Errors And Exceptions

The Basic Idea

An **exception** is a way to interrupt the normal flow of code.

An exception can be **raised** at any point. Even in the middle of a line.

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

Dessert

```
def describe_favorite(category):
    "Describe my favorite food in a category."
    favorites = {
        "appetizer": "calamari",
        "vegetable": "broccoli",
        "beverage": "coffee",
    }
    return "My favorite {} is {}.".format(
        category, favorites[category])

message = describe_favorite("dessert")
print(message)
```

No Dessert

You've seen exceptions before. Even if you don't realize it.

```
Traceback (most recent call last):
  File "favdessert.py", line 12, in <module>
    message = describe_favorite("dessert")
  File "favdessert.py", line 10, in describe_favorite
    category, favorites[category])
KeyError: 'dessert'
```

`KeyError` is an exception.

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.

Handling Exceptions

An exception needs to be *caught*. If you do not, the program exits with an error.

You catch (handle) an exception with a **try-except block**:

```
# Replace the last few lines with the following:
try:
    message = describe_favorite("dessert")
    print(message)
except KeyError:
    print("I have no favorite dessert. I love them all!")
```

New output:

```
I have no favorite dessert. I love them all!
```

How try-except Works

```
# Replace the last few lines with the following:
try:
    message = describe_favorite("dessert")
    print(message)
except KeyError:
    print("I have no favorite dessert. I love them all!")
```

- `try` block executes
- If no exception raise, `except` block skipped
- If exception raised, immediately jump to matching `except` block
- If not match, ignore `except` block, and raise to higher-level code

Inside Functions

```
def print_description(category):
    try:
        message = describe_favorite(category)
        print(message)
    except KeyError:
        print("I have no favorite {}. I love them all!".format(category))
```

```
>>> print_description("dessert")
I have no favorite dessert. I love them all!
>>> print_description("appetizer")
My favorite appetizer is calamari.
>>> print_description("beverage")
My favorite beverage is coffee.
>>> print_description("soup")
I have no favorite soup. I love them all!
```

ImportError

Exceptions are not just for errors.

```
# Always available
from json import load
# Better, but not always installed
from speedyjson import load
```

```
# If speedyjson isn't installed...
>>> from speedyjson import load
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ImportError: No module named 'speedyjson'
```

Importing something unavailable will raise `ImportError`.

Flow Control

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.

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!

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

```
try:
    line1
    line2
    # etc.
finally:
    line1
    line2
    # etc.
```

`finally` and `except`

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

```
try:
    line1
    line2
    # etc.
except FirstException:
    line1
    line2
    # etc.
except SecondException:
    line1
    line2
    # etc.
finally:
    line1
    line2
    # etc.
```

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`:

```
fleet = CloudVMFleet(fleet_config)
job = BatchJob(job_config)
try:
    fleet.start()
    running_job = fleet.submit_job(job)
    running_job.wait()
finally:
    fleet.terminate()
```

Exceptions Are Objects

An exception is an instance of an exception class.

So far, we've just been working with the type. 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 give 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(elements):
    for element in elements:
        print('Atomic number of {} is {}'.format(
              element, nobles[element]))
try:
    show_element_info(['Ne', 'Ar', 'Br'])
except KeyError as err:
    missing_element = err.args[0]
    print('Missing data for element: ' + missing_element)
```

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

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 log message:

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

Using The Built-in Exceptions

Python has many built-in exceptions. You should use some in your own code:

- `TypeError` if the value is the wrong type
- `ValueError` if the type is correct, but the value isn't valid
- `IndexError` and `KeyError` if you create your own collection class

For example:

```
>>> int("not a number")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'not a number'
```

How do you make your classes, methods and functions raise these errors?

Raising Exceptions

Use the `raise` statement.

```
raise ExceptionClass(arguments)
```

For example:

```
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 Values

How it works:

```
>>> positive_int("3")
3
>>> positive_int(7.0)
7
>>> positive_int("-3")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in positive_int
ValueError: Bad value: -3
>>> 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
```

Another Way To Raise

You can also pass an exception class directly:

```
raise IndexError
```

That's equivalent to invoking the class with no arguments:

```
raise IndexError()
```

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):
        'Renders the object nicely on the prompt.'
        return "Money({},{})".format(
            self.dollars, self.cents)
    # Plus other methods.
```

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: " + repr(amount))
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

More Understandable

```
>>> 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.info("Invalid user input: %s", value)
    raise
```

It enables other patterns too.

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.

When that's done: Open and skim through **PythonForLargeApps.pdf** - just notice what topics interest you.

File Paths Again

```
# Remember this? Python 3 code, from earlier.
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)
```

`FileExistsError` is only in Python 3. What about Python 2?

OSError

In Python 2, `os.makedirs()` 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.

To distinguish, look at the `errno`.

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'
```