Functions

February 21, 2015

```
In [1]: import os
        import sys
        import glob
        import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        %matplotlib inline
        %precision 4
```

```
Out[1]: u'%.4f'
```

References:
Functional Programming HOWTO

0.1 Functions are first class objects

In Python, functions behave like any other object, such as an int or a list. That means that you can use functions as arguments to other functions, store functions as dictionary values, or return a function from another function. This leads to many powerful ways to use functions.

```
In [2]: def square(x):
            """Square of x."""
            return x*x

        def cube(x):
            """Cube of x."""
            return x*x*x
```

```
In [3]: # create a dictionary of functions

        funcs = {
            'square': square,
            'cube': cube,
        }
```

```
In [4]: x = 2

        print square(x)
        print cube(x)

        for func in sorted(funcs):
            print func, funcs[func](x)
```

```
4
8
cube 8
square 4
```

0.2 Function argumnents

This is caution to be careful of how Python treats function arguments.

0.2.1 Call by "object reference"

Some data types, such as strings and tuples, cannot be directly modified and are called immutable. Atomic variables such as integers or floats are always immutable. Other datatypes, such as lists and dictionaries, can be directly modified and are called mutable. Passing mutable variables as function arguments can have different outcomes, depedning on what is done to the variable inside the function. When we call

```
x = [1,2,3] # mutable
f(x)
```

what is passsed to the function is a *copy* of the *name* `x` that refers to the content (a list) `[1, 2, 3]`. If we use this copy of the name to change the content directly (e.g. `x[0] = 999`) within the function, then `x` chanes *outside* the funciton as well. However, if we reassgne `x` within the function to a new object (e.g. another list), then the copy of the name `x` now points to the new object, but `x` outside the function is unhcanged.

```
In [10]: def transmogrify(x):
             x[0] = 999
             return x

         x = [1,2,3]
         print x
         print transmogrify(x)
         print x
```

```
[1, 2, 3]
[999, 2, 3]
[999, 2, 3]
```

```
In [11]: def no_mogrify(x):
             x = [4,5,6]
             return x

         x = [1,2,3]
         print x
         print no_mogrify(x)
         print x
```

```
[1, 2, 3]
[4, 5, 6]
[1, 2, 3]
```

0.2.2 Binding of default arguments occurs at function *definition*

```
In [13]: def f(x = []):
             x.append(1)
             return x

         print f()
         print f()
         print f()
         print f(x = [9,9,9])
         print f()
         print f()
```

```
[1]
[1, 1]
[1, 1, 1]
[9, 9, 9, 1]
[1, 1, 1, 1]
[1, 1, 1, 1, 1]
```

```
In [14]: # Usually, this behavior is not desired and we would write

         def f(x = None):
             if x is None:
                 x = []
             x.append(1)
             return x

         print f()
         print f()
         print f()
         print f(x = [9,9,9])
         print f()
         print f()
```

```
[1]
[1]
[1]
[9, 9, 9, 1]
[1]
[1]
```

However, sometimes in advanced usage, the behavior is intetnional. See http://effbot.org/zone/default-values.htm for details.

0.3 Higher-order functions

A function that uses another function as an input argument or returns a function (HOF) is known as a higher-order function. The most familiar examples are `map` and `filter`.

```
In [5]: # The map function applies a function to each member of a collection

        map(square, range(5))
```

```
Out[5]: [0, 1, 4, 9, 16]
```

```
In [6]: # The filter function applies a predicate to each memmber of a collection,
        # retaining only those members where the predicate is True

        def is_even(x):
            return x%2 == 0

        filter(is_even, range(5))
```

```
Out[6]: [0, 2, 4]
```

```
In [7]: # It is common to combine map and filter

        map(square, filter(is_even, range(5)))
```

Out[7]: [0, 4, 16]

In [8]:
```
# The reduce function reduces a collection using a binary operator to combine items two at a ti

def my_add(x, y):
    return x + y

# another implementation of the sum function
reduce(my_add, [1,2,3,4,5])
```

Out[8]: 15

In [10]:
```
# Custom functions can of couse, also be HOFs

def custom_sum(xs, transform):
    """Returns the sum of xs after a user specified transform."""
    return sum(map(transform, xs))

xs = range(5)
print custom_sum(xs, square)
print custom_sum(xs, cube)
```

```
30
100
```

In [11]:
```
# Returning a function is also useful

# A closure
def make_logger(target):
    def logger(data):
        with open(target, 'a') as f:
            f.write(data + '\n')
    return logger

foo_logger = make_logger('foo.txt')
foo_logger('Hello')
foo_logger('World')
```

In [12]:
```
!cat 'foo.txt'
```

```
Hello
World
Hello
World
Hello
World
```

0.4 Anonymous functions

When using functional style, there is often the need to create small specific functions that perform a limited task as input to a HOF such as `map` or `filter`. In such cases, these functions are often written as `anonymous` or `lambda` functions. If you find it hard to understand what a `lambda` function is doing, it should probably be rewritten as a regular function.

