A crash course in Python for elementary numerical computing

January 10, 2016

This document was partly cobbled together by Mark McClure for his Math/CS 441 class in numerical computing at UNCA. It is largely based on a similar document by Rick Muller called A crash course in Python for Scientists - particularly, the extensive section entitled Python Overview.

Numerical computation in the 21st century is largely performed on a computer. Thus, a full understanding of numerical methods requires a numerical environment to explore implementation. For this purpose, we will use Python and the extensive family of numerical libraries known as SciPy. Python is the programming language of choice for many scientists to a large degree because it offers a great deal of power to analyze and model scientific data with relatively little overhead in terms of learning, installation or development time. The easiest way to get a full SciPy distribution up and running is to install Anaconda by Continuum Analytics.

0.1 Glossary

- Python: A popular, general purpose programming language with a number of advantages for an introductory class in numerical computing, including
- Relatively asy to get started with
- Runs interactively in a notebook
- Interpreted (no compilation step)
- A large number of libraries for scientific exploration and computing, like ...
- SciPy: One of the oldest and most important scientific libraries. Scipy can refer to the single library by that name or to the collection of core packages including:
- Numpy: The low level numeric library
- SciPy library: Higher level functions built on top of numpy
- Matplotlib: A graphics library for plotting results
- SymPy: A rudimentary symbolic library
- IPython: Extensions for Python that make it even more interactive. This has recently evolved into ...
- Jupyter: A language agnostic extention of IPython that makes it easy to interact with a large collection of programming languages. Of key importance for us is the Jupyter notebook.
- Anaconda: A Python distribution that includes all the tools we'll need.

For a quick jump into numerical computing, you can try out the online Jupyter notebook. If you're going to continue in your studies of numerical computing, you should undoubtedly install your own version of Anaconda. Either way, the rest of this document assumes that you've got some version of Python and it's numerical tools running.

0.2 Python Overview

This is a quick introduction to Python. There are lots of other places to learn the language more thoroughly and there is a collection of useful links, including ones to other learning resources, at the end of this notebook.

0.2.1 Using Python as a Calculator

Many of the things I used to use a calculator for, I now use Python for:

```
In [1]: 2+2

Out[1]: 4

In [2]: (50-5*6)/4

Out[2]: 5.0
```

(If you're typing this into an IPython notebook, or otherwise using notebook file, you hit shift-Enter to evaluate a cell.)

You can define variables using the equals (=) sign:

```
In [3]: width = 20
        length = 30
        area = length*width
        area

Out[3]: 600
```

If you try to access a variable that you haven't yet defined, you get an error:

```
In [4]: volume


        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-4-0c7fc58f9268> in <module>()
    ----> 1 volume


        NameError: name 'volume' is not defined
```

and you need to define it:

```
In [5]: depth = 10
        volume = area*depth
        volume

Out[5]: 6000
```

You can name a variable *almost* anything you want. It needs to start with an alphabetical character or "_", can contain alphanumeric charcters plus underscores ("_"). Certain words, however, are reserved for the language:

```
and, as, assert, break, class, continue, def, del, elif, else, except,
exec, finally, for, from, global, if, import, in, is, lambda, not, or,
pass, print, raise, return, try, while, with, yield
```

Trying to define a variable using one of these will result in a syntax error:

```
In [6]: return = 0
```

```
      File "<ipython-input-6-2b99136d4ec6>", line 1
    return = 0
           ^
SyntaxError: invalid syntax
```

Now, let's take a look at an `import` statement. Python has a huge number of libraries included with the distribution. To keep things simple, most of these variables and functions are not accessible from a normal Python interactive session. Instead, you have to import the name. For example, there is a `math` module containing many useful functions. To access, say, the square root function, you can either first

```
In [7]: from math import sqrt
```

and then

```
In [8]: sqrt(81)
```

```
Out[8]: 9.0
```

or you can simply import the math library itself

```
In [9]: import math
        math.sqrt(81)
```

```
Out[9]: 9.0
```

Note, however, that a numerical analyst will often prefer to use the math functionality available in `numpy`.

0.2.2 Data types

Numbers Python is dynamically typed, but it is still typed. In particular, there are three disctinct kinds of numbers `int`, `float`, and `complex`.

