Section1_2-Programming-with-Python

August 15, 2014

1 Programming with Python

1.1 Control Flow Statements

The control flow of a program determines the order in which lines of code are executed. All else being equal, Python code is executed linearly, in the order that lines appear in the program. However, all is not usually equal, and so the appropriate control flow is frequently specified with the help of control flow statements. These include loops, conditional statements and calls to functions. Let's look at a few of these here.

for statements

One way to repeatedly execute a block of statements (*i.e.* loop) is to use a `for` statement. These statements iterate over the number of elements in a specified sequence, according to the following syntax:

```
In [1]: for letter in 'ciao':
            print('give me a {0}'.format(letter.upper()))

give me a C
give me a I
give me a A
give me a O
```

Recall that strings are simply regarded as sequences of characters. Hence, the above `for` statement loops over each letter, converting each to upper case with the `upper()` method and printing it.

Similarly, as shown in the introduction, list comprehensions may be constructed using `for` statements:

```
In [2]: [i**2 for i in range(10)]

Out[2]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Here, the expression loops over `range(10)` – the sequence from 0 to 9 – and squares each before placing it in the returned list.

if statements

As the name implies, `if` statements execute particular sections of code depending on some tested condition. For example, to code an absolute value function, one might employ conditional statements:

```
In [3]: def absval(some_list):

            # Create empty list
            absolutes = []

            # Loop over elements in some_list
            for value in some_list:

                # Conditional statement
                if value<0:
                    # Negative value
```

```
                absolutes.append(-value)

            else:
                # Positive value
                absolutes.append(value)

        return absolutes
```

Here, each value in `some_list` is tested for the condition that it is negative, in which case it is multiplied by -1, otherwise it is appended as-is. For conditions that have more than two possible values, the elif statement can be used:

```
In [4]: x = 5
        if x < 0:
            print('x is negative')
        elif x % 2:
            print('x is positive and odd')
        else:
            print('x is even and non-negative')
```

```
x is positive and odd
```

while statements

A different type of conditional loop is provided by the `while` statement. Rather than iterating a specified number of times, according to a given sequence, `while` executes its block of code repeatedly, until its condition is no longer true. For example, suppose we want to sample from a truncated normal distribution, where we are only interested in positive-valued samples. The following function is one solution:

```
In [5]: # Import function
        from numpy.random import normal

        def truncated_normals(how_many, l):

            # Create empty list
            values = []

            # Loop until we have specified number of samples
            while (len(values) < how_many):

                # Sample from standard normal
                x = normal(0,1)

                # Append if not truncateed
                if x > l: values.append(x)

            return values
```

```
In [6]: truncated_normals(15, 0)
```

```
Out[6]: [1.1738372661791174,
         0.45099478603456566,
         1.6317748177330285,
         0.41275970336306655,
         1.4035699237984376,
         0.45342045186201724,
         0.795107166506508,
```

```
 1.803591568428664,
 0.14792870498175092,
 0.31721171897525846,
 0.9227994109843113,
 1.0430789423408247,
 2.2330729023818003,
 1.7023236403370337,
 0.16640359787626927]
```

This function iteratively samples from a standard normal distribution, and appends it to the output array if it is positive, stopping to return the array once the specified number of values have been added.

Obviously, the body of the `while` statement should contain code that eventually renders the condition false, otherwise the loop will never end! An exception to this is if the body of the statement contains a `break` or `return` statement; in either case, the loop will be interrupted.

1.2 Error Handling

Inevitably, some code you write will generate errors, at least in some situations. Unless we explicitly anticipate and **handle** these errors, they will cause your code to halt (sometimes this is a good thing!). Errors are handled using `try/except` blocks.

If code executed in the `try` block generates an error, code execution moves to the `except` block. If the exception that is specified corresponsd to that which has been raised, the code in the `except` block is executed before continuing; otherwise, the exception is carried out and the code is halted.

