Errors And Exceptions

The Basic Idea

An **exception**:

- Is a way to interrupt the normal flow of code
- Can be **raised** at any point. Even in the middle of a line

```
import requests
from json.decoder import JSONDecodeError

resp = requests.get(api_url)
try:
    # Parse response body as JSON
    obj = resp.json()
except JSONDecodeError:
    # Response does not encode a valid JSON object.
    obj = {}
```

Flow Control

An exception often means an error. But it doesn't have to.

```
# Use "speedyjson" if available. If not fall back to "json" in the
# standard library.

try:
    from speedyjson import load
except ImportError:
    from json import load
```

The `load` function called in code will be the best available version.

Built-In Exceptions

Most errors you see in Python are exceptions:

- `KeyError` for dictionaries
- `IndexError` for lists
- `TypeError` for incompatible types
- `ValueError` for bad values
- `NameError` for an unknown identifier

Even `IndentationError` is an exception.

Multiple Except Blocks

```
try:
    value = int(user_input)
except ValueError:
    print("Bad value from user")
except TypeError:
    print("Invalid type (probably a bug)")
```

Often useful with logging:

```
try:
    value = int(user_input)
except ValueError:
    logging.error("Bad value from user: %r", user_input)
except TypeError:
    logging.critical(
        "Invalid type (probably a bug): %r", user_input)
```

`finally` and `except`

Sometimes you have a block of code that must ALWAYS be executed, no matter what.

```
conn = open_db_connection()
try:
    do_something(conn)
finally:
    conn.close()
```

You can also have one (or more) except clauses:

```
conn = open_db_connection(CONFIG)
try:
    do_something(conn)
except ValueError:
    logging.error("Bad data read from DB")
finally:
    conn.close()
```

Cloud Computing

```
# fleet_config is an object with the details of what
# virtual machines to start, and how to connect them.
fleet = CloudVMFleet(fleet_config)
# job_config details what kind of batch calculation to run.
job = BatchJob(job_config)
# .start() makes the API calls to rent the instances,
# blocking until they are ready to accept jobs.
fleet.start()
# Now submit the job. It returns a RunningJob handle.
running_job = fleet.submit_job(job)
# Wait for it to finish.
running_job.wait()
# And now release the fleet of VM instances, so we
# don't have to keep paying for them.
fleet.terminate()
```

Imagine `running_job.wait()` raises a network-timeout exception. Now `fleet.terminate()` is never called.

Whoops. Expen$ive.

Save Your Bank Account/Job

Protect against this with `finally`:

```
# timeout is an exception type (even though it's lowercase)
from socket import timeout

fleet = CloudVMFleet(fleet_config)
job = BatchJob(job_config)
try:
    fleet.start()
    running_job = fleet.submit_job(job)
    running_job.wait()
except timeout:
    logging.error('Network timeout running job')
finally:
    fleet.terminate()
```

Exceptions Are Objects

An exception is an instance of an exception class.

Often your `except:` clause will just specify the class. But sometimes you need the actual exception object.

Catch with "as":

```
try:
    do_something()
except ExceptionClass as exception_object:
    handle_exception(exception_object)
```

Exception Object Info

Exception objects have helpful info. The attributes vary, but it will almost always have an `args` attribute.

```
# Atomic numbers of noble gasses.
nobles = {'He': 2, 'Ne': 10,
  'Ar': 18, 'Kr': 36, 'Xe': 54}
def show_element_info(element):
    print('Atomic number of {} is {}'.format(
          element, nobles[element]))
for element in ['Ne', 'Br', 'Ar']:
    try:
        show_element_info(element)
    except KeyError as err:
        missing_element = err.args[0]
        print('Missing data for element: ' + missing_element)
```

```
Atomic number of Ne is 10
Missing data for element: Br
Atomic number of Ar is 18
```

Raising Exceptions

```
def positive_int(value):
    "Converts string value into a positive integer."
    number = int(value)
    if number <= 0:
        raise ValueError("Bad value: " + str(value))
    return number
```

Focus on the `raise` line:

- `raise` takes an exception object
- You instantiate `ValueError` inline

```
>>> positive_int(-7.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in positive_int
ValueError: Bad value: -7.0
```

(What does `positive_int("not a number")` do?)

Lab: Exceptions

Lab file: `exceptions/exceptions.py`

- In labs/py3 for 3.x; labs/py2 for 2.7
- When you are done, give a thumbs up...
- ... and then do `exceptions/exceptions_extra.py`

Example: Money

Raising exceptions can lead to more understandable error situations.

Here's a `Money` class:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    def __repr__(self):
        'Improves the default representation in stack traces.'
        return "Money({},{})".format(
            self.dollars, self.cents)
    # Plus other methods.
```

```
>>> Money(187, 27)
Money(187,27)

>>> MoneyWithoutRepr(187, 27)
<__main__.MoneyWithoutRepr object at 0x10afdf470>
```

Money Factory

A factory helper function:

```
import re
def money_from_string(amount):
    # amount is a string like "$140.75"
    match = re.search(
        r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

Huh?

