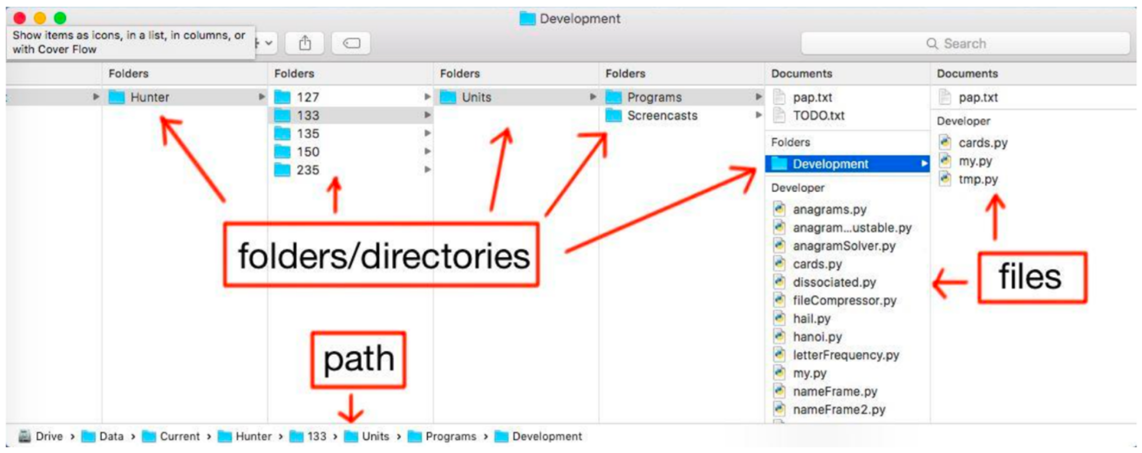
Unit 5

Learn something new (designed for UNIX and Mac users)

Note: There is no major change in content between the official 'Learn Something new' document and this document other than the examples also working for those with Mac computers and UNIX-based computers.

1. Files on a computer are normally organized into folders. For example, here is a snapshot of some of the files and folders on a computer running a Mac OS X operating system.



The file `tmp.py` is in a folder called **Development**, which is itself located in a folder called **Programs**. The main body of the snapshot shows the relationships between folders. For example, both the **Programs** and **Screencasts** folders are in a folder called **Units**.

At the bottom of this snapshot is a long sequence called the **path**, a complete specification of the location of a file on the computer. The path for the four `.py` files listed on the right-hand side of the snapshot is represented on your computer as a long string of characters:

```
/Data/Current/Hunter/133/Units/Programs/Development
```

That is, the files are in the **Development** folder which is in the **Programs** folder, which is in the **Units** folder and so on, up through the **Data** folder which is located on the hard drive, represented here as the very first forward slash character at the beginning of the path, `/`.

We can write Python programs that interact with the operating system using the `os` module. In the examples that follow, the code we run will always be contained in the `tmp.py` program file shown in the snapshot above. Naturally, you will get different results when you run these examples, since your folders will be organized differently.

The function `os.listdir` takes a string specifying a path and returns a list of all items found at that location. The function takes its name from the fact that the technical term for a folder is a **directory**. Using a path like the long one given above involves complications we want to postpone. To make things easier, we'll start by using a special path—the string `'.'` consisting of a single dot or period character. This path is an abbreviation for the **current directory**, in our case, the one where our program file `tmp.py` is located. Here's a program:

```
import os

path = '.'
for filename in os.listdir(path):
    print(filename)
```

And here's the output—the four files in the current directory.

```
cards.py
my.py
pap.txt
tmp.py
```

Another special path is `'..'`, which refers to the **parent directory** of the current directory—the folder in which it's located. In our case, the current directory, or folder, is **Development**. So the parent directory is **Programs**. The following program lists all the files in the **Programs** folder.

```
import os

path = '..'
for filename in os.listdir(path):
    print(filename)
```

Here's the output:

```
anagrams.py
anagramsAdjustable.py
anagramSolver.py
cards.py
Development
dissociated.py
fileCompressor.py
hail.py
hanoi.py
letterFrequency.py
my.py
my.pyc
nameFrame.py
nameFrame2.py
pap.txt
Program1.py
Program2.py
Program3.py
Program4.py
RecursiveCountdown.py
spiral.py
squareSpiral.py
sudoku.py
summarize.py
TODO.txt
Urlopen.py
```

As you can see from the `.py` endings, most of the items in the Programs folder are indeed program files. Note carefully, however, that the fifth item listed is **Development**, which is not a file, but a directory. Naturally, if we look at the parent of **Development**, one of the items contained must be **Development** itself.

