Your Short Cut to Knowledge

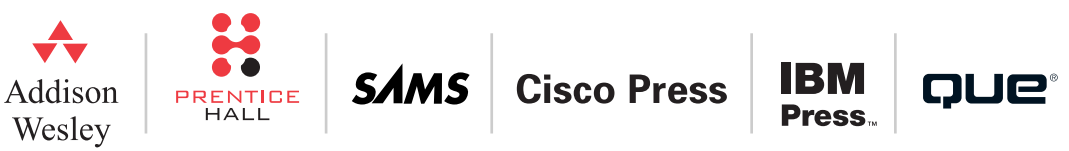
The following is an excerpt from a Short Cut published by one of the Pearson Education imprints.

Short Cuts are short, concise, PDF documents designed specifically for busy technical professionals like you.

We've provided this excerpt to help you review the product before you purchase. Please note, the hyperlinks contained within this excerpt have been deactivated.

Tap into learning—NOW!

Visit **www.informit.com/shortcuts** for a complete list of Short Cuts.



In the right-hand code snippet we create a dictionary whose keys are the valid menu choices, and whose values are function references. In the second statement we retrieve the function reference corresponding to the given action and call the function referred to using the call operator, `()`, and in this example, passing the `db` argument. Not only is the code on the right-hand side much shorter than the code on the left, but also it can scale (have far more dictionary items) without affecting its performance, unlike the left-hand code whose speed depends on how many `elif`s must be tested to find the appropriate function to call.

The `convert-incidents.py` program from the book's examples uses this technique in its `import_()` method, as this extract from the method shows:

```
call = {(".aix", "dom"): self.import_xml_dom,
        (".aix", "etree"): self.import_xml_etree,
        (".aix", "sax"): self.import_xml_sax,
        (".ait", "manual"): self.import_text_manual,
        (".ait", "regex"): self.import_text_regex,
        (".aib", None): self.import_binary,
        (".aip", None): self.import_pickle}
result = call[extension, reader](filename)
```

The complete method is 13 lines long; the extension parameter is computed in the method, and the reader is passed in. The dictionary keys are 2-tuples, and the values are methods. If we had used `if` statements, the code would be 22 lines long, and would not scale as well.

3. Generator Expressions and Functions

A *generator function* or *generator method* is one which contains a `yield` expression. When a generator function is called it returns an iterator. Values are extracted from the iterator one at a time by calling its `__next__()` method. At each call to `__next__()` the generator function's `yield` expression's value (`None` if none is specified) is returned. If the generator function finishes or executes a `return` a `StopIteration` exception is raised.

In practice we rarely call `__next__()` or catch a `StopIteration`. Instead, we just use a generator like any other iterable. Here are two almost equivalent functions. The one on the left returns a list and the one on the right returns a generator.

```
# Build and return a list
def letter_range(a, z):
    result = []
    while ord(a) < ord(z):
        result.append(a)
        a = chr(ord(a) + 1)
    return result
```

```
# Return each value on demand
def letter_range(a, z):
    while ord(a) < ord(z):
        yield a
        a = chr(ord(a) + 1)
```

We can iterate over the result produced by either function using a `for` loop, for example, `for letter in letter_range("m", "v"):`. But if we want a list of the resultant letters, although calling `letter_range("m", "v")` is sufficient for the left-hand function, for the right-hand generator function we must use `list(letter_range("m", "v"))`.

In addition to generator functions and methods it is also possible to create generator expressions. These are syntactically almost identical to list comprehensions, the difference being that they are enclosed in parentheses rather than brackets. Here are their syntaxes:

(*expression* `for` *item* `in` *iterable*)
(*expression* `for` *item* `in` *iterable* `if` *condition*)

Here are two equivalent code snippets that show how a simple `for` … `in` loop containing a `yield` expression can be coded as a generator:

```
def items_in_key_order(d):
    for key in sorted(d):
        yield key, d[key]
```

```
def items_in_key_order(d):
    return ((key, d[key])
            for key in sorted(d))
```

Both functions return a generator that produces a list of key–value items for the given dictionary. If we need all the items in one go we can pass the generator returned by the functions to `list()` or `tuple()`; otherwise, we can iterate over the generator to retrieve items as we need them.