An `int` can have unlimited precision.

```
In [10]: 2**(1001)-1
```

```
Out[10]: 21430172143725346418968500981200036211228096234110672148875007767407021022498722449863967576313
```

A `complex` has a real and imaginary part and is easily entered in the form `a+bj`.

```
In [11]: (2+3j)**3
```

```
Out[11]: (-46+9j)
```

We are mostly interested in `floats` which can be used to model real numbers and are typically implemented internally as C `doubles`. Note that there is a distinction between a `float` and an `int`.

```
In [12]: 1 is 1.0
```

```
Out[12]: False
```

Strings Strings are lists of printable characters, and can be defined using either single quotes

```
In [13]: 'Hello, World!'

Out[13]: 'Hello, World!'
```

or double quotes

```
In [14]: "Hello, World!"

Out[14]: 'Hello, World!'
```

But not both at the same time, unless you want one of the symbols to be part of the string.

```
In [15]: "He's a Rebel"

Out[15]: "He's a Rebel"

In [16]: 'She asked, "How are you today?"'

Out[16]: 'She asked, "How are you today?"'
```

Just like the other two data objects we're familiar with (ints and floats), you can assign a string to a variable

```
In [17]: greeting = "Hello, World!"
```

The **print** statement is often used for printing character strings:

```
In [18]: print(greeting)

Hello, World!
```

But it can also print data types other than strings:

```
In [19]: print("The area is ",area)

The area is  600
```

In the above snipped, the number 600 (stored in the variable "area") is converted into a string before being printed out.
You can use the + operator to concatenate strings together:

```
In [20]: statement = "Hello," + "World!"
         print(statement)

Hello,World!
```

Don't forget the space between the strings, if you want one there.

```
In [21]: statement = "Hello, " + "World!"
         print(statement)

Hello, World!
```

You can use + to concatenate multiple strings in a single statement:

```
In [22]: print("This " + "is " + "a " + "longer " + "statement.")

This is a longer statement.
```

If you have a lot of words to concatenate together, there are other, more efficient ways to do this. But this is fine for linking a few strings together.

Lists Very often in a programming language, one wants to keep a group of similar items together. Python does this using a data type called **lists**.

```
In [23]: days_of_the_week = ["Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"]
```

You can access members of the list using the **index** of that item:

```
In [24]: days_of_the_week[2]

Out[24]: 'Tuesday'
```

Python lists, like C, but unlike Fortran, use 0 as the index of the first element of a list. Thus, in this example, the 0 element is “Sunday”, 1 is “Monday”, and so on. If you need to access the *n*th element from the end of the list, you can use a negative index. For example, the -1 element of a list is the last element:

```
In [25]: days_of_the_week[-1]

Out[25]: 'Saturday'
```

You can add additional items to the list using the .append() command:

```
In [26]: languages = ["Fortran","C","C++"]
         languages.append("Python")
         print(languages)

['Fortran', 'C', 'C++', 'Python']
```

The **range()** command is a convenient way to make sequential lists of numbers:

```
In [27]: list(range(10))

Out[27]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Note that range(n) starts at 0 and gives the sequential list of integers less than n. If you want to start at a different number, use range(start,stop)

```
In [28]: list(range(2,8))

Out[28]: [2, 3, 4, 5, 6, 7]
```

The lists created above with range have a *step* of 1 between elements. You can also give a fixed step size via a third command:

```
In [29]: evens = list(range(0,20,2))
         evens

Out[29]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [30]: evens[3]

Out[30]: 6
```

Lists do not have to hold the same data type. For example,

```
In [31]: ["Today",7,99.3,""]

Out[31]: ['Today', 7, 99.3, '']
```

However, it's good (but not essential) to use lists for similar objects that are somehow logically connected. If you want to group different data types together into a composite data object, it's best to use **tuples**, which we will learn about below.

You can find out how long a list is using the **len()** command:

```
In [32]: help(len)

Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.

In [33]: len(evens)

Out[33]: 10
```

0.2.3 Iteration, Indentation, and Blocks

One of the most useful things you can do with lists is to *iterate* through them, i.e. to go through each element one at a time. To do this in Python, we use the **for** statement:

```
In [34]: for day in days_of_the_week:
             print(day)

Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

This code snippet goes through each element of the list called **days_of_the_week** and assigns it to the variable **day**. It then executes everything in the indented block (in this case only one line of code, the print statement) using those variable assignments. When the program has gone through every element of the list, it exists the block.

(Almost) every programming language defines blocks of code in some way. In Fortran, one uses END statements (ENDDO, ENDIF, etc.) to define code blocks. In C, C++, and Perl, one uses curly braces {} to define these blocks.