```
In [7]: absval(-5)
```

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

<ipython-input-7-241bbde36660> in <module>()
----> 1 absval(-5)


<ipython-input-3-e94eca0d977f> in absval(some_list)
  5
  6     # Loop over elements in some_list
----> 7     for value in some_list:
  8
  9         # Conditional statement


TypeError: 'int' object is not iterable
```

In the call to `absval`, we passed a single negative integer, whereas the function expects some sort of iterable data structure. Other than changing the function itself, we can avoid this error using exception handling.

```
In [8]: x = -5
        try:
            print(absval(x))
        except TypeError:
            print('The argument to absval must be iterable!')
```

```
The argument to absval must be iterable!
```

```
In [9]: x = -5
        try:
            print(absval(x))
        except TypeError:
            print(absval([x]))
```

```
[5]
```

We can raise exceptions manually by using the `raise` expression.

```
In [10]: raise ValueError('This is the wrong value')
```

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

        <ipython-input-10-7b8b3dfef9b4> in <module>()
    ----> 1 raise ValueError('This is the wrong value')


        ValueError: This is the wrong value
```

```
In [11]: try:
             raise ValueError('This is the wrong value')
         except ValueError:
             pass
```

1.3 Importing and Manipulating Data

Python includes operations for importing and exporting data from files and binary objects, and third-party packages exist for database connectivity. The easiest way to import data from a file is to parse delimited text file, which can usually be exported from spreadsheets and databases. In fact, file is a built-in type in python. Data may be read from and written to regular files by specifying them as file objects:

```
In [12]: microbiome = open('../data/microbiome.csv')
```

Here, a file containing microbiome data in a comma-delimited format is opened, and assigned to an object, called `microbiome`. The next step is to transfer the information in the file to a usable data structure in Python. Since this dataset contains four variables, the name of the taxon, the patient identifier (de-identified), the bacteria count in tissue and the bacteria count in stool, it is convenient to use a dictionary. This allows each variable to be specified by name.

First, a dictionary object is initialized, with appropriate keys and corresponding lists, initially empty. Since the file has a header, we can use it to generate an empty dict:

```
In [13]: column_names = microbiome.next().rstrip('\n').split(',')
```

```
In [14]: mb_dict = {name:[] for name in column_names}
```

It is then a matter of looping over each line of the data. Python file objects are iterable, essentially just a sequence of lines, and fit naturally into a for statement.

```
In [15]: for line in microbiome:
             taxon,patient,tissue,stool = line.rstrip('\n').split(',')
             mb_dict['Taxon'].append(taxon)
             mb_dict['Patient'].append(int(patient))
             mb_dict['Tissue'].append(int(tissue))
             mb_dict['Stool'].append(int(stool))
```

For each line in the file, data elements are split by the comma delimiter, using the `split` method that is built-in to string objects. Each datum is subsequently appended to the appropriate list stored in the dictionary. After all the data is parsed, it is polite to close the file:

```
In [16]: microbiome.close()
```

The data can now be readily accessed by indexing the appropriate variable by name:

```
In [17]: mb_dict['Tissue'][:10]
```

```
Out[17]: [632, 136, 1174, 408, 831, 693, 718, 173, 228, 162]
```

A second approach to importing data involves interfacing directly with a relational database management system. Relational databases are far more efficient for storing, maintaining and querying data than plain text files or spreadsheets, particularly for large datasets or multiple tables. A number of third parties have created packages for database access in Python. For example, `sqlite3` is a package that provides connectivity for SQLite databases:

```
In [18]: import sqlite3                # import database package, and connect
         db = sqlite3.connect(database='../data/baseball-archive-2011.sqlite')
         cur = db.cursor()        # create a cursor object to mediate
                                          # communication with database
```

```
In [19]: # run query
         cur.execute('SELECT playerid, HR, SB FROM Batting WHERE yearID=1970')
         baseball = cur.fetchall()        # fetch data, and assign to variable
         baseball[:10]
```

```
Out[19]: [(u'aaronha01', 38, 9),
          (u'aaronto01', 2, 0),
          (u'abernte02', 0, 0),
          (u'abernte02', 0, 0),
          (u'abernte02', 0, 0),
          (u'acosted01', 0, 0),
          (u'adairje01', 0, 0),
          (u'ageeto01', 24, 31),
          (u'aguirha01', 0, 0),
          (u'akerja01', 0, 0)]
```

1.4 Functions

Python uses the `def` statement to encapsulate code into a callable function. Here again is a very simple Python function:

```
In [20]: # Function for calulating the mean of some data
         def mean(data):

             # Initialize sum to zero
             sum_x = 0.0

             # Loop over data
             for x in data:

                 # Add to sum
                 sum_x += x

             # Divide by number of elements in list, and return
             return sum_x / len(data)
```

As we can see, arguments are specified in parentheses following the function name. If there are sensible "default" values, they can be specified as a *keyword argument*.