In [37]:
```
# Using standard functions
```

```
def square(x):
    return x*x

print map(square, range(5))
```

```
[0, 1, 4, 9, 16]
```

```
In [38]: # Using an anonymous function

         print map(lambda x: x*x, range(5))
```

```
[0, 1, 4, 9, 16]
```

```
In [4]: # what does this function do?
        s1 = reduce(lambda x, y: x+y, map(lambda x: x**2, range(1,10)))
        print(s1)
        print

        # functional expressions and lambdas are cool
        # but can be difficult to read when over-used
        # Here is a more comprehensible version
        s2 = sum(x**2 for x in range(1, 10))
        print(s2)

        # we will revisit map-reduce when we look at high-performance computing
        # where map is used to distribute jobs to multiple processors
        # and reduce is used to calculate some aggreate function of the results
        # returned by map
```

```
285

285
```

0.5 Pure functions

Functions are pure if they do not have any *side effects* and do not depend on global variables. Pure functions are similar to mathematical functions - each time the same input is given, the same output will be returned. This is useful for reducing bugs and in parallel programming since each function call is independent of any other function call and hence trivially parallelizable.

```
In [13]: def pure(xs):
             """Make a new list and return that."""
             xs = [x*2 for x in xs]
             return xs
```

```
In [14]: xs = range(5)
         print "xs =", xs
         print pure(xs)
         print "xs =", xs
```

```
xs = [0, 1, 2, 3, 4]
[0, 2, 4, 6, 8]
xs = [0, 1, 2, 3, 4]
```

```
In [15]: def impure(xs):
             for i, x in enumerate(xs):
                 xs[i] = x*2
             return xs
```

```
In [16]: xs = range(5)
         print "xs =", xs
         print impure(xs)
         print "xs =", xs
```

```
xs = [0, 1, 2, 3, 4]
[0, 2, 4, 6, 8]
xs = [0, 2, 4, 6, 8]
```

```
In [17]: # Note that mutable functions are created upon function declaration, not use.
         # This gives rise to a common source of beginner errors.

         def f1(x, y=[]):
             """Never give an empty list or other mutable structure as a default."""
             y.append(x)
             return sum(y)
```

```
In [18]: print f1(10)
         print f1(10)
         print f1(10, y =[1,2])
```

```
10
20
13
```

```
In [19]: # Here is the correct Python idiom

         def f2(x, y=None):
             """Check if y is None - if so make it a list."""
             if y is None:
                 y = []
             y.append(x)
             return sum(y)
```

```
In [20]: print f1(10)
         print f1(10)
         print f1(10, y =[1,2])
```

```
30
40
13
```

0.6 Recursion

A recursive function is one that calls itself. Recursive functions are extremely useful examples of the divide-and-conquer paradigm in algorithm development and are a direct expression of finite diffference equations. However, they can be computationally inefficient and their use in Python is quite rare in practice.

Recursive functions generally have a set of *base cases* where the answer is obvious and can be returned immediately, and a set of recursive cases which are split into smaller pieces, each of which is given to the same function called recursively. A few examples will make this clearer.

```
In [17]: # The factorial function is perhaps the simplest classic example of recursion.

         def fact(n):
             """Returns the factorial of n."""
             # base case
```

```
        if n==0:
            return 1
        # recursive case
        else:
            return n * fact(n-1)

    print [fact(n) for n in range(10)]
```

```
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

```
In [18]: # The Fibonacci sequence is another classic recursion example

         def fib1(n):
             """Fib with recursion."""

