Exceptions and Exception Handling

Intermediate Application Development

Otago Polytechnic
Dunedin, New Zealand
Kaiako: Tom Clark

The Problem

You may have noticed that things don't always go quite to plan when programming.

```
def get_item(i):
    stuff = [1, 2, 3, 4, 5]
    return stuff[i]
```

What could possibly go wrong?

One Solution

If the argument `i` is not a valid index for our list, we're going to get an error. We can add some tests.

```
def get_item(i):
    stuff = [1, 2, 3, 4, 5]
    if type(i) is int and 0 <= i and i < len(stuff):
        return stuff[i]
    else:
        return None
```

This is ok, but now we've devoted a good chunk of our login to handling cases that we don't expect to happen - to *exceptional* cases.

Exceptions

Many programming languages deal with this by providing *Exceptions*, a sort of built in event and event handling to deal with these error cases.

```
def get_item(i):
    stuff = [1, 2, 3, 4, 5]
    return stuff[i]
```

- If the argument `i` is not an integer, A `TypeError` is raised.
- If `i` is an integer outside the range of valid indices for our list, an `IndexError` is raised.

One Solution

If the argument `i` is not a valid index for our list, we're going to get an error. We can add some tests.

```
def get_item(i):
    stuff = [1, 2, 3, 4, 5]
    if type(i) is int and 0 <= i and i < len(stuff):
        return stuff[i]
    else:
        return None
```

This is ok, but now we've devoted a good chunk of our login to handling cases that we don't expect to happen - to *exceptional* cases.

Exception Handling

Basic exception handling is done with a `try/except` block.

```
def get_item(i):
    stuff = [1, 2, 3, 4, 5]
    try:
        return stuff[i]
    except IndexError:
        return None
```

Notice that we're not handling the possible `TypeError`. If you don't have a plan for how to recover from an exception, let it propagate.

Exception Handling

We can access the Exception object created when the error occured.

```
def get_item(i):
    stuff = [1, 2, 3, 4, 5]
    try:
        return stuff[i]
    except IndexError:
        return None
    except TypeError as e:
        logger.error(e)
        raise e
```

We also have the opportunity to take some action in the case of an exception and then re-raise it to pass it up the stack.

Full try/except structure

```
try:
    ...code...
except ErrorType:
    ...handle ErrorType...
except AnotherError as e:
    ...handle AnotherError with access to exception
else:
    ...executed if no exceptions are raised...
finally:
    ...aways executed after all other blocks complete...
```

Programming Activity

1. Pull the course materials repo.
2. Create a new branch, `05-practical` in your practicals repo.
3. Add a subdirectory, `05-practical` and copy `04-practical.ipynb` from the class materials into it.
4. Open a shell, cd to this directory, and run `jupyter notebook` to open the notebook. Complete the first questions.
5. We will discuss results in 30ish minutes.

User-defined Exceptions

1. It's generally preferable to use a built in exception when it suits the error.
2. Exceptions are just special classes.
3. Exception names typically end in `Error`.
4. A user-defined exception must derive from `Exception` or one of its subclasses.
5. You can do just about anything, but in general they are simple classes that hold information about the error.
6. User-defined exceptions must be explicitly raised in application code.

Example

```
class IN608Error(Exception):
    pass

class InputError(IN608Error):
    def __init__(self, badinput, message):
        self.input = badinput
        self.message = message

    def __str__(self):
        return f'InputError: {self.message}'
```

Conclusions

1. Exceptions let us extract error handling from core logic.
2. They are best used for handling things you don't expect to happen[1]
3. You don't have to handle every exception. It's generally bad practice to try.
4. Good reasons to handle exceptions include
   - 4.1 It's possible to recover from the error and continue execution.
   - 4.2 The error in unrecoverable, but there are important actions to complete before halting execution.

[1] There are notable counterexamples.