Functional Programming

Functional Programming: Overview

We may say that there are three commonly used styles (sometimes called *paradigms*) of program design: *imperative/procedural*, *object-oriented* and ***functional***.

Here we'll tackle a common problem (summing a sequence of integers, or an *arithmetic series*) using each style:

imperative or **procedural** involves a series of statements along with variables that change as a result. We call these variable values the program's *state*.

```
mysum = 0
for counter in range(11):
    mysum = mysum + counter

print mysum
```

object-oriented uses *object state* to produce outcome.

```
class Summer(object):
    def __init__(self):
        self.sum = 0
    def add(self, num):
        self.sum = self.sum + num

s = Summer()
for num in range(11):
    s.add(num)

print s.sum
```

functional combines *pure functions* to produce outcome. No *state change* is involved.

```
print sum(range(11))
```

A pure function is one that only handles input, output and its own variables -- it does not affect nor is it affected by global or other variables existing outside the function.

Because of this "air-tightness", functional programming can be tested more reliably than the other styles.

Some languages are designed around a single style or paradigm. But since Python is a "multi-paradigm" language, it is possible to use it to code in any of these styles.

To employ functional programming in our own programs, we need only seek to replace imperative code with functional code, combining pure functions in ways that replicate some of the patterns that we use to iterate, summarize, compute, etc.

After some experience coding in this style, you may recognize patterns for iteration, accumulation, etc. and more readily employ them in your programs, making them more predictable, testable and less prone to error.

Note: Python documentation provides a solid overview (https://docs.python.org/2/howto/functional.html) of functional programming in Python.

Mary Rose Cook provides a plain language introduction (https://maryrosecook.com/blog/post/a-practical-introduction-to-functional-programming) to functional programming.

O'Reilly publishes a free e-book (http://www.oreilly.com/programming/free/functional-programming-python.csp) with a comprehensive review of functional programming by Python luminary David Mertz.

Review: lambdas

Lambda functions are simply inline functions -- they can be defined entirely within a single statement, within a container initialization, etc.

Lambdas are most often used inside functions like **sorted()**:

```
# sort a list of names by last name
names = [ 'Josh Peschko', 'Gabriel Feghali', 'Billy Woods', 'Arthur Fischer-Zernin' ]
sortednames = sorted(names, key=lambda name:  name.split()[1])

# sort a list of CSV lines by the 2nd column in the file
slines = sorted(lines, lambda x: x.split(',')[2])
```

We will see lambdas used in other functions such as map(), filter() and reduce().

Review: list comprehensions; set comprehensions and dict comprehensions

List, set and dict comprehensions can filter or transform sequences in a single statement.

Functional programming (and algorithms in general) often involves the processing of sequences. List comprehensions provide a flexible way to filter and modify values within a list.

list comprehension: return a list

```
nums = [1, 2, 3, 4, 5]
dblnums = [ val * 2 for val in nums ]
print dblnums                               # [2, 4, 6, 8, 10]

print [ val * 2 for val in nums if val > 2]   # [6, 8, 10]
```

set comprehension: return a set

```
states = { line.split(':')[3]
          for line in open('student_db.txt').readlines()[1:] }
```

dict comprehension: return a dict

```
student_states = { line.split(':')[0]: line.split(':')[3]
                  for line in open('student_db.txt').readlines()[1:] }
```

map() and filter() as alternatives to list comprehensions

Although list comprehensions have nominally replaced **map()** and **filter()**, these functions are still used in many functional programming algorithms.

map(): apply a transformation function to each item in a sequence

```
# square some integers
sqrd = map(lambda x: x ** 2, range(6))    # [1, 4, 9, 16, 25]

# get string lengths
lens = map(len, ['some', 'words', 'to', 'get', 'lengths', 'from'])
print lens    # [4, 5, 2, 3, 7, 4]
```

filter(): apply a filtering function to each item in a sequence

```
pos = filter(lambda x: x > 0, [-5, 2, -3, 17, 6, 4, -9])
print pos     # [2, 17, 6, 4]
```

reduce() for accumulation of values

Like map() or filter(), **reduce()** applies a function to each item in a sequence, but *accumulates* a value as it iterates.

It accumulates values through a second variable to its processing function. In the below examples, the accumulator is **a** and the current value of the iteration is **x**. **a** grows through the accumulation, as if the function were saying **a = a + x** or **a = a * x**.

