Lecture-3

May 28, 2024

1 Introduction to Computational Software

Frank Cichos (2018)

2 Table of Contents

---

3 Lecture 3

In this lecture, we will explore conditionals, loops and other structuring elements of common programming. They are occurring in all programming languages in similar ways.

```
Populating the interactive namespace from numpy and matplotlib
```

3.1 Conditionals

Jump to top

3.1.1 if, elif, and else statements

The *if, elif*, and *else* statements are used to define conditionals in Python. The *if* keyword is followed by a condition, i.e. a logical comparison, which can either be **true** or **false**. We illustrate their use with a few examples.

Suppose we want to know if the solutions to the quadratic equation

$$ax^2 + bx + c = 0 \tag{1}$$

are real, imaginary, or complex for a given set of coefficients a, b, and c. Of course, the answer to that question depends on the value of the discriminant $d = b^2 - 4ac$. The solutions are real if $d \geq 0$, imaginary if $b = 0$ and $d < 0$,and complex if $b \neq 0$ and $d < 0$. The program below implements the above logic in a Python program. Each part of these statements has a code block to be executed. The code block to be executed is discriminated from the others by an indentation. Usually this is done with a *tab*, which indents 4 characters. You can either insert 4 spaces or press *tab*. Play around with the *tab* to get a feeling how the Jupyter notebook responds to that.

** if example**

```
Please input a number: -1
The absolute value is 1
```

** if else example**

```
Please input an integer: 212
212 is an even number.
```

** if, elif, else example **

```
What is the coefficients a? 1
What is the coefficients b? 2
What is the coefficients c? 7
Solutions are complex
Finished!
```

3.2 Logical Operators

It is important to understand that "==" in Python is not the same as "=". The operator "=" is the assignment operator: d = 5 assigns the value of 5 to the variable d. On the other hand "==" is the logical equals operator and d == 5 is a logical truth statement. It tells Python to check to see if d is equal to 5 or not, and assigns a value of True or False to the statement d == 5 depending on whether or not d is equal to 5. The table below summarizes the various logical operators available in Python.

3.3 Loops

Jump to top

In computer programming a loop is statement or block of statements that is executed repeatedly. Python has two kinds of loops, a for loop and a while loop. We first introduce the for loop and illustrate its use for a variety of tasks. We then introduce the while loop and, after a few illustrative examples, compare the two kinds of loops and discuss when to use one or the other.

3.3.1 for loops

The general form of a for loop in Python is

```
 for <itervar> in <sequence>:
     <body>
```

where < intervar > is a variable < sequence > is a sequence such as list or string or array, and < body > is a series of Python commands to be executed repeatedly for each element in the < sequence >. The < body > is indented from the rest of the text, which difines the extent of the loop. Let's look at a few examples.

```
Max
 Arf, arf!
Molly
 Arf, arf!
Buster
 Arf, arf!
Maggie
 Arf, arf!
Lucy
 Arf, arf!
All done.
```

The for loop works as follows: the iteration variable or loop index dogname is set equal to the first element in the list, "Max", and then the two lines in the indented body are executed. Then dogname is set equal to second element in the list, "Molly", and the two lines in the indented body are executed. The loop cycles through all the elements of the list,and then moves on to the code that follows the for loop and prints "All done".

```
2500

 2500
```

3.3.2 while loops

The general form of a while loop in Python is

```
while <condition>:
    <body>
```

where < condition > is a statement that can be either *True* or *False* and < body > is a series of Python commands that is executed repeatedly until < condition > becomes *false*. This means that somewhere in < body >, the truth value of < condition > must be changed so that it becomes *false* after a finite number of iterations. This is one of the great dangers of a while loop, that accidentially the condition might never become *false* and your loop runs forever.

Consider the following example. Suppose you want to calculate all the Fibonacci numbers smaller than 1000. The Fibonacci numbers are determined by starting with the integers 0 and 1. The next number in the sequence is the sum of the previous two. So, starting with 0 and 1, the next Fibonacci number is 0 + 1 = 1, giving the sequence 0, 1, 1. Continuing this process, we obtain 0, 1, 1, 2, 3, 5, 8, … where each element in the list is the sum of the previous two. Using a for loop to calculate the Fibonacci numbers is impractical because we do not know ahead of time how many Fibonacci numbers there are smaller than 1000. By contrast a while loop is perfect for calculating all the Fibonacci numbers because it keeps calculating Fibonacci numbers until it reaches the desired goal, in this case 1000. Here is the code using a while loop.

