> *And now for something completely different.*
>
> *– John Cleese*

Functional Programming, List Comprehension, Iterators, Generators

Functional Programming

Functional Programming is a programming paradigm in which problems are decomposed into collections of functions (in the mathematical sense) and computation is expression evaluation. Functional Programming languages are examples of **declarative** languages in which the programmer writes a specification of the problem to be solved and the declarative language implementation determines how to solve the problem given the specification. In functional languages the specification is a collection of definitions of mathematical functions and the problem to be solved is expressed as a mathematical expression to be evaluated based on the given function definitions.

In **pure** functional programming there is no state (program variables) that can be modified. A consequence of this is that it is much easier to reason about the correctness of programs written in that style than in procedural languages where state changes. An example of a popular (pure) functional programming language is Haskell.

Python is not a functional programming language but it does borrow some ideas from functional languages such as anonymous functions, higher-order programming, list comprehension and lazy evaluation that, for example, provide powerful list processing techniques. We will look at examples shortly.

Iterators

We are familiar with the concept of for-loops by now. They take a collection of data and look at it one piece at a time from start to end. An object that can be iterated over is called an **iterable**. For example, strings, tuples, lists, and dictionaries are all iterables. But how do they work? How can we write our own iterables? How can we take advantage of iterables to do more powerful things?

All iterables can create a 'stream of data' that can be accessed one element at a time, called an **iterator**. Python uses the iterator to perform for-loops. Iterators are made using the `iter` function, and the "one element at a time" access is done using the `next` function. `next` will return the next piece of data. If there is no more data, then a `StopIteration` exception will be raised. Here is an example using a string. In this case, `x` is an iterable, and `it` is an iterator.

```
>>> s = 'spam'
>>> it = iter(s)
>>> next(it)
's'
>>> next(it)
'p'
>>> next(it)
'a'
```

```
>>> next(it)
'm'
```

```
>>> next(it)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    next(it)
StopIteration
```

Since iterators are only accessed one element at a time, there are a few advantages. If possible, iterators can be written to calculate the data 'on the fly' when requested, instead of calculating and storing all the data at once. This idea is called **lazy evaluation**. In fact, this approach can be used to generate an infinite stream of data. As long as we do not want to get all the elements (which would lead to infinite computation) this idea can lead to elegant solutions to problems that can be difficult to express with finite structures.

Where can lazy evaluation be useful? The `range` and `enumerate` functions are example of this. Recall that `range` and `enumerate` return a special class that is a sequence of numbers and objects. The reason for this is that these two functions are mainly used in just performing a for-loop over the data. Therefore, they are iterator types, making use of them for very large data sets avoids using up too much of the computer's resources.

Iterables and Iterators

```
iter(iterable)
iter(iterator)
next(iterator)
```

Semantics

An **iterable** is an object which can be iterated over (for example, used in for-loops). An **iterator** is an object which produces a stream or sequence of data one element at a time. `iter(iterable)` will create a new iterator. `iter(iterator)` will return the same iterator.

`next(iterator)` will return the next value in the sequence. If the sequence has finished, this function will raise `StopIteration`.

How do we write an iterable class, and the corresponding iterator? An iterable class must have an `__iter__` method to support the `iter` function. The method should return an iterator. The iterator object must have a `next` method which either returns the next value or raises `StopIteration`. The iterator must also have an `__iter__` method which returns the iterator itself; this is so that the iterator itself can also be iterated over. To demonstrate, here is an example (`geometric.py`) involving a geometric sequence, which is a sequence where each term is multiplied by a fixed ratio.

```
class GeometricSequence(object) :
    """A geometric sequence of numbers.

    The sequence of numbers:
    start, start * ratio, start * ratio**2, ..., start * ratio**(length-1)
    Without a length parameter, the sequence is infinite.
    """
```

```
    def __init__(self, start, ratio, length=None) :
        self._start = start
        self._ratio = ratio
        self._len = length

    def __iter__(self):
        return GeometricIterator(self._start, self._ratio, self._len)


class GeometricIterator(object) :
    """An iterator on a geometric sequence."""

    def __init__(self, start, ratio, length) :
        # Store values for later
        self._ratio = ratio
        self._len = length
        # Store information about position in the sequence
        self._pos = 0
        self._value = start

    def __iter__(self) :
        return self

    def next(self) :
        # Check if the sequence has finished
        if self._len is not None and self._pos >= self._len :
            raise StopIteration
        tmp = self._value
        # Update for next time.
        self._value *= self._ratio
        self._pos += 1
        return tmp

>>> powers_two = GeometricSequence(1, 2)
>>> it = iter(powers_two)
>>> next(it)
1
>>> next(it)
2
>>> next(it)
4
>>> next(it)
8
>>> for x in powers_two :
    print(x, end=" ")
    if x > 1000 :
        break

1 2 4 8 16 32 64 128 256 512 1024
>>> seq = GeometricSequence(2, 3, 6)
>>> for x in seq :
        print(x, end=" ")

2 6 18 54 162 486
>>> 54 in seq
True
```

```
>>> 20 in seq
False
>>> print(' '.join(GeometricSequence('*', 2, 4)))
* ** **** ********
```