Python uses a colon (":"), followed by indentation level to define code blocks. Everything at a higher level of indentation is taken to be in the same block. In the above example the block was only a single line, but we could have had longer blocks as well:

```
In [35]: for day in days_of_the_week:
             statement = "Today is " + day
             print(statement)

Today is Sunday
Today is Monday
Today is Tuesday
Today is Wednesday
Today is Thursday
Today is Friday
Today is Saturday
```

The `range` command is particularly useful with the **for** statement to execute loops of a specified length. Note that if we enter just a `range` we get, well, a `range`.

```
In [36]: range(20)
```

```
Out[36]: range(0, 20)
```

This is why we wrapped the previous `range`'s in `list`. While this might seem odd, the point is that it is an iterable.

```
In [37]: for i in range(20):
             print("The square of ",i," is ",i*i)
```

```
The square of  0  is  0
The square of  1  is  1
The square of  2  is  4
The square of  3  is  9
The square of  4  is  16
The square of  5  is  25
The square of  6  is  36
The square of  7  is  49
The square of  8  is  64
The square of  9  is  81
The square of  10  is  100
The square of  11  is  121
The square of  12  is  144
The square of  13  is  169
The square of  14  is  196
The square of  15  is  225
The square of  16  is  256
The square of  17  is  289
The square of  18  is  324
The square of  19  is  361
```

0.2.4 Slicing

Lists and strings have something in common that you might not suspect: they can both be treated as sequences. You already know that you can iterate through the elements of a list. You can also iterate through the letters in a string:

```
In [38]: for letter in "Sunday":
             print(letter)
```

```
S
u
n
d
a
y
```

This is only occasionally useful. Slightly more useful is the *slicing* operation, which you can also use on any sequence. We already know that we can use *indexing* to get the first element of a list:

```
In [39]: days_of_the_week[0]
```

```
Out[39]: 'Sunday'
```

If we want the list containing the first two elements of a list, we can do this via

```
In [40]: days_of_the_week[0:2]
```

```
Out[40]: ['Sunday', 'Monday']
```

or simply

```
In [41]: days_of_the_week[:2]
```

```
Out[41]: ['Sunday', 'Monday']
```

If we want the last items of the list, we can do this with negative slicing:

```
In [42]: days_of_the_week[-2:]
```

```
Out[42]: ['Friday', 'Saturday']
```

which is somewhat logically consistent with negative indices accessing the last elements of the list.
You can do:

```
In [43]: workdays = days_of_the_week[1:6]
         print(workdays)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Since strings are sequences, you can also do this to them:

```
In [44]: day = "Sunday"
         abbreviation = day[:3]
         print(abbreviation)
```

```
Sun
```

If we really want to get fancy, we can pass a third element into the slice, which specifies a step length (just like a third argument to the **range()** function specifies the step):

```
In [45]: numbers = range(0,40)
         evens = numbers[2::2]
         list(evens)
```

```
Out[45]: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38]
```

Note that in this example I was even able to omit the second argument, so that the slice started at 2, went to the end of the list, and took every second element, to generate the list of even numbers less that 40.

0.2.5 Booleans and Truth Testing

We have now learned a few data types. We have integers and floating point numbers, strings, and lists to contain them. We have also learned about lists, a container that can hold any data type. We have learned to print things out, and to iterate over items in lists. We will now learn about **boolean** variables that can be either True or False.

We invariably need some concept of *conditions* in programming to control branching behavior, to allow a program to react differently to different situations. If it's Monday, I'll go to work, but if it's Sunday, I'll sleep in. To do this in Python, we use a combination of **boolean** variables, which evaluate to either True or False, and **if** statements, that control branching based on boolean values.

For example:

```
In [46]: if day == "Sunday":
             print("Sleep in")
         else:
             print("Go to work")
```

```
Sleep in
```

(Quick quiz: why did the snippet print "Go to work" here? What is the variable "day" set to?)
Let's take the snippet apart to see what happened. First, note the statement

```
In [47]: day == "Sunday"

Out[47]: True
```

If we evaluate it by itself, as we just did, we see that it returns a boolean value, False. The "==" operator performs *equality testing*. If the two items are equal, it returns True, otherwise it returns False. In this case, it is comparing two variables, the string "Sunday", and whatever is stored in the variable "day", which, in this case, is the other string "Saturday". Since the two strings are not equal to each other, the truth test has the false value.

The if statement that contains the truth test is followed by a code block (a colon followed by an indented block of code). If the boolean is true, it executes the code in that block. Since it is false in the above example, we don't see that code executed.

The first block of code is followed by an **else** statement, which is executed if nothing else in the above if statement is true. Since the value was false, this code is executed, which is why we see "Go to work".

You can compare any data types in Python:

```
In [48]: 1 == 2

Out[48]: False

In [49]: 50 == 2*25

Out[49]: True

In [50]: 3 < 3.14159

Out[50]: True

In [51]: 1 == 1.0

Out[51]: True

In [52]: 1 != 0

Out[52]: True

In [53]: 1 <= 2

Out[53]: True

In [54]: 1 >= 1

Out[54]: True
```

We see a few other boolean operators here, all of which which should be self-explanatory. Less than, equality, non-equality, and so on.

Particularly interesting is the 1 == 1.0 test, which is true, since even though the two objects are different data types (integer and floating point number), they have the same *value*. There is another boolean operator **is**, that tests whether two objects are the same object:

