Business Decision Analytics under Uncertainty

Rutgers Business School, Undergraduate New Brunswick
Python/PyCharm Background Notes, Fall 2017

This document will cover some basics of Python and PyCharm. PyCharm is a very capable interactive programming environment for the Python language.

Python is a powerful, general-purpose programming language that is quite easy to use and has numerous open-source modules. These modules can provide functions such as parsing text, scientific calculations, or producing attractive charts that can be written directly to PDF files. Generally speaking, Python programs will run more slowly than those in languages that are compiled directly to processor chip instructions, such as C or C++. But with a little care Python will be more than fast enough for our purposes.

We will not explore the language in great detail, but cover enough of its basic capabilities so that we can understand and write code to perform dynamic programming calculations.

Setting up PyCharm

PyCharm is designed for developers of large, complicated software packages. In some sense it is "overkill" for our purposes, since we will generally just run simple codes that are up to a few pages long. But we will still benefit from PyCharm's debugging environment, which allows you to easily visualize how a program is running. The PyCharm editor also has some nice features to help you avoid making simple errors.

First, let us get PyCharm set up. Rather than individual files, PyCharm is oriented to working on a *project*, which is a directory containing multiple Python code files. This project orientation is a little annoying for our purposes, but once we have a project set up, we should be able to largely forget about it.

1. First, run PyCharm (for example, from the start menu in Windows)

2. Select "New Project", the first option under the "File" menu (the first menu on the left). A dialog box will appear.

3. Here, we'll create a new, empty project. Next to "location" in the dialog box, select the box with "…". A sub-dialog box will appear.

4. Navigate to the directory in which you'd like your project to reside.

5. Click the "new folder" icon (which looks like a folder with an asterisk next to it), and then enter a name for the project (such as "BDAuU-code"), and click "OK".

6. Click "OK" and the "Create", and "OK" one more time.

7. You now have a blank, empty project. To get started, click the "Python Console" icon at the bottom left of the window. If you have an older version of PyCharm and there is no such icon, go to the "View" menu, choose the "Tool Windows" item, and then select "Python Console".

Basic Interactive Computing

In the Python console, you can type commands to interact with the Python interpreter. The console interpreter accepts Python expressions and prints their values, or executes Python commands one at a time. To indicate its readiness, it types ">>>".

For example, we could say

```
>>> 2 + 2
```

And the interpreter will respond with

```
4
```

Or we could write

```
>>> print("Hello")
```

And Python will print out

```
Hello
```

The `print( )` function simply takes its arguments and writes them to the screen. For compatibility Python version 3.*x*, our print statements must have parentheses. In older versions of Python, you can use a space after `print` instead of the parentheses.

We can also define and assign variables in the interpreter. For example,

```
>>> x = 6
```

implicitly defines a variable called `x` and sets it to 6. If we subsequently write

```
>>> print(x)
```

Then Python will respond with

```
6
```

If we subsequently say

```
>>> x = 12
>>> print(x)
```

Then Python will change the value of the variable and the print it out again, resulting in

```
12
```

Datatypes and Lists

A *datatype* is a category of information. The basic datatypes in Python are similar to those in most modern programming languages:

- Integer (whole number) values, like 0, 27, 432, or –23
- Floating-point numbers like 1.0, 3.02, –342.811, 1.233e12, or 3.41e-6
- Character strings like "Fred" or "hello there @ 23"
- Boolean (logical) values that may be either `True` or `False`

Unlike "typed" languages such as C, C++, and Java, Python implicitly establishes the datatype of a variable when creating it, and you can change the datatype of a variable over time. For example, the initial statement `x = 6` implicitly established `x` as a variable holding an integer value, but if we were to write

```
>>> x = "howdy doo"
```

then `x` would immediately change to being a character string. This flexibility can sometimes be convenient, but it can also slow down the execution of the language because the interpreter must constantly check which datatypes it is dealing with.