             # base case
             if n==0 or n==1:
                 return 1
             # recurssive caae
             else:
                 return fib1(n-1) + fib1(n-2)

         print [fib1(i) for i in range(10)]
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

```
In [19]: # In Python, a more efficient version that does not use recursion is

         def fib2(n):
             """Fib without recursion."""
             a, b = 0, 1
             for i in range(1, n+1):
                 a, b = b, a+b
             return b

         print [fib2(i) for i in range(10)]
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

```
In [20]: # Note that the recursive version is much slower than the non-recursive version

         %timeit fib1(20)
         %timeit fib2(20)

         # this is because it makes many duplicate function calls
         # Note duplicate calls to fib(2) and fib(1) below
         # fib(4) -> fib(3), fib(2)
         # fib(3) -> fib(2), fib(1)
         # fib(2) -> fib(1), fib(0)
         # fib(1) -> 1
         # fib(0) -> 1
```

```
100 loops, best of 3: 5.68 ms per loop
100000 loops, best of 3: 2.82 µs per loop
```

In [29]:
```
# Use of cache to speed up the recursive version.
# Note biding of the (mutable) dictionary as a default at run-time.

def fib3(n, cache={0: 1, 1: 1}):
    """Fib with recursion and caching."""

    try:
        return cache[n]
    except KeyError:
        cache[n] = fib3(n-1) + fib3(n-2)
        return cache[n]

print [fib3(i) for i in range(10)]

%timeit fib1(20)
%timeit fib2(20)
%timeit fib3(20)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
100 loops, best of 3: 5.68 ms per loop
100000 loops, best of 3: 2.82 µs per loop
1000000 loops, best of 3: 256 ns per loop
```

In [25]:
```
# Recursion is used to show off the divide-and-conquer paradigm

def almost_quick_sort(xs):
    """Almost a quick sort."""

    # base case
    if xs == []:
        return xs
    # recursive case
    else:
        pivot = xs[0]
        less_than = [x for x in xs[1:] if x <= pivot]
        more_than = [x for x in xs[1:] if x > pivot]
        return almost_quick_sort(less_than) + [pivot] + almost_quick_sort(more_than)

xs = [3,1,4,1,5,9,2,6,5,3,5,9]
print almost_quick_sort(xs)
```

```
[1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9, 9]
```

0.7 Iterators

Iterators represent streams of values. Because only one value is consumed at a time, they use very little memory. Use of iterators is very helpful for working with data sets too large to fit into RAM.

In [26]:
```
# Iterators can be created from sequences with the built-in function iter()

xs = [1,2,3]
x_iter = iter(xs)

print x_iter.next()
print x_iter.next()
```

```
        print x_iter.next()
        print x_iter.next()
```

```
1
2
3
```

```
    ---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)

    <ipython-input-26-eb1a17442aa0> in <module>()
      7 print x_iter.next()
      8 print x_iter.next()
----> 9 print x_iter.next()


    StopIteration:
```

```
In [27]: # Most commonly, iterators are used (automatically) within a for loop
         # which terminates when it encouters a StopIteration exception

         x_iter = iter(xs)
         for x in x_iter:
             print x
```

```
1
2
3
```

0.8 Generators

Generators create iterator streams.

```
In [28]: # Functions containing the 'yield' keyword return iterators
         # After yielding, the function retains its previous state

         def count_down(n):
             for i in range(n, 0, -1):
                 yield i
```

```
In [29]: counter = count_down(10)
         print counter.next()
         print counter.next()
         for count in counter:
             print count,
```

```
10
9
8 7 6 5 4 3 2 1
```

```
In [30]: # Iterators can also be created with 'generator expressions'
         # which can be coded similar to list generators but with parenthesis
         # in place of square brackets
```

```
xs1 = [x*x for x in range(5)]
print xs1

xs2 = (x*x for x in range(5))
print xs2

for x in xs2:
    print x,
print
```

```
[0, 1, 4, 9, 16]
<generator object <genexpr> at 0x1130a5820>
0 1 4 9 16
```

In [31]:
```
# Iterators can be used for infinte functions

def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a+b
```

In [32]:
```
for i in fib():
    # We must have a stopping condiiton since the generator returns an infinite stream
    if i > 1000:
        break
    print i,
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

In [38]:
```
# Many built-in Python functions return iterators
# including file handlers
# so with the idiom below, you can process a 1 terabyte file line by line
# on your laptop without any problem
# Inn Pyhton 3, map and filter return itnrators, not lists

for line in open('foo.txt'):
    print line,
```

```
Hello
World
Hello
World
Hello
World
```

