Lesson 6: Handling Errors

Reference:
Research Software Engineering with Python by Damien Irving, Kate Hertweck, Luke Johnston, Joel Ostblom, Charlotte Wickham, and Greg Wilson
https://merely-useful.tech/py-rse/errors.html

Two types of errors:

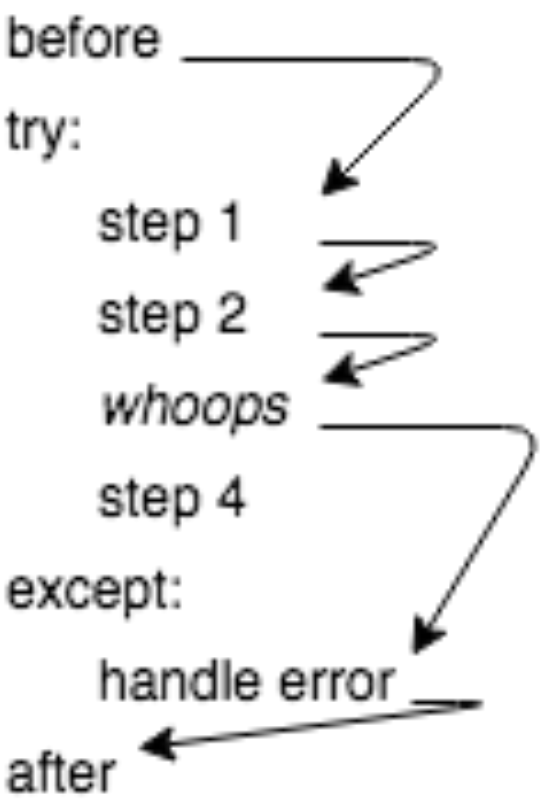
Internal error: an error in the program itself, like trying to access elements beyond the end of an array

External error: an error caused by something outside the program, like network outages or opening a file that doesn’t exist

Exceptions

```
for denom in [-5, 0, 5]:
    try:
        result = 1/denom
        print(f'1/{denom} == {result}')
    except:
        print(f'Cannot divide by {denom}')
```

```
1/-5 == -0.2
Cannot divide by 0
1/5 == 0.2
```



Exceptions

- Python (and other languages) store information about the error in an object.
- We can catch an exception and inspect it when debugging:

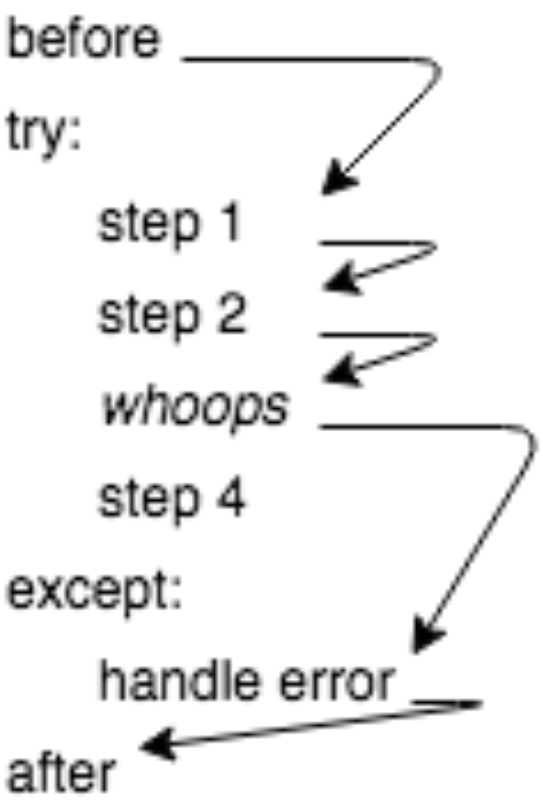
```
for denom in [-5, 0, 5]:
    try:
        result = 1/denom
        print(f'1/{denom} == {result}')
    except Exception as error:
        print(f'{denom} has no reciprocal: {error}')
```

```
1/-5 == -0.2
0 has no reciprocal: division by zero
1/5 == 0.2
```

Exceptions

```
for denom in [-5, 0, 5]:
    try:
        result = 1/denom
        print(f'1/{denom} == {result}')
    except:
        print(f'Cannot divide by {denom}')
```

```
1/-5 == -0.2
Cannot divide by 0
1/5 == 0.2
```



Try it out:

```
numbers = [-5, 0, 5]
for i in [0, 1, 2, 3]:
    try:
        denom = numbers[i]
        result = 1/denom
        print(f'1/{denom} == {result}')
    except IndexError as error:
        print(f'index {i} out of range')
    except ZeroDivisionError as error:
        print(f'{denom} has no reciprocal: {error}')
```

Raise Exceptions Explicitly:

```
for number in [1, 0, -1]:
    try:
        if number < 0:
            raise ValueError(f'no negatives: {number}')
        print(number)
    except ValueError as error:
        print(f'exception: {error}')
```

Most Common Exception Errors:

`ArithmeticError`: something has gone wrong in a calculation.