Another fundamental datatype in Python is the *list*. A list is a sequence of values enclosed in [ ] brackets and separated by commas. For example,

```
>>> L = [2,3,"fork",-7,8]
```

Defines `L` to be a list consisting of the elements 2, 3, "fork", –7, and 8. Usually, all the items in a list have the same datatype, but this is not required and is in fact not true in the case above. You can refer to individual elements of a list by a different use of the symbols [ ]: in general, `L[`*i*`]` is the i^{th} element of the list `L`. Technically, this reference method is called *indexing*. Similarly to Java, C, and C++, the Python language uses "zero-based" indexing, meaning that the first item in a list is numbered 0, the second one is numbered 1, and so forth. For example

```
>>> print(L[0])
```

now causes Python to print

```
2
```

While

```
>>> print(L[2])
```

causes Python to print out the third element of L, which produces

```
fork
```

This zero-based number scheme is sometimes irritating, but is standard in most modern programming languages.

The `len( )` function returns the length of a list. For example, in our case, writing

```
>>> len(L)
```

Computes the value

```
5
```

Note that the last element in a list `L` is at position `len(L) - 1`: for example

```
>>> L[len(L)-1]
```

Produces

```
8
```

Simple Programs: `for` *Loops,* `if` *Statements, and Ranges*

Next, we will write a simple program, rather than just executing commands on the console.

1. In PyCharm, make sure the name of your project is selected at the top left pane of the screen.

2. Go the file menu and choose "New" (if this option is grayed out, click in the large empty area in the middle of the screen to move the user interface focus away from the Python console).

3. Select "Python File" and then type a file name ending in ".py" in the resulting dialog box – for example, "test.py". The extension ".py" is the standard file type Python programs.

4. In the editor pane in the upper left corner of the screen, enter some code like the following:

```
data = [43,3,3,27,-2,34,100,302,-17]
print("Length is "+ str(len(data)))
```

Right-click in the window and select the option “run ‘test’” (if your program is named “test.py” – if you had chosen the name “fred.py”, the run option will say “run ‘fred’”). You should see output like

```
Length is 9

Process finished with exit code 0
```

The final line of output (about the process finishing) just indicates that the program finished running normally.

To explain the second statement in this program, the `str( )` function converts `len(data)` from an integer to a character string. For strings, Python interprets the “+” operation as concatenation (laying end-to-end). The sequence is thus that

1. `len(data)` produces the integer 9
2. `str( )` converts this integer into the string “9”
3. The “+” operator forms the string resulting from concatenating “Length is ” and “9”, that is, “Length is 9”
4. The `print` function displays this string.

We now introduce the notion of a `for` loop. Its simplest form is

`for` *loopVariable* `in` *list* `:`
Indented block of code

The loop works as follows: for each element of *list*, Python assigns its value to *loopVariable* and then executes the indented block of code immediately following the `for` statement. The block of code will get executed as many times as there are elements in the list. Consider the following example:

```
data = [43,3,3,27,-2,34,100,302,-17]

print("Length is "+ str(len(data)))

for item in data :
    print("There is an item with value " + str(item))
    print("Done with this item")

print(“Done with everything”)
```

This produces the output

```
Length is 9
There is an item with value 43
Done with this item
There is an item with value 3
Done with this item
There is an item with value 3
```

```
Done with this item
There is an item with value 27
Done with this item
There is an item with value -2
Done with this item
There is an item with value 34
Done with this item
There is an item with value 100
Done with this item
There is an item with value 302
Done with this item
There is an item with value -17
Done with this item
Done with everything
```

There are several key things to notice here:

1. The `for` statement ends with a colon. Especially if you are used other programming languages, this punctuation is very easy to forget, and you will get an "invalid syntax" error if you do so.

