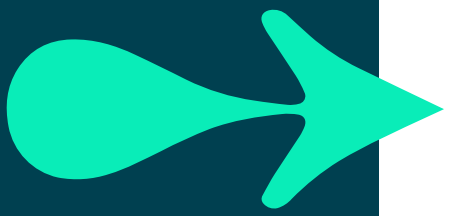
POWERING POTENTIAL

Python 3 Programming

Advanced Collections

ADVANCED COLLECTIONS



Contents

- Advanced functions - filter
- List Comprehensions
- Set and dictionary comprehensions
- Lazy lists
- Generators
- Copying collections

Summary

- Generator objects
- Co-routines and send()
- Generator delegation

Advanced functions - filter

- **`filter`(*function*, *iterable-object*)**
  - Returns an iterator for every item where function returns true
  - The function could be named, or a lambda
  - The iterator can be used in a loop

py3

```
import glob
import os

pattern = 'C:/QA/Python/*'
for fname in (filter(os.path.isdir, glob.iglob(pattern))):
    print(fname)
```

Print a list of directories

  - Or we can construct a list

```
dirs = list(filter(os.path.isdir, glob.iglob(pattern)))
```

Get a list of directories

The *function* in **`filter`** can be the name of a function, or a **`lambda`** function. Here we have used `os.path.isdir` which returns true for each item that is a directory. The first example prints each filename that is a directory. `glob.iglob` does an expansion of the glob (wildcard) pattern and returns an iterator. The second example constructs a list of directory names.
The *list* in **`filter`** can be any iterable object, that is a list, tuple, string, or even a file.
As a special case, if the function is **`None`** then the items returned are those that resolve to true (rather like *grep*).
In Python 2 **`filter`** returned a list, in **Python 3** it returns an iterator.

List comprehensions

- **A list comprehension returns a list**
- **It consists of:**
  - An expression which identifies a list item
  - A loop - typically a `for` loop

```
pattern = 'C:/QA/Python/*'
sizes = [os.path.getsize(fname) for fname in glob.iglob(pattern)]
```

Get a list of file sizes

- An optional condition to filter items

→ *Pythonic* replacement of the `filter` built-in

```
dirs = [fname for fname in glob.iglob(pattern) if os.path.isdir(fname)]
```

Get a list of directories

A list comprehension is often a more natural way of expressing list operations. The original idea comes from set theory, and became popular in the Haskell language.
There are three parts to a list comprehension: the expression which will be executed for each item, the loop (usually a `for` loop), and an optional condition. The item will only be returned when the condition is true.
List comprehensions fit naturally into a world of functions which, increasingly, return iterators rather than lists.
The **`map`** and **`filter`** built-ins are their functional equivalents, but list comprehensions are more *Pythonic*.
PY3: In Python 3, the syntax was tightened to require a single (iterable) expression after the **in**. Hence, this worked in Python 2:

```
a = [i for i in 1,2,3]
```

but that gives a syntax error in Python 3. The correct notation in Python 3 (which also works in 2) is:

```
a = [i for i in (1,2,3)]
```

In Python 3, the loop variable is within its own scope, it will not affect a variable of the same name outside the comprehension (which is not the case in Python 2).

Set and dictionary comprehensions

Python 3 allows comprehensions on sets...

py3

```
myset = {'booboo', 'yogi', 'care', 'fozzie'}
results = {do_ftp(m) for m in myset}
```

```
ftp to 'fozzie'
ftp to 'yogi'
ftp to 'booboo'
ftp to 'care'
```

... and dictionaries

```
pattern = 'C:/*.py'
tsizes = [(fname, os.path.getsize(fname))
          for fname in glob.iglob(pattern)]
```

List of tuples

```
[('C:/first.py', 90), ('C:/think.py', 0), ('C:/try.py', 21)]
```

```
dsizes = {fname:size for fname, size in tsizes
                if size > 0}
```

Dictionary

```
{'C:/first.py': 90, 'C:/try.py': 21}
```

In addition to list comprehensions, Python 3 also supports set and dictionary comprehensions.
A set comprehension takes a set and iterates through it to produce another set. In the example, a user written function called `do_ftp()` (which probably uses the standard module **`ftplib`**) carries out its task on the four machines, in an unknown order. The `results` set will hold whatever the function returns on each occasion.
Dictionary comprehensions are twice as complex, in that they need a key, value tuple to feed. In the example, a list of two item tuples is created first, using a list comprehension. This is then used to construct our dictionary, which contains a further modifier (`if size > 0`).
This example *could* be done in one statement:

```
sizes={fn:os.path.getsize(fn) for fn in glob.iglob(pn)
       if os.path.getsize(fn) > 0}
```

The variable `fname` has been shortened to `fn`, and `pattern` shortened to `pn` (to fit).