0.8.1 Generators and comprehensions

In [39]:
```
# A geneeratorr expression

print (x for x in range(10))

# A list comprehesnnion

print [x for x in range(10)]
```

```
# A set comprehension

print {x for x in range(10)}

# A dictionary comprehension

print {x: x for x in range(10)}
```

```
<generator object <genexpr> at 0x102d8a410>
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}
```

0.8.2 Utilites - enumerate, zip and the ternary if-else operator

Two useful functions and an unusual operator.

```
In [34]: # In many programming languages, loops use an index.
         # This is possible in Python, but it is more
         # idiomatic to use the enumerate function.

         # using and index in a loop
         xs = [1,2,3,4]
         for i in range(len(xs)):
             print i, xs[i]
         print

         # using enumerate
         for i, x in enumerate(xs):
             print i, x
```

```
0 1
1 2
2 3
3 4

0 1
1 2
2 3
3 4
```

```
In [35]: # zip is useful when you need to iterate over matched elements of
         # multiple lists

         xs = [1, 2, 3, 4]
         ys = [10, 20, 30, 40]
         zs = ['a', 'b', 'c', 'd', 'e']

         for x, y, z in zip(xs, ys, zs):
             print x, y, z

         # Note that zip stops when the shortest list is exhausted
```

```
1 10 a
2 20 b
3 30 c
4 40 d
```

```
In [36]: # For list comprehensions, the ternary if-else operator is sometimes very useful

         [x**2 if x%2 == 0 else x**3 for x in range(10)]
```

```
Out[36]: [0, 1, 4, 27, 16, 125, 36, 343, 64, 729]
```

0.9 Decorators

Decorators are a type of HOF that take a function and return a wrapped function that provides additional useful properties.

Examples:

- logging
- profiling
- Just-In-Time (JIT) compilation

```
In [6]: # Here is a simple decorator to time an arbitrary function

        def func_timer(func):
            """Times how long the function took."""

            def f(*args, **kwargs):
                import time
                start = time.time()
                results = func(*args, **kwargs)
                print "Elapsed: %.2fs" % (time.time() - start)
                return results

            return f
```

```
In [7]: # There is a special shorthand notation for decorating functions

        @func_timer
        def sleepy(msg, sleep=1.0):
            """Delays a while before answering."""
            import time
            time.sleep(sleep)
            print msg

        sleepy("Hello", 1.5)
```

```
Hello
Elapsed: 1.50s
```

0.10 The operator module

The `operator` module provides “function” versions of common Python operators (+, *, [] etc) that can be easily used where a function argument is expected.

```
In [46]: import operator as op

         # Here is another way to express the sum function
         print reduce(op.add, range(10))

         # The pattern can be generalized
         print reduce(op.mul, range(1, 10))
```

```
45
362880
```

```
In [49]: my_list = [('a', 1), ('bb', 4), ('ccc', 2), ('dddd', 3)]

         # standard sort
         print sorted(my_list)

         # return list sorted by element at position 1 (remember Python counts from 0)
         print sorted(my_list, key=op.itemgetter(1))

         # the key argument is quite flexible
         print sorted(my_list, key=lambda x: len(x[0]), reverse=True)
```

```
[('a', 1), ('bb', 4), ('ccc', 2), ('dddd', 3)]
[('a', 1), ('ccc', 2), ('dddd', 3), ('bb', 4)]
[('dddd', 3), ('ccc', 2), ('bb', 4), ('a', 1)]
```

0.11 The `functools` module

The most useful function in the `functools` module is `partial`, which allows you to create a new function from an old one with some arguments "filled-in".

```
In [54]: from functools import partial

         sum_ = partial(reduce, op.add)
         prod_ = partial(reduce, op.mul)
         print sum_([1,2,3,4])
         print prod_([1,2,3,4])
```

```
10
24
```

```
In [58]: # This is extremely useful to create functions
         # that expect a fixed number of arguments

         import scipy.stats as stats

         def compare(x, y, func):
             """Returne p-value for some appropriate comparison test."""
             return func(x, y)[1]
```

```
In [83]: x, y = np.random.normal(0, 1, (100,2)).T

         print "p value assuming equal variance    =%.8f" % compare(x, y, stats.ttest_ind)
         test = partial(stats.ttest_ind, equal_var=False)
         print "p value not assuming equal variance=%.8f" % compare(x, y, test)
```

```
p value assuming equal variance    =0.19593077
p value not assuming equal variance=0.19599306
```

