Formatting Strings

WE'LL COVER THE FOLLOWING

- Compound Field Names
- Format Specifiers

Let's take another look at `humansize.py`:

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],          #①
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}

def approximate_size(size, a_kilobyte_is_1024_bytes=True):
    '''Convert a file size to human-readable form.                           #②

    Keyword arguments:
    size -- file size in bytes
    a_kilobyte_is_1024_bytes -- if True (default), use multiples of 1024
                                if False, use multiples of 1000

    Returns: string

    '''                                                                      #③
    if size < 0:
        raise ValueError('number must be non-negative')                      #④

    multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
    for suffix in SUFFIXES[multiple]:
        size /= multiple
        if size < multiple:
            return '{0:.1f} {1}'.format(size, suffix)                        #⑤

    raise ValueError('number too large')
```

① `'KB'`, `'MB'`, `'GB'`... those are each strings.

② Function docstrings are strings. This docstring spans multiple lines, so it uses three-in-a-row quotes to start and end the string.

③ These three-in-a-row quotes end the docstring.

④ There's another string, being passed to the exception as a human-readable error message.

⑤ There's a… whoa, what the heck is that?

Strings can be defined with either single or double quotes.

Python 3 supports formatting values into strings. Although this can include very complicated expressions, the most basic usage is to insert a value into a string with a single placeholder.

```
username = 'mark'
password = 'PapayaWhip'                                    #①
print ("{0}'s password is {1}".format(username, password))  #②
#"mark's password is PapayaWhip"
```

① No, my password is not really `PapayaWhip`.

② There's a lot going on here. First, that's a method call on a string literal. *Strings are objects*, and objects have methods. Second, the whole expression evaluates to a string. Third, `{0}` and `{1}` are *replacement fields*, which are replaced by the arguments passed to the `format()` method.

Compound Field Names

The previous example shows the simplest case, where the replacement fields are simply integers. Integer replacement fields are treated as positional indices into the argument list of the `format()` method. That means that `{0}` is replaced by the first argument (`username` in this case), `{1}` is replaced by the second argument (`password`), &c. You can have as many positional indices as you have arguments, and you can have as many arguments as you want. But replacement fields are much more powerful than that.

```
import humansize
si_suffixes = humansize.SUFFIXES[1000]              #①
print (si_suffixes)
#['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB']

print ('1000{0[0]} = 1{0[1]}'.format(si_suffixes))  #②
#'1000KB = 1MB'
```

① Rather than calling any function in the `humansize` module, you're just grabbing one of the data structures it defines: the list of "SI" (powers-of-1000) suffixes.

② This looks complicated, but it's not. `{0}` would refer to the first argument passed to the `format()` method, `si_suffixes`. But `si_suffixes` is a list. So `{0[0]}` refers to the first item of the list which is the first argument passed to the `format()` method: `'KB'`. Meanwhile, `{0[1]}` refers to the second item of the same list: `'MB'`. Everything outside the curly braces — including `1000`, the equals sign, and the spaces — is untouched. The final result is the string `'1000KB = 1MB'`.

{0} is replaced by the 1st *format()* argument. *{1}* is replaced by the 2nd.

What this example shows is that *format specifiers can access items and properties of data structures using (almost) Python syntax*. This is called *compound field names*. The following compound field names "just work":

1. Passing a list, and accessing an item of the list by index (as in the previous example)
2. Passing a dictionary, and accessing a value of the dictionary by key
3. Passing a module, and accessing its variables and functions by name
4. Passing a class instance, and accessing its properties and methods by name
5. *Any combination of the above*

Just to blow your mind, here's an example that combines all of the above:

```
import humansize
import sys
print ('1MB = 1000{0.modules[humansize].SUFFIXES[1000][0]}'.format(sys))
#1MB = 1000KB
```

Here's how it works:

1. The `sys` module holds information about the currently running Python instance. Since you just imported it, you can pass the `sys` module itself as an argument to the `format()` method. So the replacement field `{0}` refers to the `sys` module.
2. `sys.modules` is a dictionary of all the modules that have been imported in this Python instance. The keys are the module names as strings; the values are the module objects themselves. So the replacement field `{0.modules}` refers to the dictionary of imported modules.
3. `sys.modules['humansize']` is the `humansize` module which you just imported. The replacement field `{0.modules[humansize]}` refers to the `humansize` module. Note the slight difference in syntax here. In real Python code, the keys of the `sys.modules` dictionary are strings; to refer to them, you need to put quotes around the module name (e.g. `'humansize'`). But within a replacement field, you skip the quotes around the dictionary key name (e.g. `humansize`). To quote PEP 3101: Advanced String Formatting, "The rules for parsing an item key are very simple. If it starts with a digit, then it is treated as a number, otherwise it is used as a string."
4. `sys.modules['humansize'].SUFFIXES` is the dictionary defined at the top of the `humansize` module. The replacement field `{0.modules[humansize].SUFFIXES}` refers to that dictionary.
5. `sys.modules['humansize'].SUFFIXES[1000]` is a list of `SI` suffixes: [`'KB'`, `'MB'`, `'GB'`, `'TB'`, `'PB'`, `'EB'`, `'ZB'`, `'YB'`]. So the replacement field `{0.modules[humansize].SUFFIXES[1000]}` refers to that list.
6. `sys.modules['humansize'].SUFFIXES[1000][0]` is the first item of the list of `SI` suffixes: `'KB'`. Therefore, the complete replacement field `{0.modules[humansize].SUFFIXES[1000][0]}` is replaced by the two-character string `KB`.

Format Specifiers

But wait! There's more! Let's take another look at that strange line of code from `humansize.py`:

```
if size < multiple:
```

```
    return '{0:.1f} {1}'.format(size, suffix)
```

`{1}` is replaced with the second argument passed to the `format()` method, which is `suffix`. But what is `{0:.1f}`? It's two things: `{0}`, which you recognize, and `:.1f`, which you don't. The second half (including and after the colon) defines the *format specifier*, which further refines how the replaced variable should be formatted.

> *Format specifiers allow you to munge the replacement text in a variety of useful ways, like the **printf()** function in C. You can add zero- or space-padding, align strings, control decimal precision, and even convert numbers to hexadecimal.*

Within a replacement field, a colon (:) marks the start of the format specifier. The format specifier ".1" means "round to the nearest tenth" (i.e. display only one digit after the decimal point). The format specifier "f" means "fixed-point number" (as opposed to exponential notation or some other decimal representation). Thus, given a size of `698.24` and `suffix` of `'GB'`, the formatted string would be `'698.2 GB'`, because `698.24` gets rounded to one decimal place, then the suffix is appended after the number.

```
print ('{0:.1f} {1}'.format(698.24, 'GB'))
#698.2 GB
```

For all the gory details on format specifiers, consult the Format Specification Mini-Language in the official Python documentation.