Notice that the for-loop in the first example exits with a `break`. Since the sequence is infinite, there is no way to exit the loop other than specifying a condition we are interested in (which depends on the problem we are solving). The second sequence is defined with a length of 6, so after enough calls to `next`, a `StopIteration` is raised and the for loop exits naturally. The second sequence can also perform `in` tests. Be careful of performing `in` tests on infinite sequences like `powers_two`, because it will never stop looking through the sequence if the value is not there. The third example shows a geometric sequence of strings instead of numbers.

Generators

That last example was pretty big for code that generates a simple sequence, especially having to write two classes. As always, Python has found a simpler way of doing it.

Generators are iterators that use a syntax very similar to functions, using a `yield` statement instead of `return`. When a normal function in Python is called, the body is executed, and there is a return statement (possibly implicit) that stops the function and returns control back to the caller. When a generator function is called, the body of the function is not executed, instead we get a generator object. When `next` is called on the generator object, the function body begins executing and when the `yield` statement is reached the value supplied to `yield` is returned. The execution of the function is suspended at this point and the local state of the function is preserved. When `next` is again called the program resumes from the point of suspension and continues until the next `yield` statement is reached. In this way, the generator code is constantly starting and stopping, *generating* values through repetitive `yield` statements. Below is an example of a generator that illustrates this behaviour using the geometric sequence concept from before, which is also in `geometric2.py`.

```
def geometric(start, ratio, length=None) :
    pos = 0
    value = start
    while length is None or pos < length :
        yield value
        value *= ratio
        pos += 1

>>> powers_two = geometric(1, 2)
>>> next(powers_two)
1
>>> next(powers_two)
2
>>> next(powers_two)
4
>>> list(geometric(2, 3, 6))
[2, 6, 18, 54, 162, 486]
```

Here is another simple example. The `print` statements are added to show how the execution of the generator body works.

```
def gen_range(n) :
    print('start')
    for i in range(n) :
        print('before yield: i =', i)
        yield i
        print('after yield: i =', i)

>>> gen = gen_range(3)
>>> gen
<generator object gen_range at 0x011DC350>
>>> next(gen)
start
before yield: i = 0
0
>>> next(gen)
after yield: i = 0
before yield: i = 1
1
>>> next(gen)
after yield: i = 1
before yield: i = 2
2
>>> next(gen)
after yield: i = 2
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    next(gen)
StopIteration
```

Generator Syntax

The syntax for writing a generator is the same as for functions, except the body of the generator uses `yield` statements in the function body, and cannot return a value. The syntax for a yield statement is

```
yield value
```

Semantics

Calling the function does not execute the body, but returns an iterator. Calling `next` on that iterator will begin executing the function body, stopping at the first `yield` statement and returning the associated *`value`*. Subsequent uses of `next` resumes from the point where the function body was stopped. `StopIteration` is raised when the function body ends.

List Comprehensions

We already know how to construct lists out of other iterables, using loops and append. For example, if we have a file containing a collection of numbers, such as the `data1.txt` file, and we want to read in the numbers into a list, then we would do something like this:

```
f = open('data1.txt', 'r')
data = []
for line in f:
    data.append(float(line))
f.close()
```

For what is arguably a simple operation, it takes a few lines of code, and it might not be immediately obvious what it does. There is a way of doing the same thing with a better syntax, called a **list comprehension**:

```
f = open('data1.txt', 'r')
data = [float(line) for line in f]
f.close()
```

This syntax is much easier to type and read, and it is more efficient. What if we wanted to ignore certain lines in the dataset? For example, if there are blank lines in the file, we want to skip those lines and not attempt to add `float(line)` to the list. We can ignore the unwanted values by adding an `if` test to the comprehension, as shown below.

```
f = open('data1.txt', 'r')
data = [float(line) for line in f if line]
f.close()
```

Recall that `if line` is equivalent to `if line != ""`. Below are three more examples of list comprehension in action. In the first, we simply copy the list `l`. In the second we produce the list of squares of `l` and in the third we produce the list of squares of the even elements of `l`. The last example is more complex, it shows how comprehensions can be used to generate a list of prime numbers.

