Chapter 3. Comprehensions

> *Our imagination is stretched to the utmost, not, as in fiction, to imagine things which are not really there, but just to comprehend those things which are.*
>
> — *Richard Feynman*

Diving In

Every programming language has that one feature, a complicated thing intentionally made simple. If you're coming from another language, you could easily miss it, because your old language didn't make that thing simple (because it was busy making something else simple instead). This chapter will teach you about list comprehensions, dictionary comprehensions, and set comprehensions: three related concepts centered around one very powerful technique. But first, I want to take a little detour into two modules that will help you navigate your local file system.

3.2. Working With Files And Directories

Python 3 comes with a module called `os`, which stands for "operating system." The `os` module contains a plethora of functions to get information on — and in some cases, to manipulate — local directories, files, processes, and environment variables. Python does its best to offer a unified API across all supported operating systems so your programs can run on any computer with as little platform-specific code as possible.

3.2.1. The Current Working Directory

When you're just getting started with Python, you're going to spend a lot of time in the Python Shell. Throughout this book, you will see examples that go like this:

1. Import one of the modules in the `examples` folder

2. Call a function in that module
3. Explain the result

If you don't know about the current working directory, step 1 will probably fail with an `ImportError`. Why? Because Python will look for the example module in the import search path, but it won't find it because the `examples` folder isn't one of the directories in the search path. To get past this, you can do one of two things:

1. Add the `examples` folder to the import search path
2. Change the current working directory to the `examples` folder

There is always a current working directory.

The current working directory is an invisible property that Python holds in memory at all times. There is always a current working directory, whether you're in the Python Shell, running your own Python script from the command line, or running a Python `CGI` script on a web server somewhere.

The `os` module contains two functions to deal with the current working directory.

```
>>> import os                                          ①
>>> print(os.getcwd())                                 ②
C:\Python31
>>> os.chdir('/Users/pilgrim/diveintopython3/examples')  ③
>>> print(os.getcwd())                                 ④
C:\Users\pilgrim\diveintopython3\examples
```

① The `os` module comes with Python; you can import it anytime, anywhere.

② Use the `os.getcwd()` function to get the current working directory. When you run the graphical Python Shell, the current working directory starts as the directory where the Python Shell executable is. On Windows, this depends on where you installed Python; the default directory is `c:\Python31`. If you run the

Python Shell from the command line, the current working directory starts as the directory you were in when you ran `python3`.

③ Use the `os.chdir()` function to change the current working directory.

④ When I called the `os.chdir()` function, I used a Linux-style pathname (forward slashes, no drive letter) even though I'm on Windows. This is one of the places where Python tries to paper over the differences between operating systems.

3.2.2. Working With Filenames and Directory Names

While we're on the subject of directories, I want to point out the `os.path` module. `os.path` contains functions for manipulating filenames and directory names.

```
>>> import os
>>> print(os.path.join('/Users/pilgrim/diveintopython3/examples/', 'humansize.py'))                ①
/Users/pilgrim/diveintopython3/examples/humansize.py
>>> print(os.path.join('/Users/pilgrim/diveintopython3/examples', 'humansize.py'))                 ②
/Users/pilgrim/diveintopython3/examples\humansize.py
>>> print(os.path.expanduser('~'))                                                                 ③
c:\Users\pilgrim
>>> print(os.path.join(os.path.expanduser('~'), 'diveintopython3', 'examples', 'humansize.py'))    ④
c:\Users\pilgrim\diveintopython3\examples\humansize.py
```

① The `os.path.join()` function constructs a pathname out of one or more partial pathnames. In this case, it simply concatenates strings.

② In this slightly less trivial case, calling the `os.path.join()` function will add an extra slash to the pathname before joining it to the filename. It's a backslash instead of a forward slash, because I constructed this example on Windows. If you replicate this example on Linux or Mac OS X, you'll see a forward slash instead. Don't fuss with slashes; always use `os.path.join()` and let Python do the right thing.

③ The `os.path.expanduser()` function will expand a pathname that uses ~ to represent the current user's home directory. This works on any platform where users have a home directory, including Linux, Mac OS X, and Windows. The returned path does not have a trailing slash, but the `os.path.join()` function doesn't mind.

④ Combining these techniques, you can easily construct pathnames for directories and files in the user's home directory. The `os.path.join()` function can take any number of arguments. I was overjoyed when I discovered this, since `addSlashIfNecessary()` is one of the stupid little functions I always need to write

when building up my toolbox in a new language. *Do not* write this stupid little function in Python; smart people have already taken care of it for you.