Here is our arithmetic series for integers 1-10, done with reduce():

```
def addthem(a, x):
    return a + x

intsum = reduce(addthem, range(1, 11))

# same using a lambda
intsum = reduce(lambda a, x: a + x, range(1, 11))
```

Just as easily, a factorial of integers 1-10:

```
facto = reduce(lambda a, x: a * x, range(1, 11))      # 3628800
```

default value

Since **reduce()** has to start with a value in the accumulator, it will attempt to begin with the first element in the source list. However, if each value is being transformed before being accumulated, the first computation may result in an error:

```
strsum = reduce(lambda a, x: a + int(x), ['1', '2', '3', '4', '5'])

TypeError: cannot concatenate 'str' and 'int' objects
```

This is apparently happening because python is trying to add **int('1')** to **''** (i.e., **reduce()** does not see the transform and so uses an 'empty' version of the type, in this case an empty string).

In these cases we can supply an initial value to **reduce()**, so it knows where to begin:

```
strsum = reduce(lambda a, x: a + int(x), ['1', '2', '3', '4', '5'], 0)
```

Higher-Order Functions

Any function that takes a function as an argument, or that returns a function as a return value, is a *higher-order function*.

map(), **filter()**, **reduce()**, **sorted** all take functions as arguments.

@properties, **@staticmethod**, **@classmethod** all take a given function as argument and return a modified function as a return value.

any() and all(): return True based on truth of elements

Also iterating functions,

any(): return True if any elements are True

```
any([1, 0, 2, 0, 3])        # True:  at least one item is True

any([0, [], {}, ''])        # False: none of the items is True
```

all(): return True if all elements are True

```
all([1, 5, 0.0001, 1000])   # True:  all items are True

all([1, 5, 9, 10, 0, 20])   # False:  one item is not True
```

generators

Generators are iterators that can calculate and generate any number of items.

Generators behave like iterators, except that they **yield** a value rather than **return** one, and *they remember the value of their variables* so that the next time the class' **next()** method is called, it picks up where the last left off (at the point of *yield*). .

As a generator is an iterator, **next()** calls the function again to produce the next item; and **StopIteration** causes the generator to stop iterating. (**next()** is called automatically by iterators like **for**.

Generators are particularly useful in producing a sequence of **n** values, i.e. not a fixed sequence, but an unlimited sequence. In this example we have prepared a generator that generates primes up to the specified limit.

```
def get_primes(num_max):
    """ prime number generator """
    candidate = 2
    found = []
    while True:
        if all(candidate % prime != 0 for prime in found):
            yield candidate
            found.append(candidate)
        candidate += 1
        if candidate >= num_max:
            raise StopIteration

my_iter = get_primes(100)
print my_iter.next()        # 2
print my_iter.next()        # 3
print my_iter.next()        # 5

for i in get_primes(100):
    print i
```

recursive functions

A recursive function calls itself until a condition has been reached.

Recursive functions are appropriate for processes that iterate over an unknown number of items or events. Such situations could include files within a directory tree, where *listing the directory* is performed over and over until all directories within a tree are exhausted; or similarly, visiting links to pages within a website, where *listing the links in a page* is performed repeatedly.

Recursion features three elements: a *recursive call*, which is a call by the function to itself; the function process itself; and a *base condition*, which is the point at which the chain of recursions finally returns.

In this example, we are presented with a tree of companies organized into a hierarchy of parent and child companies. We are interested in finding the "top parent" of any given company -- so if **Acme Pens'** parent is **Acme Office Supply** and taht company's parent is **Megacorp**, then **Acme Pens'** "top parent" is **Megacorp**.

The dict below shows each company's parent. If the company has no parent, its value is **None**.

```
parents = {
            'Megacorp':           None,

            'Acme Office Supply': 'Megacorp',
            'Acme Paper':         'Acme Office Supply',
            'Acme Pens':          'Acme Office Supply',

            'Best Electronics':   None,
            'Best Audio':         'Best Electronics',
            'Best TV':            'Best Electronics',

            'Celera Publishing':  'Megacorp',
            'Celera Books':       'Celera Publishing',
            'Celera Web':         'Celera Publishing'
                                                      }

def top_parent(company_id):
    if ( company_id not in parents or       # base case (no parent):  return
         not parents[company_id] ):

        return company_id

    return top_parent(parents[company_id])  # recursive call with parent id


for id in sorted(parents.keys()):
    print '{}:  {}'.format(id, top_parent(id))
```

The above outputs:

```
Acme Office Supply:  Megacorp
Acme Paper:          Megacorp
Acme Pens:           Megacorp
Best Audio:          Best Electronics
Best Electronics:    Best Electronics
Best TV:             Best Electronics
Celera Books:        Megacorp
Celera Publishing:   Megacorp
Celera Web:          Megacorp
Megacorp:            Megacorp
```

Here's a solution to the factorial problem using recursion:

```
def factorial(n):
   if n < 1:                              # base case (reached 0):  returns
       return 1
   else:
       return_num = n * factorial(n - 1)  # recursive call
       return return_num

print factorial(5)       # 120
```