More Functional Programming

Julian Singkham, Joshua Goldshteyn, Devin Illy

Modularity

A practical benefit of functional programming is that it forces you to break apart your problem into small pieces. Programs are more modular as a result. It's easier to specify and write a small function that does one thing than a large function that performs a complicated transformation. Small functions are also easier to read and to check for errors.

Ease of debugging and testing

Testing and debugging a functional-style program is easier.

Debugging is simplified because functions are generally small and clearly specified. When a program doesn't work, each function is an interface point where you can check that the data are correct. You can look at the intermediate inputs and outputs to quickly isolate the function that's responsible for a bug.

Testing is easier because each function is a potential subject for a unit test. Functions don't depend on system state that needs to be replicated before running a test; instead you only have to synthesize the right input and then check that the output matches expectations.

Composability

As you work on a functional-style program, you'll write a number of functions with varying inputs and outputs. Some of these functions will be unavoidably specialized to a particular application, but others will be useful in a wide variety of programs. For example, a function that takes a directory path and returns all the XML files in the directory, or a function that takes a filename and returns its contents, can be applied to many different situations.

Over time you'll form a personal library of utilities. Often you'll assemble new programs by arranging existing functions in a new configuration and writing a few functions specialized for the current task.

Iterators

An iterator is an object representing a stream of data; this object returns the data one element at a time. A Python iterator must support a method called `__next__()` that takes no arguments and always returns the next element of the stream. If there are no more elements in the stream, `__next__()` must raise the StopIteration exception. Iterators don't have to be finite; it's perfectly reasonable to write an iterator that produces an infinite stream of data.

The built-in `iter()` function takes an arbitrary object and tries to return an iterator that will return the object's contents or elements, raising TypeError if the object doesn't support iteration. Several of Python's built-in data types support iteration, the most common being lists and dictionaries. An object is called iterable if you can get an iterator for it.

You can experiment with the iteration interface manually:

In [1]:
```
L = [1, 2, 3, 4]
it = iter(L)
it  #doctest: +ELLIPSIS
```

Out[1]:
```
<list_iterator at 0x133833346a0>
```

In [2]:
```
print(it.__next__())  # same as next(it)
print(next(it))
```

```
1
2
```

In [3]:
```
print(next(it))
```

```
3
```

In [4]:
```
print(next(it))
```

```
4
```

Python expects iterable objects in several different contexts, the most important being the for statement. In the statement for X in Y, Y must be an iterator or some object for which iter() can create an iterator. These two statements are equivalent:

```
for i in iter(obj):
    print(i)

for i in obj:
    print(i)
```

Iterators can be materialized as lists or tuples by using the list() or tuple() constructor functions:

```
>>> L = [1, 2, 3]
>>> iterator = iter(L)
>>> t = tuple(iterator)
>>> t
(1, 2, 3)
```

Sequence unpacking also supports iterators: if you know an iterator will return N elements, you can unpack them into an N-tuple:

```
>>> L = [1, 2, 3]
>>> iterator = iter(L)
>>> a, b, c = iterator
>>> a, b, c
(1, 2, 3)
```

Data Types That Support Iterators

We've already seen how lists and tuples support iterators. In fact, any Python sequence type, such as strings, will automatically support creation of an iterator.

Calling iter() on a dictionary returns an iterator that will loop over the dictionary's keys:

In [5]:
```
m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'Jun': 6,
     'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12}

for key in m:
    print(key, m[key])
```

```
Jan 1
Feb 2
Mar 3
Apr 4
May 5
Jun 6
Jul 7
Aug 8
Sep 9
Oct 10
Nov 11
Dec 12
```

Generator expressions and list comprehensions

Two common operations on an iterator's output are 1) performing some operation for every element, 2) selecting a subset of elements that meet some condition. For example, given a list of strings, you might want to strip off trailing whitespace from each line or extract all the strings containing a given substring.

List comprehensions and generator expressions are a concise notation for such operations, borrowed from the functional programming language Haskell (https://www.haskell.org/ (https://www.haskell.org/)). You can strip all the whitespace from a stream of strings with the following code:

In [6]:
```
line_list = ['  line 1\n', 'line 2  \n']

# Generator expression -- returns iterator
stripped_iter = (line.strip() for line in line_list)
print(list(stripped_iter))

# List comprehension -- returns list
stripped_list = [line.strip() for line in line_list]
print(stripped_list)
```

```
['line 1', 'line 2']
['line 1', 'line 2']
```

You can select only certain elements by adding an "if" condition:

In [7]:
```
stripped_list = [line.strip() for line in line_list if line != ""]
print(stripped_list)
```

```
['line 1', 'line 2']
```

Generators

Generators are a special class of functions that simplify the task of writing iterators. Regular functions compute a value and return it, but generators return an iterator that returns a stream of values.