0.12 The `itertools` module

This provides many essential functions for working with iterators. The `permuations` and `combinations` generators may be particularly useful for simulations, and the `groupby` gnerator is useful for data analyiss.

```
In [84]: from itertools import cycle, groupby, islice, permutations, combinations

         print list(islice(cycle('abcd'), 0, 10))
         print

         animals = sorted(['pig', 'cow', 'giraffe', 'elephant',
                           'dog', 'cat', 'hippo', 'lion', 'tiger'], key=len)
         for k, g in groupby(animals, key=len):
             print k, list(g)
         print

         print [''.join(p) for p in permutations('abc')]
         print

         print [list(c) for c in combinations([1,2,3,4], r=2)]
```

```
['a', 'b', 'c', 'd', 'a', 'b', 'c', 'd', 'a', 'b']

3 ['pig', 'cow', 'dog', 'cat']
4 ['lion']
5 ['hippo', 'tiger']
7 ['giraffe']
8 ['elephant']

['abc', 'acb', 'bac', 'bca', 'cab', 'cba']

[[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]
```

0.13 The toolz, fn and funcy modules

If you wish to program in the functional style, check out the following packages

- toolz
- fn
- funcy

```
In [85]: # Here is a small example to convert the DNA of a
         # bacterial enzyme into the protein sequence
         # using the partition function to generate
         # cddons (3 nucleotides) for translation.

         codon_table = {
             'ATA':'I', 'ATC':'I', 'ATT':'I', 'ATG':'M',
             'ACA':'T', 'ACC':'T', 'ACG':'T', 'ACT':'T',
             'AAC':'N', 'AAT':'N', 'AAA':'K', 'AAG':'K',
             'AGC':'S', 'AGT':'S', 'AGA':'R', 'AGG':'R',
             'CTA':'L', 'CTC':'L', 'CTG':'L', 'CTT':'L',
             'CCA':'P', 'CCC':'P', 'CCG':'P', 'CCT':'P',
             'CAC':'H', 'CAT':'H', 'CAA':'Q', 'CAG':'Q',
             'CGA':'R', 'CGC':'R', 'CGG':'R', 'CGT':'R',
             'GTA':'V', 'GTC':'V', 'GTG':'V', 'GTT':'V',
             'GCA':'A', 'GCC':'A', 'GCG':'A', 'GCT':'A',
             'GAC':'D', 'GAT':'D', 'GAA':'E', 'GAG':'E',
             'GGA':'G', 'GGC':'G', 'GGG':'G', 'GGT':'G',
             'TCA':'S', 'TCC':'S', 'TCG':'S', 'TCT':'S',
```

```
    'TTC':'F', 'TTT':'F', 'TTA':'L', 'TTG':'L',
    'TAC':'Y', 'TAT':'Y', 'TAA':'_', 'TAG':'_',
    'TGC':'C', 'TGT':'C', 'TGA':'_', 'TGG':'W',
    }

gene = """
>ENA|BAE76126|BAE76126.1 Escherichia coli str. K-12 substr. W3110 beta-D-galactosidase
ATGACCATGATTACGGATTCACTGGCCGTCGTTTTACAACGTCGTGACTGGGAAAACCCT
GGCGTTACCCAACTTAATCGCCTTGCAGCACATCCCCCTTTCGCCAGCTGGCGTAATAGC
GAAGAGGCCCGCACCGATCGCCCTTCCCAACAGTTGCGCAGCCTGAATGGCGAATGGCGC
TTTGCCTGGTTTCCGGCACCAGAAGCGGTGCCGGAAAGCTGGCTGGAGTGCGATCTTCCT
GAGGCCGATACTGTCGTCGTCCCCTCAAACTGGCAGATGCACGGTTACGATGCGCCCATC
TACACCAACGTGACCTATCCCATTACGGTCAATCCGCCGTTTGTTCCCACGGAGAATCCG
ACGGGTTGTTACTCGCTCACATTTAATGTTGATGAAAGCTGGCTACAGGAAGGCCAGACG
CGAATTATTTTTGATGGCGTTAACTCGGCGTTTCATCTGTGGTGCAACGGGCGCTGGGTC
GGTTACGGCCAGGACAGTCGTTTGCCGTCTGAATTTGACCTGAGCGCATTTTTACGCGCC
GGAGAAAACCGCCTCGCGGTGATGGTGCTGCGCTGGAGTGACGGCAGTTATCTGGAAGAT