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> [i for i in l]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> [i*i for i in l]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> [i*i for i in l if i % 2 == 0]
[0, 4, 16, 36, 64]

>>> [i for i in range(2,50) if 0 not in [i % j for j in range(2,i)]]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

A list comprehension can handle more than one nested loop. The first two examples below are shown as both a loop and a comprehension.

```
>>> nums = [1, 2, 3]
>>> letters = "spam"
>>>
>>> pairs = []
>>> for i in letters :
        for j in nums :
            pairs.append((i, j))

>>> pairs
[('s', 1), ('s', 2), ('s', 3), ('p', 1), ('p', 2), ('p', 3), ('a', 1), ('a', 2),
('a', 3), ('m', 1), ('m', 2), ('m', 3)]

>>> pairs = [(i, j) for i in letters for j in nums]
>>> pairs
[('s', 1), ('s', 2), ('s', 3), ('p', 1), ('p', 2), ('p', 3), ('a', 1), ('a', 2),
('a', 3), ('m', 1), ('m', 2), ('m', 3)]

>>> sums = []
>>> for i in range(5) :
        if i % 2 == 0 :
            for j in range(4) :
                sums.append(i+j)

>>> sums
[0, 1, 2, 3, 2, 3, 4, 5, 4, 5, 6, 7]

>>> [i+j for i in range(5) if i % 2 == 0 for j in range(4)]
[0, 1, 2, 3, 2, 3, 4, 5, 4, 5, 6, 7]

>>> [i+j+k+l for i in '01' for j in '01' for k in '01' for l in '01']
['0000', '0001', '0010', '0011', '0100', '0101', '0110', '0111', '1000', '1001',
'1010', '1011', '1100', '1101', '1110', '1111']
```

List Comprehension Syntax

The syntax for list comprehension takes one of the following forms:

```
[expression for var in iterable]
[expression for var in iterable if test]
```

In general, there can be any number of "`for var in iterable`" forms used as shown below, and each one may have an optional "`if test`" after it.

```
[expression for var1 in iterable1 for var2 in iterable2 for var3 in iterable3 ...]
[expression for var1 in iterable1 if test1 for var2 in iterable2 if test2 ...]
```

Semantics

The comprehension is equivalent to constructing a list using for-loops and if statements.

If there is only one iterable (the first two forms above), set `var` to each element of `iterable` and evaluate `expression` and add it to the list. If there is an `if test`, then include only those expressions where the `test` is `True`.

If there are multiple iterables, then it is equivalent to nested for-loops, where `for var1 in iterable1` is the outermost loop.

We can also use a similar notation to that of list comprehension to create generators using **generator expressions**, simply by replacing the square brackets in the list comprehension by round brackets `()`. All of the examples of list comprehensions above can be turned into generator expressions by using round brackets. Often the result of a comprehension will be iterated over, and using a generator expression in this situation is more efficient. Below is an example.

```
>>> gen = (i*i for i in range(10))
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
4
>>> next(gen)
9
```

Aside: More Comprehensions

There are also two other types of comprehensions in Python. One is the dictionary comprehension, which can be used to make a dictionary, using `{}` braces instead of `[]` brackets, as well as a `key: value` expression.

```
>>> {i: i*i for i in range(10)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
>>> x = 'CSSE1001'
>>> {c: x.count(c) for c in x}
{'1': 2, '0': 2, 'C': 1, 'E': 1, 'S': 2}
>>> {i: x for i, x in enumerate(x)}
{0: 'C', 1: 'S', 2: 'S', 3: 'E', 4: '1', 5: '0', 6: '0', 7: '1'}
```

Another is the set comprehension. A **set** is a data type that represents an unordered collection of unique elements. They are efficient at checking whether or not an element is in a collection. Here are some examples of sets and set comprehensions:

```
>>> s = {2, 4, 6, 8, 6, 10}
>>> s
{8, 10, 4, 2, 6}
>>> 4 in s
True
>>> 5 in s
False
>>> {c for c in 'CSSE1001'}
{'1', '0', 'C', 'E', 'S'}
>>> {i*i for i in range(10)}
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
```

Higher-Order Functions

Now that we have seen iterators, list comprehension and generators we return to functional programming. Let's say that we have two very similar functions in our code. The first sums all the elements in a list, the second multiplies them. (There is already a built-in sum function, for the purposes of this section, we will ignore it and write our own.)

```
def my_sum(lst) :
    result = 0
    for x in lst :
        result += x
    return result

def product(lst) :
    result = 1
    for x in lst :
        result *= x
    return result
```

At the start of this course, we learnt to abstract similar code so that it does not need to be repeated, and turn the slight differences into parameters of the function, so we should be able to perform the same process here, and abstract these functions into a "combine" function which reduces a list into one value.