```
In [21]: def var(data, sample=True):

             # Get mean of data from function above
             x_bar = mean(data)

             # Do sum of squares in one line
             sum_squares = sum([(x - x_bar)**2 for x in data])

             # Divide by n-1 and return
             if sample:
                 return sum_squares/(len(data)-1)
             return sum_squares/len(data)
```

Non-keyword arguments must always predede keyword arguments, and must always be presented in order; order is not important for keyword arguments.

Arguments can also be passed to functions as a `tuple`/`list`/`dict` using the asterisk notation.

```
In [22]: def some_computation(a=-1, b=4.3, c=7):
             return (a + b) / float(c)

         args = (5, 4, 3)
         some_computation(*args)
```

```
Out[22]: 3.0
```

```
In [23]: kwargs = {'b':4, 'a':5, 'c':3}
         some_computation(**kwargs)
```

```
Out[23]: 3.0
```

The `lambda` statement creates anonymous one-line functions that can simply be assigned to a name.

```
In [24]: import numpy as np
         normalize = lambda data: (np.array(data) - np.mean(data)) / np.std(data)
```

or not:

```
In [25]: (lambda data: (np.array(data) - np.mean(data)) / np.std(data))([5,8,3,8,3,1,2,1])
```

```
Out[25]: array([ 0.42192651,  1.54706386, -0.32816506,  1.54706386, -0.32816506,
                -1.07825663, -0.70321085, -1.07825663])
```

Python has several built-in, higher-order functions that are useful.

```
In [26]: filter(lambda x: x > 5, range(10))
```

```
Out[26]: [6, 7, 8, 9]
```

```
In [27]: abs([5,-6])
```

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

<ipython-input-27-2fc5ae37d9b8> in <module>()
```

```
----> 1 abs([5,-6])


    TypeError: bad operand type for abs(): 'list'
```

```
In [28]: map(abs, [5, -6])
```

```
Out[28]: [5, 6]
```

1.5 Example: Least Squares Estimation

Lets try coding a statistical function. Suppose we want to estimate the parameters of a simple linear regression model. The objective of regression analysis is to specify an equation that will predict some response variable Y based on a set of predictor variables X. This is done by fitting parameter values β of a regression model using extant data for X and Y. This equation has the form:

$$Y = X\beta + \epsilon$$

where ϵ is a vector of errors. One way to fit this model is using the method of *least squares*, which is given by:

$$\hat{\beta} = (X\prime X)^{-1} X\prime Y$$

We can write a function that calculates this estimate, with the help of some functions from other modules:

```
In [29]: from numpy.linalg import inv
         from numpy import transpose, array, dot
```

We will call this function `solve`, requiring the predictor and response variables as arguments. For simplicity, we will restrict the function to univariate regression, whereby only a single slope and intercept are estimated:

```
In [30]: def solve(x,y):
             'Estimates regession coefficents from data'

             '''
             The first step is to specify the design matrix. For this,
             we need to create a vector of ones (corresponding to the intercept term,
             and along with x, create a n x 2 array:
             '''
             X = array([[1]*len(x), x])

             '''
             An array is a data structure from the numpy package, similar to a list,
             but allowing for multiple dimensions. Next, we calculate the transpose of x,
             using another numpy function, transpose
             '''
             Xt = transpose(X)

             '''
             Finally, we use the matrix multiplication function dot, also from numpy
             to calculate the dot product. The inverse function is provided by the LinearAlgebra
             package. Provided that x is not singular (which would raise an exception), this
             yields estimates of the intercept and slope, as an array
             '''
```

```
            b_hat = dot(inv(dot(X,Xt)), dot(X,y))

            return b_hat
```