What happens if you pass it bad input? The error isn't very informative.

```
>>> money_from_string("$140.75")
Money(140,75)
>>> money_from_string("$12.30")
Money(12,30)
>>> money_from_string("Big money")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in money_from_string
AttributeError: 'NoneType' object has no attribute 'group'
```

Imagine finding this error deep in a stack trace. We have better things to do than decrypt this.

Better Errors

Add a check on the `match` object. If it's `None`, meaning `amount` doesn't match the regex, raise a `ValueError`.

```
import re
def money_from_string(amount):
    match = re.search(
        r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    # Adding the next two lines here
    if match is None:
        raise ValueError("Invalid amount: " + amount)
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

More Understandable

```
>>> money_from_string("$140.75")
Money(140,75)
>>> money_from_string("$12.30")
Money(12,30)
>>> money_from_string("Big money")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in money_from_string
ValueError: Invalid amount: Big money
```

This is MUCH better. The exact nature of the error is immediately obvious.

Catch And Re-Raise

In an `except` block, you can re-raise the current exception.

Just write `raise` by itself:

```
try:
    do_something()
except ExceptionClass:
    handle_exception()
    raise
```

It's a shorthand, equivalent to this:

```
try:
    do_something()
except ExceptionClass as err:
    handle_exception()
    raise err
```

Interject Behavior

One pattern this enables: inject but delegate.

```
try:
    process_user_input(value)
except ValueError:
    logging.error("Invalid user input: %s", value)
    raise
```

It enables other patterns too.

Creating directories

`os.makedirs()` creates a directory.

```
# Creates the directory "riddles", relative
# to the current directory.
import os
os.makedirs("some-directory")
```

But if the directory already exists, it raises `FileExistsError`.

```
>>> os.makedirs("some-directory")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/lib/python3.6/os.py", line 220, in makedirs
    mkdir(name, mode)
FileExistsError: [Errno 17] File exists: 'some-directory'
```

Using In Code

Suppose if that happens, we want to log it, but then continue:

```
# First version....
import os
import logging
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except FileExistsError:
        logging.error(
            "Upload dir for new user already exists")
```

This works, but the log message is not informative:

```
ERROR: Upload dir for new user already exists
```

Logging The directory

`FileExistsError` objects have an attribute called `filename`. Let's use that to create a useful log message:

```
# Better version!
import os
import logging
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except FileExistsError as err:
        logging.error("Upload dir already exists: %s",
            err.filename)
```

MUCH better:

```
ERROR: Upload dir already exists: /var/www/uploads/joe
```

OSError

`FileExistsError` is only in Python 3. In Python 2, `os.makedirs()` instead raises `OSError`.

But `OSError` can indicate many other problems:

- filesystem permissions
- a system call getting interrupted
- a timeout over a network-mounted filesystem
- And the directory already existing.. the only one we care about.

How do you distinguish between these?

errno

`OSError` objects set an `errno` attribute. It's essentially the `errno` variable from C.

The standard constant for "file already exists" is `EEXIST`:

```
from errno import EEXIST
```

Game plan:

- Optimistically create the directory.
- if `OSError` is raised, catch it.
- Inspect the exception's `errno` attribute. If it's equal to `EEXIST`, this means the directory already existed; log that event.
- If `errno` is something else, it means we don't want to catch this exception here; re-raise the error.

create_upload_dir() in 2.x

```
# How to accomplish the same in Python 2.
import os
import logging
from errno import EEXIST
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except OSError as err:
        if err.errno != EEXIST:
            raise
        logging.error("Upload dir already exists: %s",
            err.filename)
```

The Most Diabolical Python Anti-Pattern

Design Patterns are good.

Anti-Patterns are bad.

And in Python, one Anti-Pattern most harmful of all.

I wish I could not even tell you about it. But I must.

TMDPAP

Here's the most self-destructive code a Python developer can write:

```
try:
    do_something()
except:
    pass
```

This creates the worst kind of bug.

After a FULL WEEK

After a full WEEK of engineer time, I was able to isolate the bug to a single block of code:

```
try:
    extract_address(location_data)
except:
    pass
```

Why???

Why do people do this?

1) Because they expect an exception to occur that can be safely ignored.

That's fine; the problem is being overbroad. Just target narrowly instead:

```
try:
    extract_address(location_data)
except ValueError:
    pass

# Variation: Insert logging.
try:
    extract_address(location_data)
except ValueError:
    logging.info(
        "Invalid location for user %s", username)
```

Why?

2) Because a code path must continue running regardless of what exceptions are raised.

In that case, this is better:

```
import logging
def get_number():
    return int('foo')
try:
    x = get_number()
except:
    logging.exception('Caught an error')
```

logging.exception()

Example stack trace:

```
ERROR:root:Caught an error
Traceback (most recent call last):
  File "example-logging-exception.py", line 5, in <module>
    x = get_number()
  File "example-logging-exception.py", line 3, in get_number
    return int('foo')
ValueError: invalid literal for int() with base 10: 'foo'
```