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

3.3.3 Loops and array operations

Loops are often used to sequentially modify the elements of an array. For example, suppose we want to square each element of the array *a = np.linspace(0, 32, 1e7)*. This is a hefty array with 10 million elements. Nevertheless, the following loop does the trick.

```
/Users/fci/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:2:
DeprecationWarning: object of type <class 'float'> cannot be safely interpreted
as an integer.


[0.00000000e+00 3.20000032e-06 6.40000064e-06 … 3.19999936e+01
 3.19999968e+01 3.20000000e+01]
[0.00000000e+00 1.02400020e-11 4.09600082e-11 … 1.02399959e+03
```

```
 1.02399980e+03 1.02400000e+03]
```

Running this on my computer returns the result in about 6 seconds—not bad for having performed 10 million multiplications. Of course we could have performed the same calculation using the array multiplication we learned in Lecture 1 (Strings, Lists, Arrays, and Dictionaries). Here is the code.

```
[  0.00000000e+00   3.20000032e-06   6.40000064e-06 …,   3.19999936e+01
   3.19999968e+01   3.20000000e+01]
[  0.00000000e+00   1.02400020e-11   4.09600082e-11 …,   1.02399959e+03
   1.02399980e+03   1.02400000e+03]
```

Running this on my computer returns the results in 190 millisecond. This illustrates an important point: **for loops are slow**. Array operations run much faster and are therefore to be preferred in any case where you have a choice. Sometimes finding an array operation that is equivalent to a loop can be difficult, especially for a novice. Nevertheless, doing so pays rich rewards in execution time. Moreover, the array notation is usually simpler and clearer, providing further reasons to prefer array operations over loops.

3.3.4 List Comprehensions

List comprehensions are a special feature of core Python for processing and constructing lists. We introduce them here because they use a looping process. They are used quite commonly in Python coding and they often provide elegant compact solutions to some common computing tasks.

Suppose we want to construct a vector from the diagonal elements of this matrix. We could do so with a for loop with an accumulator as follows

```
[1, 5, 9]
```

List comprehensions provide a simpler, cleaner, and faster way of doing the same thing

```
[1, 5, 9]
```

They can be used for fast computations as well. Here y serves as a dummy to access the elements in $diagLC$.

```
[1, 25, 81]
```

Obtaining a column is not as simple, but a list comprehension makes it quite straightforward:

```
[2, 5, 8]
```

```
[2, 5, 8]
```

Suppose you have a list of numbers and you want to extract all the elements of the list that are divisible by three. A slightly fancier list comprehension accomplishes the task quite simply and demonstrates a new feature:

```
[-3, 27, -9]
```

Exercise

Write a program to calculate the factorial of a positive integer input by the user. Recall that the factorial function is given by x! = x(x − 1)(x − 2)…(2)(1) so that 1! = 1, 2! = 2, 3! = 6, 4! = 24, …

- Write the factorial function using a Python while loop.
- Write the factorial function using a Python for loop.

Check your programs to make sure they work for 1, 2, 3, 5, and beyond, but especially for the first 5 integers.

3.4 Functions

Jump to top

3.4.1 User-defined functions

Every function definition begins with the word def followed by the name you want to give to the function, sinc in this case, then a list of arguments enclosed in parentheses, and finally terminated with a colon. In this case there is only one argument, x, but in general there can be as many arguments as you want, including no arguments at all. For the moment, we will consider just the case of a single argument. The indented block of code following the first line defines what the function does. In this case, the first line calculates $sinc(x) = \sin(x)/x$ and sets it equal to y. The return statement of the last line tells Python to return the value of y to the user.

Exercise

The above defined function delivers a false value at a certain point. Find the error and correct the function appropriately.

The code for *sinc(x)* works just fine when the argument is a single number or a variable that represents a single number. However, if the argument is a NumPy array, we run into a problem, as illustrated below.

The if statement in Python is set up to evaluate the truth value of a single variable, not of multielement arrays. When Python is asked to evaluate the truth value for a multi-element array, it doesn't know what to do and therefore returns an error. An obvious way to handle this problem is to write the code so that it processes the array one element at a time, which you could do using a for loop, as illustrated below.