Here is solve in action:

```
In [31]: solve((10,5,10,11,14),(-4,3,0,23,0.6))

Out[31]: array([ 2.04380952,  0.24761905])
```

1.6 Object-oriented Programming

As previously stated, Python is an object-oriented programming (OOP) language, in contrast to procedural languages like FORTRAN and C. As the name implies, object-oriented languages employ objects to create convenient abstractions of data structures. This allows for more flexible programs, fewer lines of code, and a more natural programming paradigm in general. An object is simply a modular unit of data and associated functions, related to the state and behavior, respectively, of some abstract entity. Object-oriented languages group similar objects into classes. For example, consider a Python class representing a bird:

```
In [32]: class bird:
             # Class representing a bird

             name = 'bird'

             def __init__(self, sex):
                 # Initialization method

                 self.sex = sex

             def fly(self):
                 # Makes bird fly

                 print('Flying!')

             def nest(self):
                 # Makes bird build nest

                 print('Building nest ...')

             @classmethod
             def get_name(cls):
                 # Class methods are shared among instances

                 return cls.name
```

You will notice that this `bird` class is simply a container for two functions (called *methods* in Python), `fly` and `nest`, as well as one attribute, `name`. The methods represent functions in common with all members of this class. You can run this code in Python, and create birds:

```
In [33]: Tweety = bird('male')
         Tweety.name

Out[33]: 'bird'

In [34]: Tweety.fly()
```

```
Flying!
```

```
In [35]: Foghorn = bird('male')
         Foghorn.nest()
```

```
Building nest ...
```

A `classmethod` can be called without instantiating an object.

```
In [36]: bird.get_name()
```

```
Out[36]: 'bird'
```

As many instances of the `bird` class can be generated as desired, though it may quickly become boring. One of the important benefits of using object-oriented classes is code re-use. For example, we may want more specific kinds of birds, with unique functionality:

```
In [37]: class duck(bird):
             # Duck is a subclass of bird

             name = 'duck'

             def swim(self):
                 # Ducks can swim

                 print('Swimming!')

             def quack(self,n):
                 # Ducks can quack

                 print('Quack! ' * n)
```

Notice that this new `duck` class refers to the `bird` class in parentheses after the class declaration; this is called **inheritance**. The subclass `duck` automatically inherits all of the variables and methods of the superclass, but allows new functions or variables to be added. In addition to flying and best-building, our duck can also swim and quack:

```
In [38]: Daffy = duck('male')
         Daffy.swim()
```

```
Swimming!
```

```
In [39]: Daffy.quack(3)
```

```
Quack! Quack! Quack!
```

```
In [40]: Daffy.nest()
```

```
Building nest ...
```

Along with adding new variables and methods, a subclass can also override existing variables and methods of the superclass. For example, one might define `fly` in the `duck` subclass to return an entirely different string. It is easy to see how inheritance promotes code re-use, sometimes dramatically reducing development time. Classes which are very similar need not be coded repetitiously, but rather, just extended from a single superclass.

This brief introduction to object-oriented programming is intended only to introduce new users of Python to this programming paradigm. There are many more salient object-oriented topics, including interfaces, composition, and introspection. I encourage interested readers to refer to any number of current Python and OOP books for a more comprehensive treatment.

1.7 In Python, everything is an object

Everything (and I mean *everything*) in Python is an object, in the sense that they possess attributes, such as methods and variables, that we usually associate with more "structured" objects like those we created above.