2. Block structure in Python is controlled by indentation. The "some code" lines that are executed repeatedly within the scope of the loop are indicated by being indented more than the `for` statement. As soon Python encounters a statement that isn't indented more than the `for`, like final statement `print("Done with everything")` in this case, the block of code in the loop is closed (and that statement is not part of the loop). Languages like C, C++, and Java instead indicate block structure with the characters `{` and `}`.

3. There is no semicolon at the end of each statement as in C, C++, or Java. If you want to continue a statement across multiple lines, you can use a `\` character at the end of each line that needs to be continued. Often however, it is already clear from context that a line isn't complete – for example if it has a `(` but no matching `)` character.

One can actually loop over more complicated data objects than lists. Instead of a list, Python's `for` statement allows the object being looped over to be anything "iterable", that is, any data structure that can contain multiple subsidiary data objects. A very common choice here is a special object called a "range". The simplest form of this object is created by the expression `range(`*n*`)`. This object is equivalent to a list containing the numbers 0 through $n - 1$ in sequence. For example, `range(4)` is essentially equivalent to `[0, 1, 2, 3]`.

Let's change our simple code to the following:

```
data = [43,3,3,27,-2,34,100,302,-17]

n = len(data)

for i in range(n) :
    print("item " + str(i) + " is " + str(data[i]))

print("Done")
```

Running this code produces

```
item 0 is 43
item 1 is 3
item 2 is 3
item 3 is 27
item 4 is -2
item 5 is 34
item 6 is 100
item 7 is 302
item 8 is -17
Done
```

Here, `n` gets set to the length of the list `data`, which is 9, and the statement

```
for i in range(n) :
```

loops through the values `i` = 0, `i` = 1, through `i` = 8. This gives us another way of looping through the elements of `data`, with the variable `i` now telling us which position we are currently inspecting.

The `range` function can be used to generate other kinds of simple arithmetic sequences. Its full form is `range(`*start*`,` *stop*`,` *step*`)`. The idea is the function creates an object that is equivalent to list obtained by starting to count from *start* by increments of *step*, until it reaches *stop*. The *stop* point is always "exclusive", meaning that the first element equal to or beyond the stop point is not part of the resulting list. For example, to count from 1 to *n* instead of 0 to *n* – 1, we use `range(1,` *n* + 1`, 1)`. For instance, we have

```
>>> list(range(1,11,1))

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Here, the `list` function converts the range to an explicitly visible list; without this, we just get a compact range object that behaves in a similar way but uses less memory and doesn't print very informatively:

```
>>> range(1,11,1)

range(1, 11)
```

Note that the *step* argument defaults to 1, so that `range(1,11,1)` is the same as `range(1,11)`.

By making *step* negative, we can count backwards. For example, `range(10,0,-1)` counts from 10 backwards to 1:

```
>>> list(range(10,0,-1))

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Remember, the *stop* point is exclusive and thus not included in the output, so the result of `range(10,0,-1)` stops after 1 and does not include zero.

The other principal programming construct we will use is the `if` statement. The simplest form of the `if` statement is

`if` *logicalExpression* `:`
    Indented block of code

The indented block of code is only executed if *logicalExpression* evaluated to `True`, and otherwise it is skipped. For example, suppose we write

```
data = [43,3,3,27,-2,34,100,302,-17]

n = len(data)

for i in range(n) :
    print("item" + str(i) + " is " + str(data[i]))
    if data[i] > 30 :
        print("Big")

print("Done")
```

Then we get the output

```
item 0 is 43
Big
item 1 is 3
item 2 is 3
item 3 is 27
item 4 is -2
item 5 is 34
Big
item 6 is 100
Big
item 7 is 302
Big
item 8 is -17
Done
```