3.4.2 Faster Array Processing

While using loops to process arrays works just fine, it is usually not the best way to accomplish the task in Python. The reason is that loops in Python are executed rather slowly. To deal with this problem, the developers of NumPy introduced a number of functions designed to process arrays quickly and efficiently. For the present case, what we need is a conditional statement or function that can process arrays directly. The function we want is called where and it is a part of the NumPy library. There where function has the form

```
where(condition, output if True, output if False)
```

e first argument of the where function is a conditional statement involving an array. The where function applies the condition to the array element by element, and returns the second argument for those array elements for which the condition is True, and returns the third argument for those array elements that are False. We can apply it to the sinc(x) function as follows

The where function creates the array y and sets the elements of y equal to 1.0 where the corresponding elements of x are zero, and otherwise sets the corresponding elements to sin(x)/x. This code executes much faster, 25 to 100 times, depending on the size of the array, than the code using a for loop. Moreover, the new code is much simpler to write and read.

3.4.3 Functions with more than one input or output

Python functions can have any number of input arguments and can return any number of variables. For example, suppose you want a function that outputs n (x, y) coordinates around a circle of radius r centered at the point (x0 , y0). The inputs to the function would be r, x0, y0, and n. The outputs would be the n (x, y) coordinates. The following code implements this function.

This function has four inputs and two outputs. In this case, the four inputs are simple numeric variables and the two outputs are NumPy arrays. In general, the inputs and outputs can be any combination of data types: arrays, lists, strings, etc. Of course, the body of the function must be written to be consistent with the prescribed data types. Functions can also return nothing to the calling program but just perform some task. For example, here is a program that clears the terminal screen

---

3.4.4 Positional and keyword arguments

It is often useful to have function arguments that have some default setting. This happens when you want an input to a function to have some standard value or setting most of the time, but you would like to reserve the possibility of giving it some value other than the default value.

The default values of the arguments x0, y0, and n are specified in the argument of the function definition in the def line. Arguments whose default values are specified in this manner are called keyword arguments, and they can be omitted from the function call if the user is content using those values.

```
(array([ 3.00000000e+00,  2.59807621e+00,  1.50000000e+00,  1.83697020e-16,
        -1.50000000e+00, -2.59807621e+00, -3.00000000e+00, -2.59807621e+00,
        -1.50000000e+00, -5.51091060e-16,  1.50000000e+00,  2.59807621e+00]),
 array([ 0.00000000e+00,  1.50000000e+00,  2.59807621e+00,  3.00000000e+00,
         2.59807621e+00,  1.50000000e+00,  3.67394040e-16, -1.50000000e+00,
        -2.59807621e+00, -3.00000000e+00, -2.59807621e+00, -1.50000000e+00]))
```

3.4.5 Variable number of arguments

While it may seem odd, it is sometimes useful to leave the number of arguments unspecified. A simple example is a function that computes the product of an arbitrary number of numbers:

The print("args…) statement in the function definition is not necessary, of course, but is put in to show that the argument args is a tuple inside the function. Here it used because one does not know ahead of time how many numbers are to be multiplied together.

The *args argument is also quite useful in another context: when passing the name of a function as an argument in another function. In many cases, the function name that is passed may have a number of parameters that must also be passed but aren't known ahead of time. If this all sounds a bit confusing—functions calling other functions—a concrete example will help you understand.

```
125
```

The order of the parameters is important. The function *test* uses x, the first argument of f1, as its principal argument, and then uses a and p, in the same order that they are defined in the function f1, to fill in the additional arguments—the parameters—of the function f1.