Check it out:

```
In [41]: dir(1)
```

```
Out[41]: ['__abs__',
          '__add__',
          '__and__',
          '__class__',
          '__cmp__',
          '__coerce__',
          '__delattr__',
          '__div__',
          '__divmod__',
          '__doc__',
          '__float__',
          '__floordiv__',
          '__format__',
          '__getattribute__',
          '__getnewargs__',
          '__hash__',
          '__hex__',
          '__index__',
          '__init__',
          '__int__',
          '__invert__',
          '__long__',
          '__lshift__',
          '__mod__',
          '__mul__',
          '__neg__',
          '__new__',
          '__nonzero__',
          '__oct__',
          '__or__',
          '__pos__',
          '__pow__',
          '__radd__',
          '__rand__',
          '__rdiv__',
          '__rdivmod__',
          '__reduce__',
          '__reduce_ex__',
          '__repr__',
          '__rfloordiv__',
          '__rlshift__',
          '__rmod__',
          '__rmul__',
          '__ror__',
          '__rpow__',
          '__rrshift__',
```

```
 '__rshift__',
 '__rsub__',
 '__rtruediv__',
 '__rxor__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__sub__',
 '__subclasshook__',
 '__truediv__',
 '__trunc__',
 '__xor__',
 'bit_length',
 'conjugate',
 'denominator',
 'imag',
 'numerator',
 'real']
```

```
In [42]: (1).bit_length()
```

```
Out[42]: 1
```

This has implications for how assignment works in Python.
Let's create a trivial class:

```
In [43]: class Thing: pass
```

and instantiate it:

```
In [44]: x = Thing()
         x
```

```
Out[44]: <__main__.Thing instance at 0x10b88c1b8>
```

Here, `x` is simply a "label" for the object that we created when calling `Thing`. That object resides at the memory location that is identified when we print `x`. Notice that if we create another `Thing`, we create an new object, and give it a label. We know it is a new object because it has its own memory location.

```
In [45]: y = Thing()
         y
```

```
Out[45]: <__main__.Thing instance at 0x10b88c0e0>
```

What if we assign `x` to `z`?

```
In [46]: z = x
         z
```

```
Out[46]: <__main__.Thing instance at 0x10b88c1b8>
```

We see that the object labeled with `z` is the same as the object as that labeled with `x`. So, we say that `z` is a label (or name) with a *binding* to the object created by `Thing`.

So, there are no "variables", in the sense of a container for values, in Python. There are only labels and bindings.

```
In [47]: x.name = 'thing x'
```

```
In [48]: z.name
```

```
Out[48]: 'thing x'
```

This can get you into trouble. Consider the following (seemingly inoccuous) way of creating a dictionary of emtpy lists:

```
In [49]: evil_dict = dict.fromkeys(column_names, [])
         evil_dict
```

```
Out[49]: {'Patient': [], 'Stool': [], 'Taxon': [], 'Tissue': []}
```

```
In [50]: evil_dict['Tissue'].append(5)
```

```
In [51]: evil_dict
```

```
Out[51]: {'Patient': [5], 'Stool': [5], 'Taxon': [5], 'Tissue': [5]}
```

Why did this happen?

1.8 Generators

When a Python functions is called, it creates a namespace for the function, executes the code that comprises the function (creating objects inside the namespace as required), and returns some result to its caller. After the return, everything inside the namespace (including the namespace itself) is gone, and is created anew when the function is called again.

However, one particular class of functions in Python breaks this pattern, returning a value to the caller while still active, and able to return subsequent values as needed. Python ***generators*** employ `yield` statements in place of `return`, allowing a sequence of values to be generated without having to create a new function namespace each time. In other languages, this construct is known as a *coroutine*.

For example, we may want to have a function that returns a sequence of values; let's consider, for a simple illustration, the Fibonacci sequence:

$$F_i = F_{i-2} + F_{i-1}$$

its certaintly possible to write a standard Python function that returns however many Fibonacci numbers that we need:

```
In [52]: def fibonacci(size):
             F = np.empty(size, 'int')
             a, b = 0, 1
             for i in xrange(size):
                 F[i] = a
                 a, b = b, a + b
             return F
```

and this works just fine:

```
In [53]: fibonacci(20)
```

```
Out[53]: array([   0,    1,    1,    2,    3,    5,    8,   13,   21,   34,   55,
                  89,  144,  233,  377,  610,  987, 1597, 2584, 4181])
```

However, what if we need one number at a time, or if we need a million or 10 million values? In the first case, you would somehow have to store the values from the last iteration, and restore the state to the function each time it is called. In the second case, you would have to generate and then store a very large number of values, most of which you may not need right now.