```
In [55]: 1 is 1.0

Out[55]: False
```

We can do boolean tests on lists as well:

```
In [56]: [1,2,3] == [1,2,4]
```

```
Out[56]: False
```

```
In [57]: [1,2,3] < [1,2,4]
```

```
Out[57]: True
```

Finally, note that you can also string multiple comparisons together, which can result in very intuitive tests:

```
In [58]: hours = 5
         0 < hours < 24
```

```
Out[58]: True
```

If statements can have **elif** parts (“else if”), in addition to if/else parts. For example:

```
In [59]: if day == "Sunday":
             print("Sleep in")
         elif day == "Saturday":
             print("Do chores")
         else:
             print("Go to work")
```

```
Sleep in
```

Of course we can combine if statements with for loops, to make a snippet that is almost interesting:

```
In [60]: for day in days_of_the_week:
             statement = "Today is " + day
             print(statement)
             if day == "Sunday":
                 print("   Sleep in")
             elif day == "Saturday":
                 print("   Do chores")
             else:
                 print("   Go to work")
```

```
Today is Sunday
   Sleep in
Today is Monday
   Go to work
Today is Tuesday
   Go to work
Today is Wednesday
   Go to work
Today is Thursday
   Go to work
Today is Friday
   Go to work
Today is Saturday
   Do chores
```

This is something of an advanced topic, but ordinary data types have boolean values associated with them, and, indeed, in early versions of Python there was not a separate boolean object. Essentially, anything that was a 0 value (the integer or floating point 0, an empty string "", or an empty list []) was False, and everything else was true. You can see the boolean value of any data object using the **bool()** function.

```
In [61]: bool(1)

Out[61]: True

In [62]: bool(0)

Out[62]: False

In [63]: bool(["This "," is "," a "," list"])

Out[63]: True
```

1 Defining functions

A function is a block of reusable code that is used to perform a single task, typically repeatedly. The easiest functions, from the numerical perspective, simply define a mathematical function. For example, $f(x) = x^2 - x - 1$ can be written (using the `def` operator) and used as follows:

```
In [64]: def f(x): return x**2 - x - 1
         f(3)

Out[64]: 5
```

Note the `**` to represent exponentiation! Let's examine a deeper example involving the Fibonacci sequence. This the sequence (F_n) defined by $F_0 = 0$, F_1=1, and $F_{n+1} = F_n + F_{n-1}$. In words, it's the sequence that starts with 0 and 1, and then each successive entry is the sum of the previous two. Thus, the sequence goes 0,1,1,2,3,5,8,13,21,34,55,89,...

A very common exercise in programming books is to compute the Fibonacci sequence up to some number **n**. First I'll show the code, then I'll discuss what it is doing.

```
In [65]: n = 10
         sequence = [0,1]
         for i in range(2,n): # This is going to be a problem if we ever set n <= 2!
             sequence.append(sequence[i-1]+sequence[i-2])
         print(sequence)

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Let's go through this line by line. First, we define the variable **n**, and set it to the integer 20. **n** is the length of the sequence we're going to form, and should probably have a better variable name. We then create a variable called **sequence**, and initialize it to the list with the integers 0 and 1 in it, the first two elements of the Fibonacci sequence. We have to create these elements "by hand", since the iterative part of the sequence requires two previous elements.

We then have a for loop over the list of integers from 2 (the next element of the list) to **n** (the length of the sequence). After the colon, we see a hash tag "#", and then a **comment** that if we had set **n** to some number less than 2 we would have a problem. Comments in Python start with #, and are good ways to make notes to yourself or to a user of your code explaining why you did what you did. Better than the comment here would be to test to make sure the value of **n** is valid, and to complain if it isn't; we'll try this later.

In the body of the loop, we append to the list an integer equal to the sum of the two previous elements of the list.

After exiting the loop (ending the indentation) we then print out the whole list. That's it!

We might want to use the Fibonacci snippet with different sequence lengths. We could cut an paste the code into another cell, changing the value of **n**, but it's easier and more useful to make a function out of the code. We do this with the **def** statement in Python:

```
In [66]: def fibonacci(sequence_length):
             "Return the Fibonacci sequence of length *sequence_length*"
             sequence = [0,1]
             if sequence_length < 1:
                 print("Fibonacci sequence only defined for length 1 or greater")
                 return
             if 0 < sequence_length < 3:
                 return sequence[:sequence_length]
             for i in range(2,sequence_length):
                 sequence.append(sequence[i-1]+sequence[i-2])
             return sequence
```

We can now call **fibonacci()** for different sequence_lengths:

```
In [67]: fibonacci(2)

Out[67]: [0, 1]

In [68]: fibonacci(12)

Out[68]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

We've introduced a several new features here. First, note that the function itself is defined as a code block (a colon followed by an indented block). This is the standard way that Python delimits things. Next, note that the first line of the function is a single string. This is called a **docstring**, and is a special kind of comment that is often available to people using the function through the python command line:

```
In [69]: help(fibonacci)

Help on function fibonacci in module __main__:

fibonacci(sequence_length)
    Return the Fibonacci sequence of length *sequence_length*
```

If you define a docstring for all of your functions, it makes it easier for other people to use them, since they can get help on the arguments and return values of the function.

Next, note that rather than putting a comment in about what input values lead to errors, we have some testing of these values, followed by a warning if the value is invalid, and some conditional code to handle special cases.

1.1 Recursion and Factorials

Functions can also call themselves, something that is often called *recursion*. We're going to experiment with recursion by computing the factorial function. The factorial is defined for a positive integer **n** as

$$n! = n(n-1)(n-2)\cdots 1$$

First, note that we don't need to write a function at all, since this is a function built into the standard math library. Let's use the help function to find out about it:

```
In [70]: from math import factorial
         help(factorial)
```

```
Help on built-in function factorial in module math:

factorial(...)
    factorial(x) -> Integral

    Find x!. Raise a ValueError if x is negative or non-integral.
```

This is clearly what we want.

```
In [71]: factorial(20)
```

```
Out[71]: 2432902008176640000
```

However, if we did want to write a function ourselves, we could do recursively by noting that

$$n! = n(n-1)!$$

The program then looks something like:

```
In [72]: def fact(n):
             if n <= 0:
                 return 1
             return n*fact(n-1)
```

```
In [73]: fact(20)
```

```
Out[73]: 2432902008176640000
```

Recursion can be very elegant, and can lead to very simple programs.

1.2 Two More Data Structures: Tuples and Dictionaries

Before we end the Python overview, I wanted to touch on two more data structures that are very useful (and thus very common) in Python programs.

A **tuple** is a sequence object like a list or a string. It's constructed by grouping a sequence of objects together with commas, either without brackets, or with parentheses:

```
In [74]: t = (1,2,'hi',9.0)
         t
```

```
Out[74]: (1, 2, 'hi', 9.0)
```

Tuples are like lists, in that you can access the elements using indices:

```
In [75]: t[1]
```

```
Out[75]: 2
```

However, tuples are *immutable*, you can't append to them or change the elements of them:

```
In [76]: t.append(7)
```

```
        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-76-50c7062b1d5f> in <module>()
    ----> 1 t.append(7)


        AttributeError: 'tuple' object has no attribute 'append'
```

```
In [77]: t[1]=77


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-77-03cc8ba9c07d> in <module>()
    ----> 1 t[1]=77


        TypeError: 'tuple' object does not support item assignment
```

Tuples are useful anytime you want to group different pieces of data together in an object, but don't want to create a full-fledged class (see below) for them. For example, let's say you want the Cartesian coordinates of some objects in your program. Tuples are a good way to do this:

```
In [78]: ('Bob',0.0,21.0)

Out[78]: ('Bob', 0.0, 21.0)
```

Again, it's not a necessary distinction, but one way to distinguish tuples and lists is that tuples are a collection of different things, here a name, and x and y coordinates, whereas a list is a collection of similar things, like if we wanted a list of those coordinates:

```
In [79]: positions = [
                      ('Bob',0.0,21.0),
                      ('Cat',2.5,13.1),
                      ('Dog',33.0,1.2)
                      ]
```

Tuples can be used when functions return more than one value. Say we wanted to compute the smallest x- and y-coordinates of the above list of objects. We could write:

```
In [80]: def minmax(objects):
             minx = 1e20 # These are set to really big numbers
             miny = 1e20
             for obj in objects:
                 name,x,y = obj
                 if x < minx:
                     minx = x
                 if y < miny:
                     miny = y
             return minx,miny

         x,y = minmax(positions)
         print(x,y)

0.0 1.2
```

Here we did two things with tuples you haven't seen before. First, we unpacked an object into a set of named variables using *tuple assignment*:

```
>>> name,x,y = obj
```

We also returned multiple values (minx,miny), which were then assigned to two other variables (x,y), again by tuple assignment. This makes what would have been complicated code in C++ rather simple.

Tuple assignment is also a convenient way to swap variables:

```
In [81]: x,y = 1,2
         y,x = x,y
         x,y

Out[81]: (2, 1)
```

Dictionaries are an object called "mappings" or "associative arrays" in other languages. Whereas a list associates an integer index with a set of objects:

```
In [82]: mylist = [1,2,9,21]
```

The index in a dictionary is called the *key*, and the corresponding dictionary entry is the *value*. A dictionary can use (almost) anything as the key. Whereas lists are formed with square brackets [], dictionaries use curly brackets {}:

```
In [83]: ages = {"Rick": 46, "Bob": 86, "Fred": 21}
         print("Rick's age is ",ages["Rick"])

Rick's age is  46
```

There's also a convenient way to create dictionaries without having to quote the keys.

