A quick tour of the os module

This isn't a comprehensive guide to the use of `os`, for that you'll want to visit `docs.python.org`, or just google "python os". The objective here is not to provide detailed documentation, but to introduce the reader to some commonly-used features and provide some intuition for what they do. `os` is a standard-library module which means you'll have to `import os` (only builtins don't need to be imported), but it will be availble on every installation of Python.

The primary function of the os module is to get information from or interact with an operating system.

Basic use cases

`os` can get basic information about the current system with `os.uname()`, which can let a program react intelligently to what computer it is running on.

There are also sufficient functions to replace all the terminal commands. They're not nearly as terse, but `os.chdir()` can replace `cd`, `os.getcwd()` can replace `pwd`, `os.mkdir()` and `os.mkdirs()` can replace `mkdir`. For file copying refer to the module `shutil`, but there is `os.remove()` and `os.rename()`

Absolute vs relative paths

Aside from making directories, mostly `os` is used to navigate directories and find files of interest. Say we have some directory, and within it are a bunch of files. I know there are some fits files I want to open and process, but there could be other things in the directory, and I'm not certain where the script is going to be but I'm certain of where the files will be. For this, I need to know the absolute path to the files. An absolute path starts with `/` on linux or OSX and a drive letter in Windows, such as `C:\`. There are also relative paths, which point to a location based on the location of the script running. So if I had a script running in `/home/ben/examples` and I want to get to `/home/ben/examples/data/test.fits`, I could use `fits.open('data/test.fits')` or `fits.open('/home/ben/examples/data/test.fits')`. The latter will always point to the same location, the former depends on where the script contains that code is.

Also note that since we have the function `os.chdir`, code will be much more robust if you always use absolute paths since you're free to change directory at any time and the location that an absolute path points to will not change. Most likely changing directory with code that relies on relative paths would just crash but it could produce mysterious behavior.

Iterating through files- Python 3

So we have some absolute path to a directory, and we need absolute paths to all the things in the directory that end in `.fits`. For this, Python 3 has a clearly better solution. PEP 20 says "There should be one-- and preferably only one --obvious way to do it." In Python 3, the solution is `os.scandir`, which does not exist in Python 2.

```
for entry in os.scandir('/home/ben/examples'):
    if entry.is_file() and entry.name.endswith('.fits'):
        image = fits.open(entry.path)[0].data
        # Process away
```

`os.scandir` returns an iterator that produces `DirEntry` objects. `DirEntry` objects have attributes `path` and `name` which are both strings. `path` is the absolute path to the directory, file, or symlink (shortcut) while `name` is just the last item on the path; the file or directory name. The `DirEntry` objects also know if they are files or directories. While it's unlikely to have a directory that ends with `.fits`, where it's easy to idiot-proof code you should.

One of the big changes from Python 2 to Python 3 is that many functions return generators instead of lists. `os.scandir` is one such example. Trying to `print(os.scandir('some/path'))` will get you something totally unhelpful. The reason to use generators over lists is mostly for memory usage. A generator stores very little information in memory, regardless of how many things it can produce whereas returning a list requires much more memory usage. In all common use cases, generators are at least as fast and often faster than the list-based functions they replace. In

particular, the code above using `os.scandir` is much faster than the solutions available in Python 2, though the speed difference will be insiginficant compared to opening a single fits file.

Iterating through files- Python 2

In Python 2 there are two common ways to get items in a directory (these both work in Python 3 but are not preferred). They are `os.listdir` and `os.walk`.

`os.listdir`

This is the simpler solution; `os.listdir` takes a path to a directory and returns a list of strings, where each string is the name of an entry in the directory. Unlike `os.scandir`, there isn't an easy way to get absolute paths or even relative paths from the location of the script running to the entries returned by `os.listdir`. Some examples:

```
target_dir = '/home/ben/examples'
for entry in os.listdir(target_dir):
    path_to_file = os.path.join(target_dir, entry)
    if os.path.isfile(path_to_file) and entry.endswith('.fits'):
        image = fits.open(path_to_file)[0].data
        # Process away
```

Compared to using `os.scandir`, it looks like we've added much more complexity to what should be a very simple operation. Note that if we wanted to actually get absolute paths, we need another bulky function call to make sure we get absolute paths.

```
path_to_file = os.path.abspath(os.path.join(target_dir, entry))
```

`os.walk`

In most cases, `os.listdir` is the better solution; just because it's simpler. `os.walk` is much more powerful and not really designed for this use case, but it's a viable option. The normal use is:

```
for root, dirs, files in os.walk('/home/ben'):
    for entry in files:
        if entry.endswith('.fits'):
            path_to_file = os.path.join(root, entry)
            image = fits.open(path_to_file)[0].data
            # Process away
```

The power of `os.walk` is that it gives you a list of all the directories and files in some target directory, then does the same for every directory in the sub-directories, then the same for... ect. It's a recursive algorithm that can be very useful. If we only want to go one layer deep, we can do

```
target_dir = '/home/ben'
for entry in next(os.walk(target_dir))[2]:
    if entry.endswith('.fits'):
        path_to_file = os.path.join(target_dir, entry)
        image = fits.open(path_to_file)[0].data
        # Process away
```

This has one advantage that we don't have to check if the entry is a file, because `os.walk` already separates the directories and files, but the logic is a bit obfuscated with the `next` call.