There are two extra arguments that the `combine` function needs: an initial value to set `result` to, and an operation to combine elements with. A way that we can do this is to use a *function* as a parameter which represents the operation to perform on `result` and `x`. Now we can write the abstracted function.

```
def combine(operation, lst, initial) :
    result = initial
    for x in lst :
        result = operation(result, x)
    return result
```

This function is different to ones we have seen before, since the `operation` parameter is actually another function. A function which uses other functions in the parameter or return values, such as `combine`, is called a **higher-order function**. We have seen this before, when creating tkInter Button widgets: `Button(frame, command=function)`, and using the `bind` method: `widget.bind("<Button-1>", function)`. Notice that what is really happening here is that the function is being treated as an object that can be used in the function call. Not all programming languages offer the ability to do this, but it is still very useful.

Anonymous Functions

So how do we use the `combine` function? If we want to sum a list of numbers, we can do this:

```
>>> lst = [2, 4, 1, 5, 3]
>>> def add(x, y) :
        return x + y
>>> combine(add, lst, 0)
15
```

It works! But, that seems like a lot of effort, having to define an addition function just so that the combine function works. Then for every other operation, we would need another trivial function definition just so that the combine function can use it. It would be helpful if there was a way of specifying simple functions without needing to write a `def` statement. Functions like this are called **anonymous functions**. The Python syntax to write an anonymous function is called a **lambda form**. The name lambda comes from the Lambda Calculus upon which functional languages are built. The equivalent of the `add` function above is just:

```
lambda x, y: x + y
```

Here it is used with `combine`. Some more examples of lambda expressions are shown below.

```
>>> combine(lambda x,y: x+y, lst, 0)
15

>>> double = lambda x: 2*x
>>> double
<function <lambda> at 0x011F2270>

>>> double(3)
6

>>> double(8)
16

>>> multiply = lambda x,y: x * y
>>> multiply(3, 4)
12

>>> multiply(2, 'abc')
'abcabc'

>>> zero = lambda: 0
>>> zero
<function <lambda> at 0x011F22F0>

>>> zero()
0
```

Lambda Expression Syntax

The syntax for a lambda form is:

```
lambda arg1, arg2, ...: expression
```

There may be zero or more *`args`*.

Semantics

The lambda form is a function which can be treated as an expression. This means it can be assigned to variables, used in function calls and return values, or used in larger expressions. It is often used as a way of defining simple functions instead of using the `def` statement. It is equivalent to the function definition below, except that the lambda expression does not have the name `f`.

```
def f(arg1, arg2, ...) :
    return expression
```

> Note that the only thing a lambda function can do is evaluate and return an expression. A function that requires loops or large blocks of code should be created using a `def` statement.

As a final thought, we can redefine the original `my_sum` and `product` functions using `combine`, and specifying what the operation and initial value are. The third `concat` function concatenates a list of strings into one string. As a challenge, try thinking of other operations that can be used with `combine`.

```
def my_sum(lst) :
    return combine(lambda x,y: x+y, lst, 0)

def product(lst) :
    return combine(lambda x,y: x*y, lst, 1)

def concat(strings) :
    return combine(lambda x,y: x+y, strings, '')
```

These examples can be downloaded, including the original definitions of `my_sum`, `product` and `combine`: `higher_order.py`

> **Aside: reduce and operator**
>
> The `combine` function is a very useful function, so it is no surprise that there is a Python library for functional programming (called `functools`), which has a function that does the same thing, called `functools.reduce`. In functional programming languages, this operation is known as a *fold*.
>
> The most common use of `reduce` is with operations like `+` and `*`, so it would be nice to have these as functions instead of having to write `lambda x,y: x+y`. The `operator` module provides many operators as functions. Try `import operator`.