`IndexError` and **`KeyError:`** something has gone wrong indexing a list or lookup something up in a dictionary.

`OSError`: thrown when a file is not found, the program doesn't have permission to read it, and so on.

More information on Python's built-in exceptions at:
https://docs.python.org/3/library/exceptions.html#exception-hierarchy

Exceptions: Throw Low, Catch High

- If an exception occurs inside a function and there is no except for it there, Python checks to see if whoever called the function is willing to handle the error.
- It keeps working its way up through the call stack until it finds a matching except.
- If there isn’t one, Python takes care of the exception itself.

Exceptions: Throw Low, Catch High

Write most of your code without exception handlers, but put a few handlers in the higher-level functions of your program to catch and report all errors.

```
def sum_reciprocals(values):
    result = 0
    for v in values:
        result += 1/v
    return result

numbers = [-1, 0, 1]
try:
    one_over = sum_reciprocals(numbers)
except ArithmeticError as error:
    print(f'Error trying to sum reciprocals: {error}')
```

Exceptions: Throw Low, Catch High

Simplified Code Maintenance: Inner functions and methods remain **cleaner** and **more focused** on their **specific tasks**, making the code easier to maintain and understand.

Enhanced Debugging: Catching exceptions at a higher level allows for more comprehensive error reporting and logging. It **provides a better context for the error**, making it easier to trace the root cause.

Scalability: As applications grow, this approach scales better. It's more **efficient** to have **centralized error handling** rather than scattered exception handling throughout the code.

Writing Useful Error Messages

- **Tell the user what they did, not what the program did.**
  - Putting it another way, the message shouldn't state the effect of the error, it should state the cause.
- **Be spatially correct.**
  - i.e., point at the actual location of the error. Few things are as frustrating as being pointed at line 28 when the problem is really on line 35.
- **Be as specific as possible without being or seeming wrong from a user's point of view.**
  - For example, "file not found" is very different from "don't have permissions to open file" or "file is empty."
- **Write for your audience's level of understanding.**
  - For example, error messages should never use programming terms more advanced than those you would use to describe the code to the user.

Writing Useful Error Messages

- **Do not blame the user, and do not use words like fatal, illegal, etc.**
  - The former can be frustrating—in many cases, “user error” actually isn’t—and the latter can make people worry that the program has damaged their data, their computer, or their reputation.
- **Do not try to make the computer sound like a human being.**
  - In particular, avoid humor: very few jokes are funny on the dozenth re-telling, and most users are going to see error messages at least that often.
- **Use a consistent vocabulary.**
  - This rule can be hard to enforce when error messages are written by several different people, but putting them all in one module makes review easier.

Reporting Errors: Logging

Logging frameworks:

- let us leave debugging statements in our code and turn them on or off at will
- can send an output to any of several destinations

```
if LOG_LEVEL >= 0:
    print('Processing files...')
for fname in args.infiles:
    if LOG_LEVEL >= 1:
        print(f'Reading in {fname}...')
    if fname[-4:] != '.csv':
        msg = ERRORS['not_csv_suffix'].format(fname=fname)
        raise OSError(msg)
    with open(fname, 'r') as reader:
        if LOG_LEVEL >= 1:
            print(f'Computing word counts...')
        update_counts(reader, word_counts)
```

Using the Python logging library

```
import logging


logging.info('Processing files...')
for fname in args.infiles:
    logging.debug(f'Reading in {fname}...')
    if fname[-4:] != '.csv':
        msg = ERRORS['not_csv_suffix'].format(fname=fname)
        raise OSError(msg)
    with open(fname, 'r') as reader:
        logging.debug('Computing word counts...')
        update_counts(reader, word_counts)
```

Logging Levels:

- `DEBUG`: very detailed information used for localizing errors.
- `INFO`: confirmation that things are working as expected.
- `WARNING`: something unexpected happened, but the program will keep going.
- `ERROR`: something has gone badly wrong, but the program hasn't hurt anything.
- `CRITICAL`: potential loss of data, security breach, etc.

Logging levels: Customizing logging messages

```
import logging


logging.basicConfig(level=logging.DEBUG, filename='logging.log')

logging.debug('This is for debugging.')
logging.info('This is just for information.')
logging.warning('This is a warning.')
logging.error('Something went wrong.')
logging.critical('Something went seriously wrong.')
```

```
DEBUG:root:This is for debugging.
INFO:root:This is just for information.
WARNING:root:This is a warning.
ERROR:root:Something went wrong.
CRITICAL:root:Something went seriously wrong.
```