A more sensible solution is to create a `generator`, which calculates a single value in the sequence, then *returns control back to the caller*. This allows the generator to be called again, resuming the sequence generation where it left off. Here's the Fibonacci function, implemented as a generator:

```
In [54]: def gfibonacci(size):
             a, b = 0, 1
             for _ in xrange(size):
                 yield a
                 a, b = b, a + b
```

Notice that there is no `return` statement at all; just `yield`, which is where a value is returned each time one is requested. The `yield` statement is what defines a generator.

When we call our generator, rather than a sequence of Fibonacci numbers, we get a generator object:

```
In [55]: f = gfibonacci(100)
         f

Out[55]: <generator object gfibonacci at 0x10b88e640>
```

A generator has a `__next__()` method that can be called either via the method `generator.next()` or the builtin function `next()`. The call to `next` executes the generator until the `yield` statement is reached, returning the next generated value, and then pausing until another call to `next` occurs:

```
In [56]: f.next(), f.next(), next(f)

Out[56]: (0, 1, 1)
```

A generator is a type of `iterator`. If we call a function that supports iterables using a generator as an argument, it will know how to use the generator.

```
In [57]: np.array(list(f))

Out[57]: array([2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584,
                4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,
                317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887, 9227465,
                14930352, 24157817, 39088169, 63245986, 102334155, 165580141,
                267914296, 433494437, 701408733, 1134903170, 1836311903, 2971215073,
                4807526976, 7778742049, 12586269025, 20365011074, 32951280099,
                53316291173, 86267571272, 139583862445, 225851433717, 365435296162,
                591286729879, 956722026041, 1548008755920, 2504730781961,
                4052739537881, 6557470319842, 10610209857723, 17167680177565,
                27777890035288, 44945570212853, 72723460248141, 117669030460994,
                190392490709135, 308061521170129, 498454011879264, 806515533049393,
                1304969544928657, 2111485077978050, 3416454622906707,
                5527939700884757, 8944394323791464, 14472334024676221,
                23416728348467685, 37889062373143906, 61305790721611591,
                99194853094755497, 160500643816367088, 259695496911122585,
                420196140727489673, 679891637638612258, 1100087778366101931,
                1779979416004714189, 2880067194370816120, 4660046610375530309,
                7540113804746346429, 12200160415121876738L, 19740274219868223167L,
                31940434634990099905L, 51680708854858323072L, 83621143489848422977L,
                135301852344706746049L, 218922995834555169026L], dtype=object)
```

What happens when we reach the "end" of a generator?

```
In [58]: a_few_fibs = gfibonacci(2)

In [59]: a_few_fibs.next()

Out[59]: 0
```

```
In [60]: a_few_fibs.next()

Out[60]: 1

In [61]: a_few_fibs.next()


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

        <ipython-input-61-0eaea96ec098> in <module>()
    ----> 1 a_few_fibs.next()


        StopIteration:
```

Thus, generators signal when there are no further values to generate by throwing a `StopIteration` exception. We must either handle this exception, or create a generator that is infinite, which we can do in this example by replacing a `for` loop with a `while` loop:

```
In [62]: def infinite_fib():
             a, b = 0, 1
             while True:
                 yield a
                 a, b = b, a + b

In [63]: f = infinite_fib()
         vals = [f.next() for _ in range(10000)]
         vals[-1]

Out[63]: 207936082371334980721126489886428368250870360940159031196829458665285014234556866489274560343305
```

1.9 Exercise: Translate R to Python

Recode the secant search function from Bios 301 from R to Python.

```
In [64]: %load http://git.io/-2DM8Q

In [65]: # Write your code here

In [66]: # Run this cell to load the answer
         %load http://git.io/h5jqBg
```

How about a modified secant functiont that uses a generator?

```
In [67]: # Write your code here

In [68]: # Run this cell to load the answer
         %load http://git.io/-CSooQ
```

1.10 References

- Learn Python the Hard Way
- Learn X in Y Minutes (where X=Python)

- 29 common beginner Python errors on one page
- Understanding Python's Execution Model

```
In [69]: from IPython.core.display import HTML
         def css_styling():
             styles = open("styles/custom.css", "r").read()
             return HTML(styles)
         css_styling()
```

```
Out[69]: <IPython.core.display.HTML at 0x10b895290>
```