Remember that `if` statements must end with a colon, just like `for` statements. An `if` statement can be optionally followed by any number of `elif` statement (an abbreviation of "else if") and finally by an `else` statement. For example, consider

```
data = [43,3,3,27,-2,34,100,302,-17]

n = len(data)

for i in range(n) :
    print("item" + str(i) + " is " + str(data[i]))
    if data[i] > 30 :
        print("Big")
    elif data[i] < 10 :
        print("Small")
    else :
        print("Meh")

print("Done")
```

which produces the output

```
item 0 is 43
Big
item 1 is 3
Small
item 2 is 3
Small
item 3 is 27
Meh
item 4 is -2
Small
item 5 is 34
Big
item 6 is 100
Big
item 7 is 302
Big
item 8 is -17
Small
Done
```

Just was with `if`, be sure that your `elif` and `else` statements end with a colon. The block after an `elif` executes if all the immediately preceding conditions have evaluated to `False` and its condition evaluates to `True`. The block following an `else` executes if all the immediately preceding conditions have evaluated to `False`.

Remembering the Largest or Smallest of a Sequence of Calculations

One standard programming pattern that we are going to be using repeatedly is to remember which of a sequence of calculations resulted in the largest or smallest value. One simple possibility is to use Python's `max` or `min` function. For example, we may write

```
>>> L = [2, 3, 8, -7, 2]
>>> max(L)
8
```

However, the result of the `max` does not tell us that the third list element is the largest one; that is, it tells us the largest value but does not specify its position. Furthermore, it will sometimes be convenient to remember which of sequence of calculations yielded the largest or smallest value without having to remember all the values simultaneously. For example, suppose that we want to remember which of the values sin(0), sin(1), sin(2), sin(3), … , sin(19) is the smallest, without having to store all of them simultaneously. We may do this as follows:

```
import math

bestY = float("inf")

for x in range(20) :
    y = math.sin(x)
    print("sin(" + str(x) + ") = " + str(y))
    if y < bestY :
        bestY = y
        bestX = x

print("Lowest value is " + str(bestY) + " at " + str(bestX))
```

The `import math` statement makes sure that the `math.sin` function is available. In the loop, which runs for `x` from 0 to 19, we first compute `y` to be sin(`x`). If it is lower than the best value seen so far (`y < bestY`) then we update `bestY` and also set `bestX` to be the current value of `x`. After the loop, we simple print the best *x* and *y* value.

A key point here is the initialization `bestY = float("inf")`, which means that the loop starts with `bestY` set to positive infinity. That means that the first time through the loop, whatever value is calculated for `y` will have to look better than `bestY`, and so `bestY` will always be updated and `bestX` is always set (to zero). The values of `bestX` and `bestY` will subsequently be replaced only when better values of `y` are encountered.

There are other ways to accomplish the same thing, but they have the disadvantage that the code that calculates `y` needs to be repeated in two places. In situations when that calculation is more complicated than in the example above, that could require duplicating some fairly large segment of code or require you to create a new function.

The same pattern can be used to calculate the largest rather than smallest value. We just have to initialize `bestY` to minus infinity (by `bestY = float("-inf")`), and reverse the comparison in the `if` statement to `y > bestY`.

NumPy Arrays and Installing a Package

The final ingredient we'll use to implement dynamic programming algorithm is the NumPy array class. It is not absolutely necessary for the kind of computations we need to do (we could get by with just lists), but NumPy arrays will make most of our programs run faster and slightly simplify some of the syntax.

Conceptually, a NumPy array is very similar to a list, except for a few features:

1. Every element of the array must have the same datatype.

2. An array may have more than one dimension (we will generally stick to one or two dimensions). If an array has two dimensions, it is a two-dimensional table of numbers.

3. Compared to a list, it is relatively difficult and inefficient to change the size of an array after it has been first created. On the other hand, doing computations on individual elements is much faster than with lists.

By making a list whose elements are themselves lists, it is possible to use lists to make data structures that behave similarly to two-dimensional or higher-dimensional array, but this technique is much less efficient in terms of both run time and memory.