`os.path` also contains functions to split full pathnames, directory names, and filenames into their constituent parts.

```
>>> pathname = '/Users/pilgrim/diveintopython3/examples/humansize.py'
>>> os.path.split(pathname)                                          ①
('/Users/pilgrim/diveintopython3/examples', 'humansize.py')
>>> (dirname, filename) = os.path.split(pathname)                    ②
>>> dirname                                                          ③
'/Users/pilgrim/diveintopython3/examples'
>>> filename                                                         ④
'humansize.py'
>>> (shortname, extension) = os.path.splitext(filename)              ⑤
>>> shortname
'humansize'
>>> extension
'.py'
```

① The `split` function splits a full pathname and returns a tuple containing the path and filename.

② Remember when I said you could use multi-variable assignment to return multiple values from a function? The `os.path.split()` function does exactly that. You assign the return value of the `split` function into a tuple of two variables. Each variable receives the value of the corresponding element of the returned tuple.

③ The first variable, `dirname`, receives the value of the first element of the tuple returned from the `os.path.split()` function, the file path.

④ The second variable, `filename`, receives the value of the second element of the tuple returned from the `os.path.split()` function, the filename.

⑤ `os.path` also contains the `os.path.splitext()` function, which splits a filename and returns a tuple containing the filename and the file extension. You use the same technique to assign each of them to separate variables.

3.2.3. Listing Directories

The `glob` module is another tool in the Python standard library. It's an easy way to get the contents of a directory programmatically, and it uses the sort of wildcards that you may already be familiar with from working on the command line.

```
>>> os.chdir('/Users/pilgrim/diveintopython3/')
>>> import glob
>>> glob.glob('examples/*.xml')                    ①
['examples\\feed-broken.xml',
 'examples\\feed-ns0.xml',
 'examples\\feed.xml']
>>> os.chdir('examples/')                          ②
>>> glob.glob('*test*.py')                         ③
['alphameticstest.py',
 'pluraltest1.py',
 'pluraltest2.py',
 'pluraltest3.py',
 'pluraltest4.py',
 'pluraltest5.py',
 'pluraltest6.py',
 'romantest1.py',
 'romantest10.py',
 'romantest2.py',
 'romantest3.py',
 'romantest4.py',
 'romantest5.py',
 'romantest6.py',
 'romantest7.py',
 'romantest8.py',
 'romantest9.py']
```

The `glob` module uses shell-like wildcards.

① The `glob` module takes a wildcard and returns the path of all files and directories matching the wildcard. In this example, the wildcard is a directory path plus "`*.xml`", which will match all `.xml` files in the `examples` subdirectory.

② Now change the current working directory to the `examples` subdirectory. The `os.chdir()` function can take relative pathnames.

③ You can include multiple wildcards in your glob pattern. This example finds all the files in the current working directory that end in a `.py` extension and contain the word `test` anywhere in their filename.

3.2.4. Getting File Metadata

Every modern file system stores metadata about each file: creation date, last-modified date, file size, and so on. Python provides a single API to access this metadata. You don't need to open the file; all you need is the filename.

```
>>> import os
>>> print(os.getcwd())                  ①
c:\Users\pilgrim\diveintopython3\examples
>>> metadata = os.stat('feed.xml')      ②
>>> metadata.st_mtime                   ③
1247520344.9537716
>>> import time                         ④
>>> time.localtime(metadata.st_mtime)   ⑤
time.struct_time(tm_year=2009, tm_mon=7, tm_mday=13, tm_hour=17,
  tm_min=25, tm_sec=44, tm_wday=0, tm_yday=194, tm_isdst=1)
```

① The current working directory is the `examples` folder.

② `feed.xml` is a file in the `examples` folder. Calling the `os.stat()` function returns an object that contains several different types of metadata about the file.

③ `st_mtime` is the modification time, but it's in a format that isn't terribly useful. (Technically, it's the number of seconds since the Epoch, which is defined as the first second of January 1st, 1970. Seriously.)

④ The `time` module is part of the Python standard library. It contains functions to convert between different time representations, format time values into strings, and fiddle with timezones.

⑤ The `time.localtime()` function converts a time value from seconds-since-the-Epoch (from the `st_mtime` property returned from the `os.stat()` function) into a more useful structure of year, month, day, hour, minute, second, and so on. This file was last modified on July 13, 2009, at around 5:25 PM.

```
# continued from the previous example
>>> metadata.st_size                    ①
3070
>>> import humansize
```

```
>>> humansize.approximate_size(metadata.st_size)   ②
'3.0 KiB'
```

① The `os.stat()` function also returns the size of a file, in the `st_size` property. The file `feed.xml` is `3070` bytes.

② You can pass the `st_size` property to the `approximate_size()` function.