Generators provide a means of performing lazy evaluation, which means that they compute only the values that are actually needed. This can be more efficient than, say, computing a very large list in one go. Some generators produce as many values as we ask for—without any upper limit. For example:

```
def quarters(next_quarter=0.0):
    while True:
        yield next_quarter
        next_quarter += 0.25
```

This function will return 0.0, 0.25, 0.5, and so on, forever. Here is how we could use the generator:

```
result = []
for x in quarters():
    result.append(x)
    if x >= 1.0:
        break
```

The `break` statement is essential—without it the `for ... in` loop will never finish. At the end the result list is `[0.0, 0.25, 0.5, 0.75, 1.0]`.

Every time we call `quarters()` we get back a generator that starts at 0.0 and increments by 0.25; but what if we want to reset the generator's current value? It is possible to pass a value into a generator, as this new version of the generator function shows:

```
def quarters(next_quarter=0.0):
    while True:
        received = (yield next_quarter)
        if received is None:
            next_quarter += 0.25
        else:
            next_quarter = received
```

The `yield` expression returns each value to the caller in turn. In addition, if the caller calls the generator's `send()` method, the value sent is received in the generator function as the result of the `yield` expression. Here is how we can use the new generator function:

```
result = []
generator = quarters()
while len(result) < 5:
    x = next(generator)
    if abs(x - 0.5) < sys.float_info.epsilon:
        x = generator.send(1.0)
    result.append(x)
```

We create a variable to refer to the generator and call the built-in `next()` function which retrieves the next item from the generator it is given. (The same effect can be achieved by calling the generator's `__next__()` special method, in this case, `x = generator.__next__()`.) If the value is equal to 0.5 we send the value 1.0 into the generator (which immediately yields this value back). This time the result list is `[0.0, 0.25, 1.0, 1.25, 1.5]`.

In the next subsection we will review the `magic-numbers.py` program which processes files given on the command line. Unfortunately, the Windows shell program (`cmd.exe`) does not provide wildcard expansion (also called *file globbing*), so if a program is run on Windows with the argument `*.*`, the literal text "*.*" will go into the `sys.argv` list instead of all the files in the current directory. We solve

this problem by creating two different `get_files()` functions, one for Windows and the other for Unix, both of which use generators. Here's the code:

```
if sys.platform.startswith("win"):
    def get_files(names):
        for name in names:
            if os.path.isfile(name):
                yield name
            else:
                for file in glob.iglob(name):
                    if not os.path.isfile(file):
                        continue
                    yield file
else:
    def get_files(names):
        return (file for file in names if os.path.isfile(file))
```

In either case the function is expected to be called with a list of filenames, for example, `sys.argv[1:]`, as its argument.

On Windows the function iterates over all the names listed. For each filename, the function yields the name, but for nonfiles (usually directories), the `glob` module's `glob.iglob()` function is used to return an iterator to the names of the files that the name represents after wildcard expansion. For an ordinary name like `autoexec.bat` an iterator that produces one item (the name) is returned, and for a name that uses wildcards like `*.txt` an iterator that produces all the matching files (in this case those with extension `.txt`) is returned. (There is also a `glob.glob()` function that returns a list rather than an iterator.)

On Unix the shell does wildcard expansion for us, so we just need to return a generator for all the files whose names we have been given.*

Generator functions can be used to create *coroutines*—functions that have multiple entry and exit points (the `yield` expressions) and that can be suspended and resumed at certain points (again at `yield` expressions). Coroutines are often used to provide simpler and lower-overhead alternatives to threading. Several coroutine modules are available from the Python Package Index, `pypi.python.org/pypi`.

*The `glob.glob()` functions are not as powerful as, say, the Unix bash shell, since although they support the `*`, `?`, and `[]` syntaxes, they don't support the `{}` syntax.