CAGGATATGTGGCGGATGAGCGGCATTTTCCGTGACGTCTCGTTGCTGCATAAACCGACT
ACACAAATCAGCGATTTCCATGTTGCCACTCGCTTTAATGATGATTTCAGCCGCGCTGTA
CTGGAGGCTGAAGTTCAGATGTGCGGCGAGTTGCGTGACTACCTACGGGTAACAGTTTCT
TTATGGCAGGGTGAAACGCAGGTCGCCAGCGGCACCGCGCCTTTCGGCGGTGAAATTATC
GATGAGCGTGGTGGTTATGCCGATCGCGTCACACTACGTCTGAACGTCGAAAACCCGAAA
CTGTGGAGCGCCGAAATCCCGAATCTCTATCGTGCGGTGGTTGAACTGCACACCGCCGAC
GGCACGCTGATTGAAGCAGAAGCCTGCGATGTCGGTTTCCGCGAGGTGCGGATTGAAAAT
GGTCTGCTGCTGCTGAACGGCAAGCCGTTGCTGATTCGAGGCGTTAACCGTCACGAGCAT
CATCCTCTGCATGGTCAGGTCATGGATGAGCAGACGATGGTGCAGGATATCCTGCTGATG
AAGCAGAACAACTTTAACGCCGTGCGCTGTTCGCATTATCCGAACCATCCGCTGTGGTAC
ACGCTGTGCGACCGCTACGGCCTGTATGTGGTGGATGAAGCCAATATTGAAACCCACGGC
ATGGTGCCAATGAATCGTCTGACCGATGATCCGCGCTGGCTACCGGCGATGAGCGAACGC
GTAACGCGAATGGTGCAGCGCGATCGTAATCACCCGAGTGTGATCATCTGGTCGCTGGGG
AATGAATCAGGCCACGGCGCTAATCACGACGCGCTGTATCGCTGGATCAAATCTGTCGAT
CCTTCCCGCCCGGTGCAGTATGAAGGCGGCGGAGCCGACACCACGGCCACCGATATTATT
TGCCCGATGTACGCGCGCGTGGATGAAGACCAGCCCTTCCCGGCTGTGCCGAAATGGTCC
ATCAAAAAATGGCTTTCGCTACCTGGAGAGACGCGCCCGCTGATCCTTTGCGAATACGCC
CACGCGATGGGTAACAGTCTTGGCGGTTTCGCTAAATACTGGCAGGCGTTTCGTCAGTAT
CCCCGTTTACAGGGCGGCTTCGTCTGGGACTGGGTGGATCAGTCGCTGATTAAATATGAT
GAAAACGGCAACCCGTGGTCGGCTTACGGCGGTGATTTTGGCGATACGCCGAACGATCGC
CAGTTCTGTATGAACGGTCTGGTCTTTGCCGACCGCACGCCGCATCCAGCGCTGACGGAA
GCAAAACACCAGCAGCAGTTTTTCCAGTTCCGTTTATCCGGGCAAACCATCGAAGTGACC
AGCGAATACCTGTTCCGTCATAGCGATAACGAGCTCCTGCACTGGATGGTGGCGCTGGAT
GGTAAGCCGCTGGCAAGCGGTGAAGTGCCTCTGGATGTCGCTCCACAAGGTAAACAGTTG
ATTGAACTGCCTGAACTACCGCAGCCGGAGAGCGCCGGGCAACTCTGGCTCACAGTACGC
GTAGTGCAACCGAACGCGACCGCATGGTCAGAAGCCGGGCACATCAGCGCCTGGCAGCAG
TGGCGTCTGGCGGAAAACCTCAGTGTGACGCTCCCCGCCGCGTCCCACGCCATCCCGCAT
CTGACCACCAGCGAAATGGATTTTTGCATCGAGCTGGGTAATAAGCGTTGGCAATTTAAC
CGCCAGTCAGGCTTTCTTTCACAGATGTGGATTGGCGATAAAAAACAACTGCTGACGCCG
CTGCGCGATCAGTTCACCCGTGCACCGCTGGATAACGACATTGGCGTAAGTGAAGCGACC
CGCATTGACCCTAACGCCTGGGTCGAACGCTGGAAGGCGGCGGGCCATTACCAGGCCGAA
GCAGCGTTGTTGCAGTGCACGGCAGATACACTTGCTGATGCGGTGCTGATTACGACCGCT
CACGCGTGGCAGCATCAGGGGAAAACCTTATTTATCAGCCGGAAAACCTACCGGATTGAT
GGTAGTGGTCAAATGGCGATTACCGTTGATGTTGAAGTGGCGAGCGATACACCGCATCCG
GCGCGGATTGGCCTGAACTGCCAGCTGGCGCAGGTAGCAGAGCGGGTAAACTGGCTCGGA
TTAGGGCCGCAAGAAAACTATCCCGACCGCCTTACTGCCGCCTGTTTTGACCGCTGGGAT
CTGCCATTGTCAGACATGTATACCCCGTACGTCTTCCCGAGCGAAAACGGTCTGCGCTGC
```

```
GGGACGCGCGAATTGAATTATGGCCCACACCAGTGGCGCGGCGACTTCCAGTTCAACATC
AGCCGCTACAGTCAACAGCAACTGATGGAAACCAGCCATCGCCATCTGCTGCACGCGGAA
GAAGGCACATGGCTGAATATCGACGGTTTCCATATGGGGATTGGTGGCGACGACTCCTGG
AGCCCGTCAGTATCGGCGGAATTCCAGCTGAGCGCCGGTCGCTACCATTACCAGTTGGTC
TGGTGTCAAAAATAA
"""
from toolz import partition

# convert FASTA into single DNA sequence
dna = ''.join(line for line in gene.strip().split('\n')
              if not line.startswith('>'))

# partition DNA into codons (of length 3) and translate to amino acid
codons = (''.join(c) for c in partition(3, dna))
''.join(codon_table[codon] for codon in codons)
```

```
Out[85]: 'MTMITDSLAVVLQRRDWENPGVTQLNRLAAHPPFASWRNSEEARTDRPSQQLRSLNGEWRFAWFPAPEAVPESWLECDLPEADTVVVPSNWQM
```