Lazy lists

- **Generating lists in memory can be an overhead**
  - How big is a list?
  - What about sequences that have no end?
- **Lazy lists only return a value when it is needed**
  - One item at a time, as and when required
- **Particularly suitable when iterators are used**
  - An iterator function returns items one at a time
- **Many Python 3 functions return iterators rather than lists**
  - map(), filter(), range(), reversed(), zip(), and so on

Historically, many functions used or generated in-memory lists for iteration. This was fine for relatively small data sets, but when the list becomes large then the overhead is unacceptable. Enter the lazy list.
A lazy list is when the next item is supplied when needed, and not before. If we stop processing before reaching the end of the list, then those unused items will never be generated or use up resources. Lazy lists are implemented by supplying an iterator, i.e. position, to the list, rather than the whole list itself.

Generators

A generator is a function which yields a lazy list

- A lazy list item is returned at the `yield` statement

```
def get_dir(path):
    pattern = os.path.join(path, '*')
    for file in glob.iglob(pattern):
        if os.path.isdir(file):
            yield file
```

- Generators can often replace list comprehensions
- Can be used anywhere an iterator is expected

```
for dir in get_dir('C:/QA/Python'):
    print(dir)
```

Print a list of directories

```
dirs = list(get_dir('C:/QA/Python'))
```

Get a list of directories

A generator can be used to perform a lazy evaluation, often replacing list comprehensions.
In each example on the slide, the function is entered just once. The **`yield`** statement temporarily suspends operation of the loop within the function and returns an intermediate value which supplies the next item in the list.
Executing a **`return`** from within a loop containing a **`yield`** expression will give a `SyntaxError`, you should **`break`** out of the loop instead.
Contrast this with the code on the slides for filters and list comprehensions. In those structures, a temporary result list was created in memory. With generators and lazy lists, only the result immediately being processed is held.
Note: the following imports have been omitted for clarity:

```
import os.path
import glob
```

List comprehensions as generators

- **A list comprehension may be used instead of `yield`**
  - Sometimes - this does not support `send`ing values
  - Enclose the comprehension in `()` instead of `[]`

```
def get_dir(path):
    pattern = os.path.join(path, '*')
    for file in glob.iglob(pattern):
        if os.path.isdir(file):
            yield file
```

- Rewritten as a list comprehension:
- Function returns a generator object, as before

```
def get_dir(path):
    pattern = os.path.join(path, '*')
    return (file
            for file in glob.iglob(pattern)
            if os.path.isdir(file))
```

py3

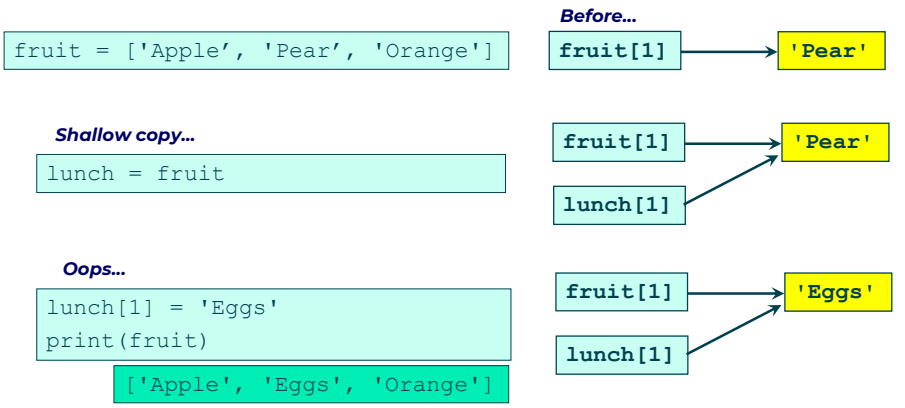
In Python 3*, we have an alternative syntax to yield, and that is to use a list comprehension instead. The syntactical difference is the use of rounded brackets (parentheses) instead of squared brackets.
The slide shows an earlier example of **`yield`**, with the list comprehension equivalent. The two code fragments are functionally the same, both return a generator object. An obvious downside is that there is no way for the caller to return a value to the generator function, as there is with **`yield`**.

* actually it was introduced into 2.6

Copying collections - problem

Any problems with assignments?

- Remember that Python objects are references



Since Python uses reference objects, copying one container (collection) to another can give unexpected results. An assignment copies the references, not the data to which they are referring. All is well with *immutable* objects, such as string literals, integers or tuples, but *mutable* objects (by definition) can be altered, and that can have unexpected effects, as shown.
During an assignment of one collection to another, only a shallow copy is done (we are using lists here, but this applies to any collection object).
Don't panic, there is a simple solution...

Copying collections - slice solution?