Returning Functions

Another form of higher-order functions are those that return a function. Imagine a situation with lots of similar incrementing functions:

```
def add1(x) :
    return x + 1

def add5(x) :
    return x + 5

def add10(x) :
    return x + 10
```

We want to abstract these, but in particular we want to write a function that takes a number `n` (like 1, 5 or 10), and gives back a function that adds `n`. A function that adds `n` to its input `x` is simply:

```
lambda x: x + n
```

Then this lambda expression is what we must return. The "add n" function is shown below with examples.

```
def add_n(n) :
    return lambda x: x + n

>>> add_n(3)(4)
7

>>> add_n(1)(2)
3

>>> add1 = add_n(1)
>>> add5 = add_n(5)
>>> add10 = add_n(10)
```

Another way to think of this is as a **partial application**, because when we call the `add_n` function, we are not giving it all the information it needs to work out an answer. So instead, `add_n` gives back another function which waits for the remaining input.

Itertools

We have seen some of the potential of using iterators and generators. Python includes a module called `itertools` that contains a variety of useful iterators. Here we will briefly explore some of these. There are more details in the Python documentation.

```
>>> from itertools import count
```

The `count` function returns an iterator that counts numbers from a given starting number (or 0 if no number is given).

```
>>> c = count(2)
>>> next(c)
2

>>> next(c)
3

>>> next(c)
4

>>> next(c)
5
```

`map` applies a function to each value in an iterable. In the first example below, it uses a squaring function on the values of `count(1)` which has the effect of making an iterator of square numbers. `filter` applies a test to each value in the iterable, and only gives back the values that pass the test. In the second example below, only the numbers which satisfy `x%2 == 0` are allowed. The third example shows a combination of both a filter and a map.

```
>>> squares = map(lambda x:x*x, count(1))
>>> next(squares)
1

>>> next(squares)
4

>>> next(squares)
9

>>> evens = filter(lambda x: x%2 == 0, count(1))
```

```
>>> next(evens)
2
>>> next(evens)
4
>>> next(evens)
6
>>> even_squares = map(lambda x:x*x, filter(lambda x: x%2 == 0, count(1)))
>>> next(even_squares)
4
>>> next(even_squares)
16
>>> next(even_squares)
36
>>> x = "This is a short sentence"
>>> list(map(lambda x:(x, len(x)), x.split()))
[('This', 4), ('is', 2), ('a', 1), ('short', 5), ('sentence', 8)]
```

The `product`, `permutations`, `combinations` and `combinations_with_replacement` functions provide the Cartesian product and other combinatoric selections of elements.

```
>>> from itertools import product, permutations, combinations,
combinations_with_replacement
>>> list(product('ABC', '123'))
[('A', '1'), ('A', '2'), ('A', '3'), ('B', '1'), ('B', '2'), ('B', '3'), ('C',
'1'), ('C', '2'), ('C', '3')]
>>> [''.join(x) for x in product('01', repeat=3)]
['000', '001', '010', '011', '100', '101', '110', '111']
>>> list(permutations((1, 2, 3)))
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
>>> list(combinations((1, 2, 3), 2))
[(1, 2), (1, 3), (2, 3)]
>>> list(combinations_with_replacement((1, 2, 3), 2))
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

To finish this section we give two more advanced examples using this module.

Adding Line Numbers

For the first example we want to write a function that takes a text file as input and prints out the contents of the file with each line prepended with the line number. Here is the code (`add_linenum.py`).

```
import sys
import itertools

in_file = sys.argv[1]

f = open(in_file, 'r')

for line in map("{0:4d}: {1}".format, count(1), f):
    print(line, end="")

f.close()
```

This program is designed to be run from the command line as follows:

```
python add_linenum.py name_of_text_file
```

The `sys` module, among other things, provides a mechanism to access the arguments of the command. The attribute `argv` is the list of arguments. The 0'th argument is the name of the command itself (in this case `add_linenum.py`) and the first argument is the name of the file we want processed. The first step is to open the file. Now `f` is a generator and so we can use the iterator functions on it. We use `map` with three arguments: a function that takes two arguments and two iterators. The result will be an iterator that uses the function to pairwise combine the contents of the two input iterators. Note that, by using the infinite iterator `count` we do not have to go to the trouble of constructing some list that has the same length as the contents of the file.

Sieve of Eratosthenes

The final example is to implement the Sieve of Eratosthenes for computing the list of prime numbers using generators. This is a very complicated but dramatic example of the power of lazy evaluation (using generators)! Here is the code (`sieve.py`).

```
import itertools

def isnotdiv(p) :
    return lambda v: (v % p) != 0

def primes() :
    ints = itertools.count(2)
    while True :
        prime = next(ints)
        yield prime
        ints = filter(isnotdiv(prime), ints)


>>> prime = primes()
>>> next(prime)
2
>>> next(prime)
3
>>> next(prime)
5
```