```
In [84]: dict(Rick=46,Bob=86,Fred=20)

Out[84]: {'Bob': 86, 'Fred': 20, 'Rick': 46}
```

The **len()** command works on both tuples and dictionaries:

```
In [85]: len(t)

Out[85]: 4

In [86]: len(ages)

Out[86]: 3
```

2 Important libraries for numerical analysis

In addition to being an excellent and easy to use general purpose language, there are a large number of libraries that make Python an excellent tool for numerical computing. In this section, we exlore a few of these. Note that this is a *very* brief intro since we will be learning more and more about these packages throughout the text.

2.1 Numpy and Scipy

Numpy and Scipy are absolutely central for numerical computing with Python. Numpy is the lower level of the two and contains core routines for doing fast vector, matrix, and linear algebra-type operations in Python. SciPy, built on top of NumPy, is higher level and contains additional routines for optimization, special functions, and so on. Both contain modules written in C and Fortran so that they're as fast as possible. Together, they give Python roughly the same capability that the Matlab program offers. (In fact, if you're an experienced Matlab user, there a guide to Numpy for Matlab users just for you.)

There are several ways to import NumPy:

```
In [87]: # Import all of NumPy into the global namespace
         from numpy import *
```

```
In [88]: # Import NumPy into it's own namespace
         import numpy
```

```
In [89]: # Import NumPy into a namespace with a shorter name
         import numpy as np
```

The first version is particularly convenient for interactive work but frowned upon for serious programming. We will often use the third version. Among other things, this provides some trig and related tools. Here's how to compute the cosine of π:

```
In [90]: np.cos(np.pi)

Out[90]: -1.0
```

SciPy provides higher level functionality largely broken into submodules. If we want to integrate a function numerically, for example, we can use the `quad` function provided in the `scipy.integrate` module.

```
In [91]: from scipy.integrate import quad
         quad(np.sin, 0, np.pi)

Out[91]: (2.0, 2.220446049250313e-14)
```

This says that the value of the integral is 2.0 to an accuracy of about 2.22×10^{-14}. We'll talk much more about this later!

NumPy and SciPy fundamentally work very well with arrays. These might represent vectors, matrices, or some other structured data. When they do represent vectors or matrices, SciPy has natural built in tools for linear algebra. In numerical analysis, we are often interested in studying the behavior of functions over an interval by sampling the function over an interval; the `linspace` command is useful in this context. For example, the following generates 101 points evenly distributed throughout the interval $[-2, 2]$.

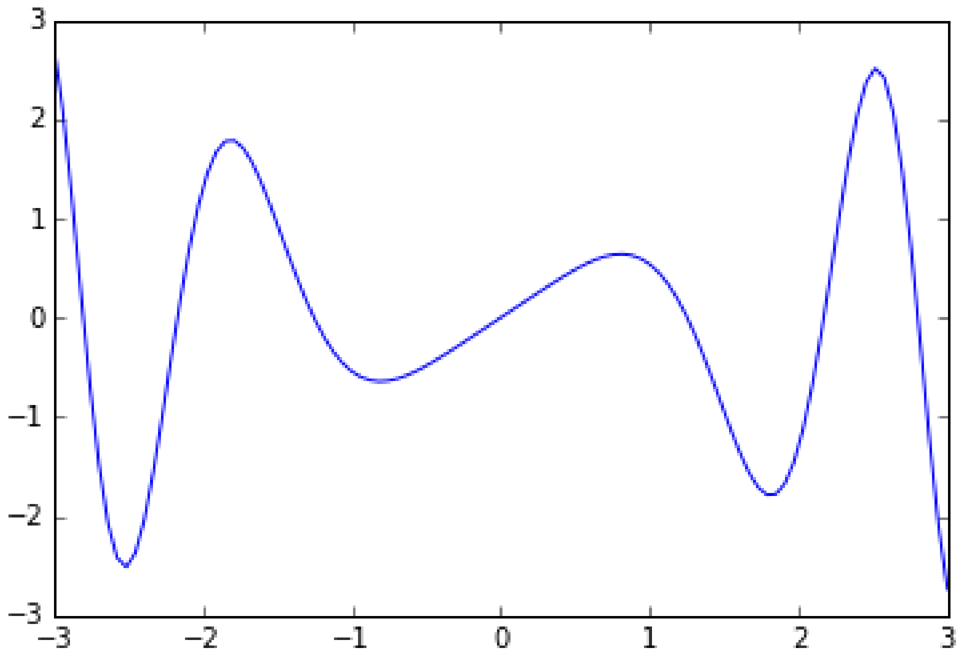
```
In [92]: np.linspace(-2,2,101)