3.5 Numerical differentiation

We want to use our current knowledge of python and in particular also functions to start with some very useful thing for our physical modeling: * Numerical Differentiation *. What we want to calculate, is the derivative of a function $f(x)$ where the function values are given at certain positions x_i. Since we do not want to calculate the symbolic derivative, we have to get along with an numerical approximation. This can be obtained by looking at the definition of the derivative, i.e. the first derivative

$$f'(x) = \lim_{\delta x -> 0} \frac{f(x+\delta x) - f(x)}{\delta x} \tag{2}$$

If the function values are given at the positions x_i with $\delta x_i = x_{i+1} - x_i$, the an approximate value of the first derivative can be found from

$$f'(x_i) \approx \frac{f(x+\delta x) - f(x)}{\delta x} = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} \tag{3}$$

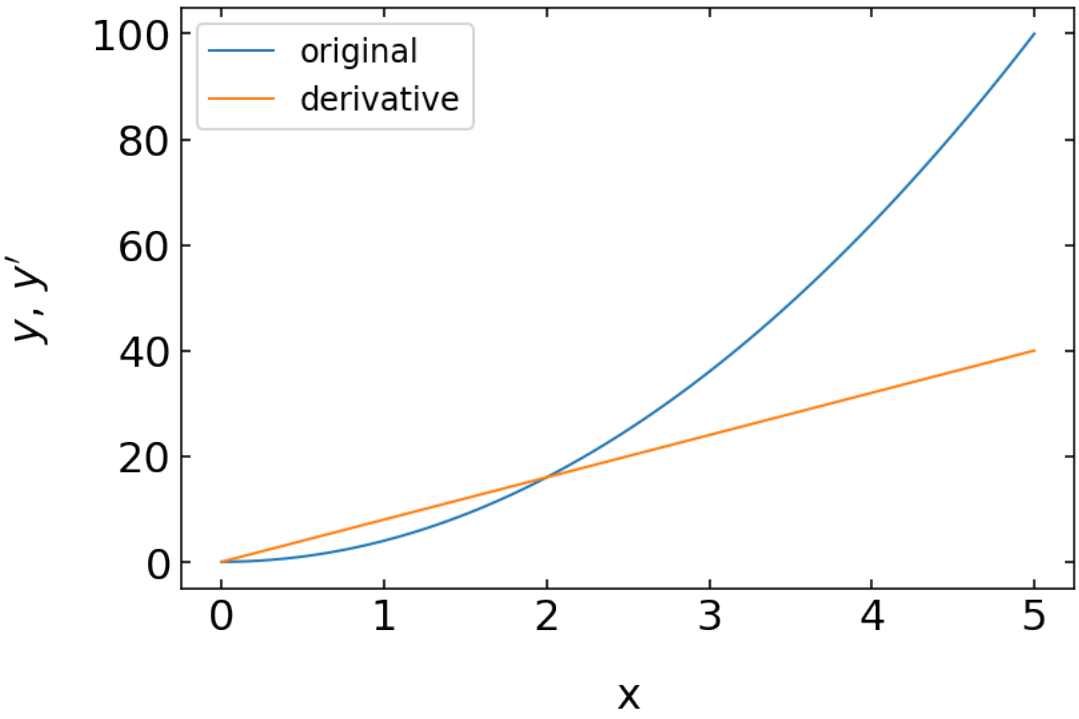
This already delivers a good approximation of the first derivative of a function as we see in the next examples. At frist we want to use our knowledge of multiple arguments and functions as an argument.

Suppose we have the following function that numerically computes the value of the derivative of an arbitrary function $f(x)$:

where our function shall be given by:

```
24.000001985768904
```

```
<matplotlib.legend.Legend at 0x152b290860>
```



3.5.1 More differentiation

While the above method is good for calculating the derivative of a function if this function is given at certain values, we will later often encounter the situation, that we want to solve a differential equation. As we will see later, this may involve the solution of the Schrödinger equation for an harmonic oscillator or a band structure.

$$H\Psi = (T + U)\Psi \tag{4}$$

where T and U are the kinetic and potential energies and H is the Hamilton operator. For solving these systems of equations numerically at certain positions (or times) in space (time), it is sometimes usefult to represent a discretized version of the derivatives in form of a matrix. Such a matrix can be efficiently generated with various python modules. Here we will use the *scipy.sparse* module, which is a module which generated sparse matrices.