The `partition` function can also be used for doing statistics on sequence windows, for example, in calculating a moving average.

0.14 Exercises

1. Rewrite the following nested loop as a list comprehension

```
ans = []
for i in range(3):
    for j in range(4):
        ans.append((i, j))
print ans
```

```
In [46]: # YOUR CODE HERE
```

2. Rewrite the following as a list comprehension

```
ans = map(lambda x: x*x, filter(lambda x: x%2 == 0, range(5)))
print ans
```

```
In [47]: # YOUR CODE HERE
```

3. Convert the function below into a pure function with no global variables or side effects

```
x = 5
def f(alist):
    for i in range(x):
        alist.append(i)
    return alist

alist = [1,2,3]
ans = f(alist)
print ans
print alist # alist has been changed!
```

```
In [48]: # YOUR CODE HERE
```

4. Write a decorator `hello` that makes every wrapped function print "Hello!"
For example

```
@hello
def square(x):
    return x*x
```

when called will give the following result

```
[In]
square(2)
[Out]
Hello!
4
```

```
In [49]: # YOUR CODE HERE
```

5. Rewrite the factorial function so that it does not use recursion.

```
def fact(n):
    """Returns the factorial of n."""
    # base case
    if n==0:
        return 1
    # recursive case
    else:
        return n * fact(n-1)
```

```
In [50]: # YOUR CODE HERE
```

Exercise 6. Rewrite the same factorail funciotn so that it uses a cache to speed up calculations

```
In [50]: # YOUR CODE HERE
```

7. Rewrite the following anonymous functiona as a regular named fucntion.

```
lambda x, y: x**2 + y**2
```

```
In [51]: # YOUR CODE HERE
```

8. Find an efficient way to extrac a subset of `dict1` into a a new dictionary `dict2` that only contains entrires with the keys given in the set `good_keys`. Note that good_keys may include keys not found in dict1 - these must be excluded when building dict2.

```
In [110]: import numpy as np
          import cPickle

          try:
              dict1 = cPickle.load(open('dict1.pic'))
          except:
              numbers = np.arange(1e6).astype('int') # 1 million entries
              dict1 = dict(zip(numbers, numbers))
              cPickle.dump(dict1, open('dict1.pic', 'w'), protocol=2)

          good_keys = set(np.random.randint(1, 1e7, 1000))

          # YOUR CODE HEREß
```

```
In []:
```