There are several standard ways to create an array. One is to make an array out of a list. For example, suppose we change our program of the previous section to

```
import numpy

data = [43,3,3,27,-2,34,100,302,-17]

a = numpy.array(data)

n = len(a)

for i in range(n) :
    print("item " + str(i) + " is " + str(a[i]))
    if a[i] > 30 :
        print("Big")
    elif a[i] < 10 :
        print("Small")
    else :
        print("Meh")

print("Done")
```

Here the statement `import numpy` loads the NumPy module (you may notice a slight delay when this statement is executed, since NumPy is a large module). If you get an error message about NumPy being unknown at this point, you need to make sure that PyCharm is aware of the NumPy module. To do this,

1. Select “Settings…” from the File menu.

2. Find your project in the left-hand side of the resulting window, and click the little triangle next to it until you see “Project interpreter” just below its name.

3. Click on “Project interpreter”.

4. Double click in the resulting grid of information.

5. Type “numpy” in the resulting search box.

6. Select “numpy” in the resulting list and click “Install Package” at the lower left of the window.

7. Once you see the message “Package installed successfully”, close the resulting window and then click “OK” in the settings box.

Returning to our code, the statement `a = numpy.array(data)` makes a NumPy array called `a` containing the same information as `data`. After that, the program operates similarly, but using `a` instead of `data`.

Another function that creates arrays is `numpy.zeros( )`. This will make an array of a specified size containing all zeros. For example,

```
>>> import numpy
>>> numpy.zeros(5)
```

results in a one-dimensional array containing 5 zeros, as follows:

```
array([ 0.,  0.,  0.,  0.,  0.])
```

By default, NumPy arrays consist of floating point numbers. If we know we are only going to store integers in an array, we can use something like

```
>>> numpy.zeros(8,dtype=int)
```

which returns

```
array([0, 0, 0, 0, 0, 0, 0, 0])
```

The lack of periods in this output indicates that the data are integer, whereas the periods above indicate floating-point (fractions allowed) data.

Finally, we can create arrays that have two or more dimensions. For example,

```
>>> numpy.zeros([3,4])
```

produces a two-dimensional array of all floating-point zeroes with three rows and four columns, which prints like this:

```
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

Take careful note of where the square brackets are placed in the invocation of `numpy.zeros` above. It is tempting to omit the square brackets, but unfortunately that results in an error message.

Here is an example of a program that creates an array like the one above and fills it with random numbers:

```
import numpy
import random

X = numpy.zeros([3,4])

print(X)
print("")

for i in range(3) :
    for j in range(4) :
        X[i,j] = random.normalvariate(0,1)

print(X)

print("")

print(X[1,2])
```

Some observations about this program:

1. The statement `import random` loads the `random` module, which generates pseudo-random numbers. The expression `random.normalvariate(0,1)` generates a normal random variable with mean 0 and standard deviation 1. If your program generates an error about the `random` module being unknown, load it into PyCharm using the same kind of procedure that we used above for NumPy.

2. The statements `print("")` print blank lines (for clarity of the output).

3. Individual elements of a two-dimensional array are referenced like `X[i,j]`, where `i` is the row and `j` is the column. The rows and columns are both numbered starting at 0. In this case, the rows are numbered 0, 1, 2, and the columns are numbered 0, 1, 2, 3.

This program is our first example of a *nested loop* construction, which is one loop inside another. The outer loop in this case runs over `i`, and corresponds to processing rows of the array. For each value of `i`, the inner loop runs over `j` and fills element (`i`, `j`) of the array with a normally distributed random number. The overall effect is to fill the entire array with normally distributed (pseudo) random numbers, which we can see in the output:

```
[[ 0.57036017  1.7178748  -0.63823385  1.14276599]
 [ 0.6653349   1.53646982  0.00262121  2.33460792]
 [ 0.26782831  0.65844576 -0.8899263   0.82130578]]
```

Depending on how Python and PyCharm are configured, you might see different random numbers in the output. Remembering that rows and columns are both numbered starting at 0, element (1, 2) of this array is the second row and third column, as we can confirm by the final line of output

```
0.00262121232964
```