3.2.5. Constructing Absolute Pathnames

In the previous section, the `glob.glob()` function returned a list of relative pathnames. The first example had pathnames like `'examples\feed.xml'`, and the second example had even shorter relative pathnames like `'romantest1.py'`. As long as you stay in the same current working directory, these relative pathnames will work for opening files or getting file metadata. But if you want to construct an absolute pathname — *i.e.* one that includes all the directory names back to the root directory or drive letter — then you'll need the `os.path.realpath()` function.

```
>>> import os
>>> print(os.getcwd())
c:\Users\pilgrim\diveintopython3\examples
>>> print(os.path.realpath('feed.xml'))
c:\Users\pilgrim\diveintopython3\examples\feed.xml
```

*
**

3.3. List Comprehensions

A list comprehension provides a compact way of mapping a list into another list by applying a function to each of the elements of the list.

```
>>> a_list = [1, 9, 8, 4]
>>> [elem * 2 for elem in a_list]          ①
[2, 18, 16, 8]
>>> a_list                                 ②
```

```
[1, 9, 8, 4]
>>> a_list = [elem * 2 for elem in a_list]  ③
>>> a_list
[2, 18, 16, 8]
```

You can use any Python expression in a list comprehension.

① To make sense of this, look at it from right to left. `a_list` is the list you're mapping. The Python interpreter loops through `a_list` one element at a time, temporarily assigning the value of each element to the variable `elem`. Python then applies the function `elem * 2` and appends that result to the returned list.

② A list comprehension creates a new list; it does not change the original list.

③ It is safe to assign the result of a list comprehension to the variable that you're mapping. Python constructs the new list in memory, and when the list comprehension is complete, it assigns the result to the original variable.

You can use any Python expression in a list comprehension, including the functions in the `os` module for manipulating files and directories.

```
>>> import os, glob
>>> glob.glob('*.xml')                                    ①
['feed-broken.xml', 'feed-ns0.xml', 'feed.xml']
>>> [os.path.realpath(f) for f in glob.glob('*.xml')]  ②
['c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-broken.xml',
```

```
 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-ns0.xml',
 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed.xml']
```

① This returns a list of all the `.xml` files in the current working directory.

② This list comprehension takes that list of `.xml` files and transforms it into a list of full pathnames.

List comprehensions can also filter items, producing a result that can be smaller than the original list.

```
>>> import os, glob
>>> [f for f in glob.glob('*.py') if os.stat(f).st_size > 6000]  ①
['pluraltest6.py',
 'romantest10.py',
 'romantest6.py',
 'romantest7.py',
 'romantest8.py',
 'romantest9.py']
```

① To filter a list, you can include an `if` clause at the end of the list comprehension. The expression after the `if` keyword will be evaluated for each item in the list. If the expression evaluates to `True`, the item will be included in the output. This list comprehension looks at the list of all `.py` files in the current directory, and the `if` expression filters that list by testing whether the size of each file is greater than `6000` bytes. There are six such files, so the list comprehension returns a list of six filenames.

All the examples of list comprehensions so far have featured simple expressions — multiply a number by a constant, call a single function, or simply return the original list item (after filtering). But there's no limit to how complex a list comprehension can be.

```
>>> import os, glob
>>> [(os.stat(f).st_size, os.path.realpath(f)) for f in glob.glob('*.xml')]          ①
[(3074, 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-broken.xml'),
 (3386, 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed-ns0.xml'),
 (3070, 'c:\\Users\\pilgrim\\diveintopython3\\examples\\feed.xml')]
>>> import humansize
>>> [(humansize.approximate_size(os.stat(f).st_size), f) for f in glob.glob('*.xml')]  ②
[('3.0 KiB', 'feed-broken.xml'),
```

```
 ('3.3 KiB', 'feed-ns0.xml'),
 ('3.0 KiB', 'feed.xml')]
```

① This list comprehension finds all the `.xml` files in the current working directory, gets the size of each file (by calling the `os.stat()` function), and constructs a tuple of the file size and the absolute path of each file (by calling the `os.path.realpath()` function).

② This comprehension builds on the previous one to call the `approximate_size()` function with the file size of each `.xml` file.

⁂

3.4. Dictionary Comprehensions

A dictionary comprehension is like a list comprehension, but it constructs a dictionary instead of a list.

```
>>> import os, glob
>>> metadata = [(f, os.stat(f)) for f in glob.glob('*test*.py')]     ①
>>> metadata[0]                                                      ②
('alphameticstest.py', nt.stat_result(st_mode=33206, st_ino=0, st_dev=0,
 st_nlink=0, st_uid=0, st_gid=0, st_size=2509, st_atime=1247520344,
 st_mtime=1247520344, st_ctime=1247520344))
>>> metadata_dict = {f:os.stat(f) for f in glob.glob('*test*.py')}   ③
>>> type(metadata_dict)                                              ④
<class 'dict'>
>>> list(metadata_dict.keys())                                       ⑤
['romantest8.py', 'pluraltest1.py', 'pluraltest2.py', 'pluraltest5.py',
 'pluraltest6.py', 'romantest7.py', 'romantest10.py', 'romantest4.py',
 'romantest9.py', 'pluraltest3.py', 'romantest1.py', 'romantest2.py',
 'romantest3.py', 'romantest5.py', 'romantest6.py', 'alphameticstest.py',
 'pluraltest4.py']
>>> metadata_dict['alphameticstest.py'].st_size                      ⑥
2509
```

① This is not a dictionary comprehension; it's a list comprehension. It finds all `.py` files with `test` in their

name, then constructs a tuple of the filename and the file metadata (from calling the `os.stat()` function).