When you call a function in Python, it gets a private namespace where its local variables are created. When the function reaches a `return` statement, the local variables are destroyed and the value is returned to the caller. A later call to the same function creates a new private namespace and a fresh set of local variables.

What if the local variables weren't thrown away on exiting a function? What if you could later resume the function where it left off? This is what generators provide; they can be thought of as resumable functions.

Here's the simplest example of a generator function:

In [8]:
```
def generate_ints(N):
    for i in range(N):
        yield i
```

Any function containing a `yield` keyword is a generator function; this is detected by Python's `bytecode` compiler which compiles the function specially as a result.

When you call a generator function, it doesn't return a single value; instead it returns a generator object that supports the iterator protocol. On executing the `yield` expression, the generator outputs the value of `i`, similar to a `return` statement. The big difference between `yield` and a `return` statement is that on reaching a `yield` the generator's state of execution is suspended and local variables are preserved. On the next call to the generator's `__next__()` method, the function will resume executing.

In the cell below, call the generate_ints() function several times to observe this generator behavior.

In [9]:
```
generate_test = generate_ints(10)
print(generate_test.__next__())
print(generate_test.__next__())
print(generate_test.__next__())
print(generate_test.__next__())
print(generate_test.__next__())
print(generate_test.__next__())
```

```
0
1
2
3
4
5
```

Built-in functions

Let's look in more detail at built-in functions often used with iterators.

Two of Python's built-in functions, `map()` and `filter()` duplicate the features of generator expressions:

`map(f, iterA, iterB, ...)` returns an iterator over the sequence

In [10]:
```
def upper(s):
    return s.upper()

# map function
print(list(map(upper, ['sentence', 'fragment'])))

# list comprehension
print([upper(s) for s in ['sentence', 'fragment']])
```

```
['SENTENCE', 'FRAGMENT']
['SENTENCE', 'FRAGMENT']
```

`filter(predicate, iter)` returns an iterator over all the sequence elements that meet a certain condition, and is similarly duplicated by list comprehensions. A **predicate** is a function that returns the truth value of some condition; for use with `filter()`, the predicate must take a single value.

In [11]:
```
def is_even(x):
    return (x % 2) == 0

# filter function
print(list(filter(is_even, range(10))))

# list comprehension
print(list(x for x in range(10) if is_even(x)))
```

```
[0, 2, 4, 6, 8]
[0, 2, 4, 6, 8]
```

`enumerate(iter, start=0)` counts off the elements in the iterable returning 2-tuples containing the count (from start) and each element.

In [12]:
```
for item in enumerate(['subject', 'verb', 'object']):
    print(item)
```

```
(0, 'subject')
(1, 'verb')
(2, 'object')
```

`enumerate()` is often used when looping through a list and recording the indexes at which certain conditions are met:

```
f = open('data.txt', 'r')
for i, line in enumerate(f):
    if line.strip() == '':
        print('Blank line at line #%i' % i)
```

`sorted(iterable, key=None, reverse=False)` collects all the elements of the iterable into a list, sorts the list, and returns the sorted result. The key and reverse arguments are passed through to the constructed list's `sort()` method.

In [13]:
```
import random
# Generate 8 random numbers between [0, 10000)
rand_list = random.sample(range(10000), 8)

print(rand_list)
print(sorted(rand_list))
print(sorted(rand_list, reverse=True))
```

```
[1041, 128, 8553, 9146, 8688, 356, 2304, 8097]
[128, 356, 1041, 2304, 8097, 8553, 8688, 9146]
[9146, 8688, 8553, 8097, 2304, 1041, 356, 128]
```

`zip(iterA, iterB, ...)` takes one element from each iterable and returns them in a tuple:

In [14]:
```
print(list(zip(['a', 'b', 'c'], (1, 2, 3))))
```

```
[('a', 1), ('b', 2), ('c', 3)]
```

It doesn't construct an in-memory list and exhaust all the input iterators before returning; instead tuples are constructed and returned only if they're requested. (The technical term for this behavior is lazy evaluation.)

This iterator is intended to be used with iterables that are all of the same length. If the iterables are of different lengths, the resulting stream will be the same length as the shortest iterable.

In [15]:
```
print(list(zip(['a', 'b'], (1, 2, 3))))
```

```
[('a', 1), ('b', 2)]
```