If we consider the above finite difference formulas for a set of positions x_i, we can represent the derivative at these positions by matrix operation as well:

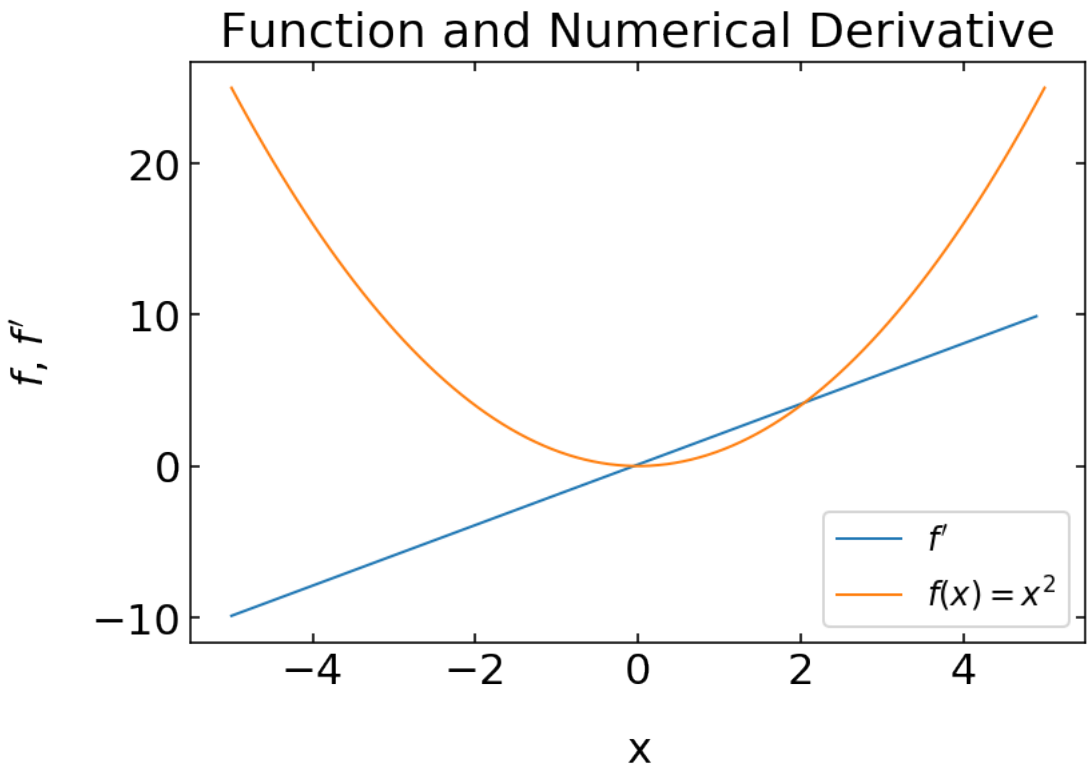
$$f' = \frac{1}{\delta x} \begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} (x_2 - x_1)/\delta x \\ (x_3 - x_2)/\delta x \\ (x_4 - x_3)/\delta x \\ (x_5 - x_4)/\delta x \\ (x_6 - x_5)/\delta x \\ (0 - x_6)/\delta x \end{bmatrix}$$

Each row of the matrix multiplied by the vector containing the positions is then containing the derivative of the function f at the position x_i and the resulting vector represents the deravitave in a certain position region.

We can construct such a matrix for the first derivative with the help of the *diags* function of the *scipy.sparse* module. The *diags* function uses a set of numbers, that shoud be distributed along the diagonal of the matrix. If you supply a list like in the example below, the numbers are distributed using the offsets as defined in the second list. The *shape* keyword defines the shape of the matrix. Try the example in the over next cell with the *.todense()* suffix. This converts the otherwise unreadable sparse output to a readable matrix form.

```
matrix([[-1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0., -1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0., -1.,  1.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0., -1.,  1.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0., -1.,  1.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0., -1.,  1.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0., -1.,  1.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0., -1.,  1.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0., -1.,  1.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0., -1.]])
```

If this matrix is now multiplied from the right with a vector of function values $f(x_i)$, you obtain the corresponding derivatives.



Similar to what we did above with the first derivative, we can also obtain a version for the second derivative according to the finite differences for the second derivative:

$$f'' \approx \frac{f(x+\delta x) - 2f(x) + f(x-\delta x)}{\delta x^2} \tag{5}$$

```
matrix([[-2.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 1., -2.,  1.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  1., -2.,  1.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  1., -2.,  1.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  1., -2.,  1.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  1., -2.,  1.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  1., -2.,  1.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  1., -2.,  1.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  1., -2.,  1.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  1., -2.]])
```

Equivalently, you may also find a general form for higher order finite difference which can be found here.