② Each item of the resulting list is a tuple.

③ This is a dictionary comprehension. The syntax is similar to a list comprehension, with two differences. First, it is enclosed in curly braces instead of square brackets. Second, instead of a single expression for each item, it contains two expressions separated by a colon. The expression before the colon (`f` in this example) is the dictionary key; the expression after the colon (`os.stat(f)` in this example) is the value.

④ A dictionary comprehension returns a dictionary.

⑤ The keys of this particular dictionary are simply the filenames returned from the call to `glob.glob('*test*.py')`.

⑥ The value associated with each key is the return value from the `os.stat()` function. That means we can "look up" a file by name in this dictionary to get its file metadata. One of the pieces of metadata is `st_size`, the file size. The file `alphameticstest.py` is `2509` bytes long.

Like list comprehensions, you can include an `if` clause in a dictionary comprehension to filter the input sequence based on an expression which is evaluated with each item.

```
>>> import os, glob, humansize
>>> metadata_dict = {f:os.stat(f) for f in glob.glob('*')}                    ①
>>> humansize_dict = {os.path.splitext(f)[0]:humansize.approximate_size(meta.st_size) \
...                   for f, meta in metadata_dict.items() if meta.st_size > 6000}    ②
>>> list(humansize_dict.keys())                                               ③
['romantest9', 'romantest8', 'romantest7', 'romantest6', 'romantest10', 'pluraltest6']
>>> humansize_dict['romantest9']                                              ④
'6.5 KiB'
```

① This dictionary comprehension constructs a list of all the files in the current working directory (`glob.glob('*')`), gets the file metadata for each file (`os.stat(f)`), and constructs a dictionary whose keys are filenames and whose values are the metadata for each file.

② This dictionary comprehension builds on the previous comprehension, filters out files smaller than `6000` bytes (`if meta.st_size > 6000`), and uses that filtered list to construct a dictionary whose keys are the filename minus the extension (`os.path.splitext(f)[0]`) and whose values are the approximate size of each file (`humansize.approximate_size(meta.st_size)`).

③ As you saw in a previous example, there are six such files, thus there are six items in this dictionary.

④ The value of each key is the string returned from the `approximate_size()` function.

3.4.1. Other Fun Stuff To Do With Dictionary Comprehensions

Here's a trick with dictionary comprehensions that might be useful someday: swapping the keys and values of a dictionary.

```
>>> a_dict = {'a': 1, 'b': 2, 'c': 3}
>>> {value:key for key, value in a_dict.items()}
{1: 'a', 2: 'b', 3: 'c'}
```

Of course, this only works if the values of the dictionary are immutable, like strings or tuples. If you try this with a dictionary that contains lists, it will fail most spectacularly.

```
>>> a_dict = {'a': [1, 2, 3], 'b': 4, 'c': 5}
>>> {value:key for key, value in a_dict.items()}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <dictcomp>
TypeError: unhashable type: 'list'
```

*
**

3.5. Set Comprehensions

Not to be left out, sets have their own comprehension syntax as well. It is remarkably similar to the syntax for dictionary comprehensions. The only difference is that sets just have values instead of key:value pairs.

```
>>> a_set = set(range(10))
>>> a_set
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> {x ** 2 for x in a_set}           ①
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
>>> {x for x in a_set if x % 2 == 0}  ②
{0, 8, 2, 4, 6}
```

```
>>> {2**x for x in range(10)}          ③
{32, 1, 2, 4, 8, 64, 128, 256, 16, 512}
```

① Set comprehensions can take a set as input. This set comprehension calculates the squares of the set of numbers from 0 to `9`.

② Like list comprehensions and dictionary comprehensions, set comprehensions can contain an `if` clause to filter each item before returning it in the result set.

③ Set comprehensions do not need to take a set as input; they can take any sequence.

3.6. Further Reading

- `os` module
- `os` — Portable access to operating system specific features
- `os.path` module
- `os.path` — Platform-independent manipulation of file names
- `glob` module
- `glob` — Filename pattern matching
- `time` module
- `time` — Functions for manipulating clock time
- List comprehensions
- Nested list comprehensions
- Looping techniques