parse_args()
    save_customer_to_db(args.customer)
```

main.py

```
from merry import Merry
merry = Merry()

@merry._except(ValidationError, IOError)
def handle_error(e):
    print('Error: ' + str(e))

@merry._except(Exception)
def catch_all(e):
    print('Unexpected error: ' + str(e))
```

errors.py

Debug Mode with Merry

- With merry, debug mode is turned on or off in the constructor.

```
merry = Merry(debug=True)  # debug=True suspends exception handling
                           # and bubbles exceptions up
```

Production Mode with Merry

- Merry writes stack traces for all exceptions to a logger object.
- A catch-all handler easily prevents application crashes.

```
@merry._except(Exception)
def catch_all():
    # recover unexpected errors here
```

Summary

- Exceptions and EAFP are Pythonic. Error codes and LBYL are not.
- If an error is recoverable, recover it and move on.
- If an error is unrecoverable, let it bubble up until it becomes recoverable.
- During development, let errors crash your application.
- During production, log all errors and recover them as best you can.
- Put error handlers at the higher scope levels.
- Keep error handlers away from your application logic whenever possible.
- Try “merry” and share your feedback. Pull requests gladly accepted. :)

Thank You!

Questions?