2. The output just produced does not make it clear that **Development** is a directory. Suppose we'd like to flag directories in the listing with stars like this:

```
anagrams.py
anagramsAdjustable.py
anagramSolver.py
cards.py
*** Development
dissociated.py
fileCompressor.py
hail.py
.
.
.
```

The function `os.path.isdir` takes a string and returns `True` if the string specifies a path, that is if it is a directory. We'll also need a function called `os.path.join` that takes any number of pathname elements and joins them together according to the rules of the operating system. For example,

```
os.path.join('.', 'tmp.py')
```

returns

```
./tmp.py
```

Note that the strings `'.'` and `'tmp.py'` have been joined using a forward-slash (`/`) character. This is correct on a computer running Mac OS X. What happens on a different computer will depend on the operating system it is running.

Using these two new functions, we can write a listing program that flags directories, giving the output above.

```
import os

path = '..'
for filename in os.listdir(path):
    newpath = os.path.join(path, filename)
    if os.path.isdir(newpath):
        print('***', filename)
    else:
        print(filename)
```

The only tricky point here is the condition for the `if` statement. Keep in mind that `os.path.isdir` must be given a path, not the name of a file. As we get to each file name in the `for` statement, we want to check if adding that name to the existing path produces a new path which is a directory. For example, when filename is **Development**, we want to check

/Data/Current/Hunter/133/Units/Programs/Development

Since `'..'` is an abbreviation for the parent directory

/Data/Current/Hunter/133/Units/Programs/

what we want to check is `'../Development '`. This is exactly what `newpath` is assigned to represent.

3. For a reason that will be clear in a moment, it will be handy to encapsulate the file listing code we've just written in a function. Here's a revised version of the program.

```
import os

def lister(path):
    for filename in os.listdir(path):
        newpath = os.path.join(path, filename)
        if os.path.isdir(newpath):
            print('***', filename)
        else:
            print(filename)

lister('.')
print('---')
lister('..')
```

As you can see, one immediate benefit we get is that the function can be called twice just as easily as once. The three lines at the bottom produce the following output, which gives the listing for both the current directory *and* the parent directory.

```
cards.py
my.py
pap.txt
tmp.py
---
anagrams.py
anagramsAdjustable.py
anagramSolver.py
cards.py
*** Development
dissociated.py
fileCompressor.py
hail.py
.
.
.
```

4. Now here's a big payoff. When we get to a name like **Development** which is a directory, it's nice to flag it with stars in the listing, but it would be even better to show the files (and any directories) it contains. Consider the following code and note carefully that only one line in the function definition is new.

```
import os

def lister(path):
    for filename in os.listdir(path):
        newpath = os.path.join(path, filename)
        if os.path.isdir(newpath):
            print('***', filename)
            lister(newpath)
        else:
            print(filename)

lister('..')
```

Here, when we encounter a directory, we display it flagged with stars as before, but we also call the `lister` function with the new path just constructed. For example, we display `*** Development` and then call `lister` with `'../Development'`. Naturally this produces a listing of the four files in that location.

Here's the new output. Note that one call—**`lister('..')`**—produces a complete, organized listing of two directories. We say the listing is organized, because files within a subdirectory like **Development** are listed just under the directory name—we'll improve how the organization is reflected in the listing in just a moment.

```
anagrams.py
anagramsAdjustable.py
anagramSolver.py
cards.py
*** Development
cards.py
my.py
pap.txt
tmp.py
dissociated.py
fileCompressor.py
hail.py
```

5. The full power of the **`lister`** function takes a little while to sink in. Take a moment to consider what happens if we call it this way: **`lister('/')`**. The function would list out all items contained at the top level of the directory structure. Some of these would themselves be folders, so `lister` would automatically be called again to produce a listing of the items they contain. But some of *these* would also be directories, meaning that `lister` would be called yet again one level deeper.

   Continuing in this way, the single function call, `lister('/')`, will produce a listing of *every file in every folder on the entire hard drive!*

   Before we produce anything like this amount of output, let's improve the way it's displayed. If you look back at the previous result, you'll see that there's no easy way to tell that only the first four files listed after **`*** Development`** are contained within that folder. We'd much rather display the contained file names indented, like this:

```
anagrams.py
anagramsAdjustable.py
anagramSolver.py
cards.py
*** Development
    cards.py
    my.py
    pap.txt
    tmp.py
dissociated.py
fileCompressor.py
hail.py
.
.
.
```

   Let's pass the `lister` function a string of spaces to print before each file name. With each new call one level deeper, we'll make this string a little longer.

Here's the revised code.

```
import os

def lister(path, indent):
    for filename in os.listdir(path):
        newpath = os.path.join(path, filename)
        if os.path.isdir(newpath):
            print(indent, '***', filename)
            lister(newpath, indent+'    ')
        else:
            print(indent, filename)

parentOfParent = os.path.join('..', '..')
lister(parentOfParent, '')
```

The last line here is where the main work begins, with a call to `lister`. This initial call passes the function a path and a string to use for indentation. The path is `'../..'`, which means the parent of the parent of the current directory. In our case, the current directory is **Development**, the parent is **Programs,** and the parent of the parent is **Units**—check the figure at the very beginning of this unit to see how these are related.

