Exceptions and a bit on saving data

This segment is going to be a bit messier, because it's mostly about pickling, but also is an excuse to talk a bit about exceptions.

I recently got a question about how to pickle. Pickling is a way to save data in Python, in Python's own special format. Pickling is both powerful and dangerous. Powerful because you can put in a variable you can save to disk. You can save functions, astropy HDUlists, dictionaries, etc. It's dangerous because pickles contain Python bytecode, which you can't read and will be run when the object is unpickled. It's possible to have the unpickling process do something malicious. That said, the very sort tutorial on how to pickle:

```
import pickle

with open('filename.p', 'wb') as output_file:
    pickle.dump(some_variable, output_file)

with open('filename.p', 'rb') as input_file:
    some_variable = pickle.load(input_file)
```

That's really all there is to pickling. It's wonderful to use because it's so easy. In fact, it's possible to save and load pickles in a single line. So why don't we?

Users of other languages may be used to writing multiple lines of code to open a file. This is because interacting with a disk can be problematic, and when a program encounters a serious problem it produces some sort of exception. Exceptions are another form of flow control, but they're somewhat unique in that their default behavior is to terminate a program. To prevent this, we need to handle or catch the exception. It may sound like exceptions are awful things and should always be caught and handled if possible, but that's only partially true. Exceptions are wonderful, and should only be caught if we know how to handle the problem that they indicate. If you're not sure what should be done, let the exception go.

Whenever we read or write from the disk, Python may produce an `IOError` (Input/Output Error), which is a type of exception. So to explicitly catch what could happen when we try to pickle:

```
try:
    output_file = open('filename.p', 'wb')
    pickle.dump(some_variable, output_file)
except IOError:
    print('Oh no!')
    # Do some other stuff like trying again or notifying the user
```

This is the simplest form of a try/except block. The code in the `try` block is executed. If an exception is encountered, execution immediately jumps down to the `except` block, where it checks if the type of exception encountered is listed. If the exception encountered matches a type listed, the code in the `except` block is run. We can have multiple except blocks (which are checked for a matchiing exception in order), which can all list multiple kinds of exceptions, using the syntax `except (IOError, RuntimeError) as err:`.

There are two other clauses that can follow `try... except`, an `else` and a `finally`. The `else` must come after the last `except`, and is run only if no exceptions are encountered in the `try`.

The `finally` clause is what's often important for file I/O. The `finally` block comes at the very end and the code within is executed when execution leaves `try`. `finally` ensures that certain cleanup happens after code execution, regardless of if an exeption appears or if it can be properly handled. If the code in `try` sucks up a bunch of system resources and those resources need to be freed, that is handled in `finally`.

A proper use of pickling without a `with` looks like:

```
try:
    output_file = open('filename.p', 'wb')
    pickle.dump(some_variable, output_file)
finally:
    output_file.close()
```

This is why the `with` was introduced. It handles setup and cleanup neatly by responding to what follows the `with`.

So why are exceptions nice? They let you respond intelligently to errors. If I'm trying to download a file to my hard drive from an online data service and I get a `ConnectionError`, I want to try to download it again. If I get an `IOError` something has gone wrong with my hard drive, in which case I want the program to stop running anyway, and I want the information that comes with the traceback that not handling the exception will produce.

There are also neat hacks like this, where I can convert everything that's number-like to floating point:

```
some_list = ['1', '1.5', '+1.67', 'foo']
numbers = []
for thing in some_list:
    try:
        numbers.append(float(thing))
    except ValueError:
        numbers.append(thing)
```