- **For a sequence, take a slice**

```
fruit  = ['Apple', 'Pear', 'Orange']
lunch = fruit[:]
lunch[1] = 'Eggs'
print('fruit:', fruit,'\nlunch:', lunch)
```

```
fruit: ['Apple', 'Pear', 'Orange']
lunch: ['Apple', 'Eggs', 'Orange']
```

- **We need a better solution for more complex structures**
- A slice is still a shallow copy

```
fruit = ['knife', 'plate', ['Apple', 'Pear', 'Orange']]
lunch = fruit[:]
lunch[2][1] = 'Eggs'
print('fruit:', fruit, '\nlunch:', lunch)
```

```
fruit: ['knife', 'plate', ['Apple', 'Eggs', 'Orange']]
lunch: ['knife', 'plate', ['Apple', 'Eggs', 'Orange']]
```

For a simple sequence, as we saw in the previous example, the solution is easy to implement: use a slice. The rather strange syntax of an empty slice means to take all the elements. A slice is always an independent copy of the original.
Unfortunately, this is not good enough when we have a more complex data structure - even though we don't have to get *too* complex for the slice solution to fail.
In the second example, the assignment of `fruit[:]` to `lunch` will make an independent copy of `fruit`, but only the *reference* to the nested list will be copied, not the nested list itself.

Copying collections - deepcopy solution

- **A better solution for more complex structures**
- **The copy module, distributed with Python**
  - Can do a shallow copy or a deep-copy

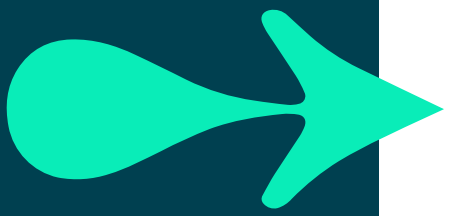
```
import copy
fruit = ['knife', 'plate', ['Apple', 'Pear', 'Orange']]
lunch = copy.deepcopy(fruit)
lunch[2][1] = 'Eggs'
print('fruit:', fruit, '\nlunch:', lunch)
```

```
fruit: ['knife', 'plate', ['Apple', 'Pear', 'Orange']]
lunch: ['knife', 'plate', ['Apple', 'Eggs', 'Orange']]
```

- **Beware! "copy" usually means a shallow copy**

To solve the problem of nested collections we need to do a deep copy, and a standard module, called **copy**, is provided for that.
There are two methods exposed, **copy** (which does a shallow copy) and **deepcopy**.
Not all objects can be copied using the **copy** module, those include other modules, class objects, functions and methods, files, and so on.
When we use the term "copy" we usually mean a shallow-copy, so be careful!

SUMMARY



- **filter() returns items that are true**
- Maybe with the help of a lambda
- **List comprehensions replace filter() and map()**
- Possibly with the help of a lambda
- **Generators yield values as they are needed**
- **Generators can replace list comprehensions**
- **Copying collections might not be a simple assignment**
- A deep copy might be required

Generator objects and next

- **A generator function returns a generator object**
- Can be used when a 'for' loop is not appropriate

```
gen = get_dir('C:/QA/Python')
```

Using the generator function get_dir from earlier

The `next` built-in gets the next item from a generator

```
while True:
    name = next(gen, False)
    if name: print(name)
    else: break
```

```
C:/QA/Python\Appendicies
C:/QA/Python\bak
```

A loop does not have to be used

```
gen = get_dir('C:/QA/Python')
dir1 = next(gen, False)
dir2 = next(gen, False)
dir3 = next(gen, False)
```

Generator objects can be created from a generator function call. Each time a generator function is called the iteration is restarted. If used outside a loop, the **`next`** built-in will get the next yield item. Alternatively, call the `__next__` method on the generator object, for example `gen.__next__()` (this is done by the `for` loop). The optional second parameter to `next()` gives the value returned when the generator sequence ends.

Co-routines and send() method

Data can be returned to the generator using `send`

```
import glob
import os
import os.path

def get_dir(path):
    while True:
        pattern = os.path.join(path,'*')
        path = None
        for file in glob.iglob(pattern):
            if os.path.isdir(file):
                path = yield file
                if path: break

        if not path: break
    return

gen = get_dir('C:/QA/Python')

print(next(gen))
print(next(gen))
print(gen.send('C:/MinGW'))
print(next(gen))
```

```
C:/QA/Python\AdvancedPython
C:/QA/Python\Appendicies
C:/MinGW\bin
C:/MinGW\dist
```

Both `next()` and *gen*`.send()` get the next yielded value

The `send()` generator method can be used instead of `next()`, the difference is that `send()` also sends a value back to the generator function, which can be picked-up as the return value from `yield()`.

When the `send()` method is not used, `yield` returns `None`.

These expressions were introduced at Python 2.5 and are called coroutines. They are described in PEP 342 - *Coroutines via Enhanced Generators*.

In the example on the slide, the generator function (`get_dir()`) searches for sub-directories in the given directory. After reporting the first two, it is sent, as the return value from `yield()`, a different directory to search.

Generator delegation (3.3)

Generator delegation allows a large and complex generator to be decomposed into *sub-generators*

- In the same way as a large and complex function might be split into smaller components

Simplistically:

`yield from` *iterable*

```
for file in glob.iglob(pattern):
    yield file
```

py3

Can be written as:

```
yield from glob.iglob(pattern)
```

Generators, like any other code, can get unwieldy. The **`yield from`** syntax introduced at 3.3 allows us to delegate to a sub-generator, which means we can split-up (decompose) complex code.
Most useful in closures and sub-generators (generators called from other generators).
See PEP 380 for further details and examples.