Our initial call thus passes the function an abbreviation for the path to **Units** and an empty string consisting of zero spaces. This results in an un-indented listing of the items in **Units** and, along the way, calls to **lister** for each of the directories contained in **Units**— that is, for **Programs** and **Screencasts**. In these calls, the indentation string passed to the function contains four spaces, meaning these subdirectory listings will be indented. The listing for **Programs** causes yet another call to lister when **Development** is found within the folder.

Here's the output (with dots substituted for some very long parts of the listing to save space):

```
dir.png
IDLE.png
*** Programs
    anagrams.py
    anagramsAdjustable.py
    anagramSolver.py
    cards.py
    *** Development
        cards.py
        my.py
        pap.txt
        tmp.py
    dissociated.py
    fileCompressor.py
    hail.py
    .
    .
    .
    summarize.py
    TODO.txt
    urlopen.py
    wptrans.py
*** Screencasts
    1.1.swf
    1.2.swf
    1.3.swf
    .
    .
    .
    4.3.swf
    4.4.swf
    4.5.swf
    4.6.swf
Thumbs.db
unit01.doc
unit02.doc
Unit03.doc
Unit04.doc
unit05.BAK
unit05.doc
~$unit05.doc
~WRL0001.tmp
```

6. Rather than displaying all file names, we might like to pick out a selection. For example, suppose we'd like to see just Python program files. These are easy to identify because their file names end in `.py`. If `filename` refers to a string like `'cards.py'`, we can pick out individual characters using indexing, just as we did with lists in Unit 4. All we need to remember is that position numbers begin with zero. So, in this example, `filename[0]` is `'c'`, `filename[1]` is `'a'` and so on, through `filename[7]`, which is `'y'`.

   The index for the very last character in a string is always one less than the length of the string. In our example, the string `'cards.py'` has 8 characters, but since we count positions starting with zero, the final `y` is in position 7—that is, 8–1. That means we can refer to the last character in filename this way:

   ```
   filename[len(filename)-1]
   ```

   We calculate the length of the string (8), subtract 1 (giving 7) and use this as the index. Luckily, since this is a common task, Python also provides a much easier method. *Negative* indices count backward from the right-hand end of the string. So, the easy way to refer to the last character is `filename[-1]`. The next-to-last character is `filename[-2]` and so on. Just carefully note that in counting backward we start from one, not zero*.

7. At this point, we could check if a file name refers to a Python program file by checking the last few characters one by one. But Python also provides a very handy mechanism called **slicing** for specifying a group of adjacent indices all at once. If `filename` refers to the string `'cards.py'`, then `filename[0:5]` has the value `'cards'`. And `filename[5:8]` has the value `'.py'`. To specify a slice, we give two indices—a starting position and an ending position—separated by a colon (`:`).

   The only tricky point is that the slice corresponds to all indices from the first up to *one less* than the last. This takes some getting used to, but it is convenient for several reasons. First, if we subtract the two indices, we get the number of characters in the slice. In our examples, there are 5-0 or 5 characters in `filename[0:5]` and 8–5 or 3 in `filename[5:8]`. Second, two slices that divide a string share an index—5 in our example. Finally, if we want a slice to go right up to—and including—the last character of a string, the second index is simply the length of the string, since the index of the last character is one less than the length.

   Starting a slice at the beginning of a string or continuing it to the end of a string is so common that Python provides shortcuts—just leave out either the first or the last index. The two slices we've been using as examples can be written as just `filename[:5]` and `filename[5:]`.

* This isn't as arbitrary as it may seem.
Normal indices tell you how far to go forward from the beginning of the string, so 0 naturally means to go no distance at all—the first character is located at the very beginning.
Negative indices are combined with the length of the string just as in our example: length minus one is the index of the last character, length minus two is the index of the next-to-last character and so on.

Finally, note that we've been illustrating slices using strings, but everything we've said applies equally to lists. If you want a section of a list, just use a slice that specifies the desired range of indices.

Now here's a quick application of slices. We'll add a line to our previous program so that it displays only directory names and Python program files.

```
import os

def lister(path, indent):
    for filename in os.listdir(path):
        newpath = os.path.join(path, filename)
        if os.path.isdir(newpath):
            print(indent, '***', filename)
            lister(newpath, indent+'  ')
        else:
            if filename[:3] == '.py':
                print(indent, filename)

parentOfParent = os.path.join('..', '..')
lister(parentOfParent, '')
```

Here's the output:

```
*** Programs
    anagrams.py
    anagramsAdjustable.py
    anagramSolver.py
    cards.py
    *** Development
        cards.py
        my.py
        pap.txt
        tmp.py
    dissociated.py
    fileCompressor.py
    hail.py
    hanoi.py
    letterFrequency.py
    my.py
    nameFrame.py
    nameFrame2.py
    Program1.py
    Program2.py
    Program3.py
    Program4.py
    recursiveCountdown.py
    spiral.py
    squareSpiral.py
    sudoku.py
    summarize.py
    urlopen.py
    wptrans.py
*** Screencasts
```

Now it's immediately clear that neither the top-level folder—**Units**—nor the **Screencasts** subfolder contain any program files.