Out[92]: array([-2.  , -1.96, -1.92, -1.88, -1.84, -1.8 , -1.76, -1.72, -1.68,
                -1.64, -1.6 , -1.56, -1.52, -1.48, -1.44, -1.4 , -1.36, -1.32,
                -1.28, -1.24, -1.2 , -1.16, -1.12, -1.08, -1.04, -1.  , -0.96,
                -0.92, -0.88, -0.84, -0.8 , -0.76, -0.72, -0.68, -0.64, -0.6 ,
                -0.56, -0.52, -0.48, -0.44, -0.4 , -0.36, -0.32, -0.28, -0.24,
                -0.2 , -0.16, -0.12, -0.08, -0.04,  0.  ,  0.04,  0.08,  0.12,
                 0.16,  0.2 ,  0.24,  0.28,  0.32,  0.36,  0.4 ,  0.44,  0.48,
                 0.52,  0.56,  0.6 ,  0.64,  0.68,  0.72,  0.76,  0.8 ,  0.84,
                 0.88,  0.92,  0.96,  1.  ,  1.04,  1.08,  1.12,  1.16,  1.2 ,
                 1.24,  1.28,  1.32,  1.36,  1.4 ,  1.44,  1.48,  1.52,  1.56,
                 1.6 ,  1.64,  1.68,  1.72,  1.76,  1.8 ,  1.84,  1.88,  1.92,
                 1.96,  2.  ])
```

2.2 Matplotlib

Often, we need to visualize our work. While there are a number of visualization and plotting libraries, Matplotlib is the most widely used and works well with output prodcues by NumPy and SciPy.

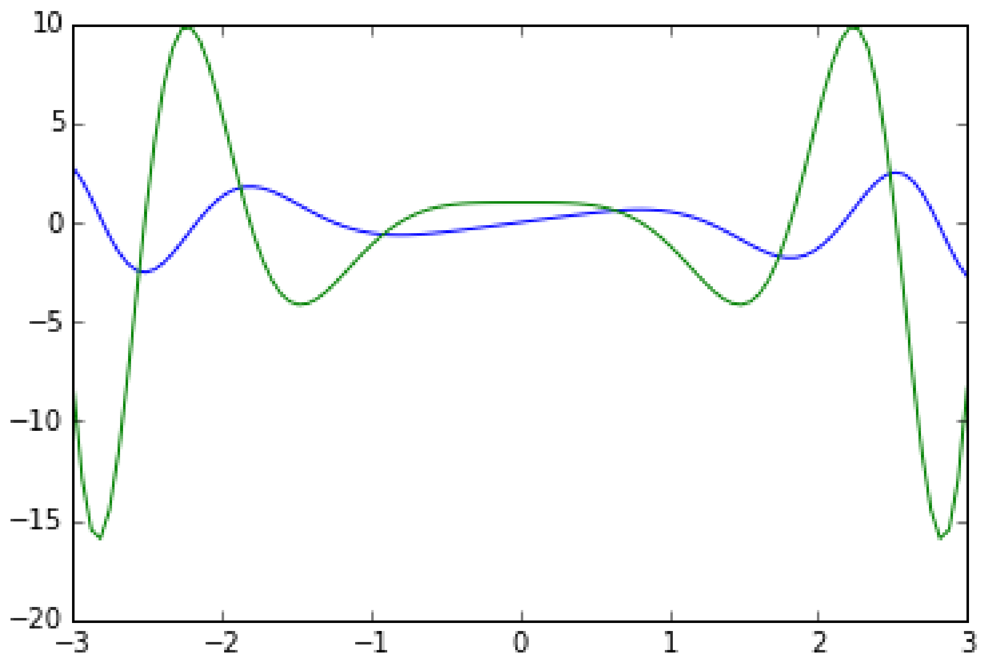
```
In [93]: %matplotlib inline
         import matplotlib.pyplot as plt
         import numpy as np
```

```
In [94]: def f(x): return x*np.cos(x**2)
         x = np.linspace(-3,3,100)
         y = f(x)
         plt.plot(x,y)

Out[94]: [<matplotlib.lines.Line2D at 0x108783c18>]
```



We can plot more than one function by simply calling `plt.plot` multiple times. Here's the graph of f together with is derivative.

```
In [95]: def fp(x): return np.cos(x**2) - 2*x**2 * np.sin(x**2)
         yp = fp(x)
         plt.plot(x,y)
         plt.plot(x,yp)

Out[95]: [<matplotlib.lines.Line2D at 0x10875c860>]
```



There is *much* more that can be done with Matplotlib, as we'll see throughout this book. Just this much is very useful, however. Wouldn't it be useful if we could have computed f' in the last example symbolically, though?

2.3 SymPy

SymPy is a rudimentary symbolic library for Python. I write "rudimentary" because, frankly, if you are used to working with a commerical computer algebra system, like Mathematica or Maple, or even a good open source tool, like Maxima, you might be disapointed in SymPy. It is simply much slower and less capable than those tools. Nonetheless, it is can be convenient to have a tool for basic symbolic computation and it works well with the other tools we are using for numerical computation. Other advantages of SymPy include the fact that it is open source and can easily be used a library.

SymPy is really not central to our needs; we will typically use it to run a few side symbolic computations. In this context, it makes sense to simply import all of SymPy into the global namespace and work with it interactively. In fact, SymPy is set up largely to work in exactly that fashion, emulating the way most computer algebra systems work.

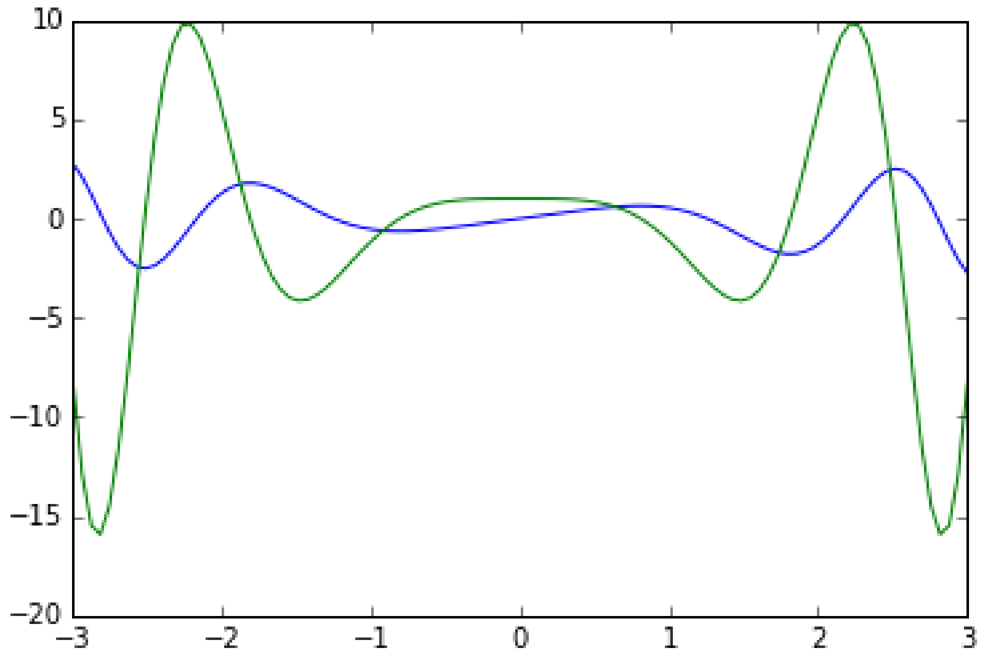
```
In [96]: from sympy import *
```

Unlike most computer algebra systems, symbols must be declared as such. After doing so, we can do fun things like differentiate the function symbolically.

```
In [97]: x = symbols('x')
         expr = x*cos(x**2)
         deriv = diff(expr,x)
         deriv

Out[97]: -2*x**2*sin(x**2) + cos(x**2)
```

Now, we can plot the function together with it's derivative like so:

```
In [98]: f = lambdify(x,expr, 'numpy')
         fp = lambdify(x,diff(expr,x), 'numpy')
         xs = np.linspace(-3,3,100)
         plt.plot(xs,f(xs))
         plt.plot(xs,fp(xs))

Out[98]: [<matplotlib.lines.Line2D at 0x10871e518>]
```



3 References

3.1 Learning Resources

- Official Python Documentation, including
  - Python Tutorial
  - Python Language Reference
- If you're interested in Python 3, the Official Python 3 Docs are here.
- IPython tutorial.
- Learn Python The Hard Way
- Dive Into Python, in particular if you're interested in Python 3.
- Invent With Python, probably best for kids.
- Python Functional Programming HOWTO
- The Structure and Interpretation of Computer Programs, written in Scheme, a Lisp dialect, but one of the best books on computer programming ever written.
- Generator Tricks for Systems Programmers Beazley's slides on just what generators can do for you.
- Python Module of the Week is a series going through in-depth analysis of the Python standard library in a very easy to understand way.

3.2 Badass Jupyter notebooks for Python

- Rob Johansson's excellent notebooks, including Scientific Computing with Python and Computational Quantum Physics with QuTiP lectures;
- XKCD style graphs in matplotlib;
- A collection of Notebooks for using IPython effectively
- A gallery of interesting IPython Notebooks
- Cross-disciplinary computational analysis IPython Notebooks From Hadoop World 2012
- Quantites Units in Python.