# 2_flowcontrol

May 28, 2024

## 1 Flow Control

File as PDF

### 1.1 Conditionals: 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.

**Note:** code blocks in if and loop statements

Python uses colons (`:`) to indicate that that a code block belonging to the `if` statement follows. The code block starting on the next line is idented with whitespace (often four spaces) to structure conditional statements. If the condition is True, the indented code block after the colon (`:`) is executed. The code block may contain several lines of code with identical identation:

```
number = 42

if number ==  42:
    print('This is the answer to  life,')
    print('the universe')
    print('and everything.')
```

Wrong identation will yield an `IndentationError` or a `SyntaxError`.

#### 1.1.1 If example

```
Please input a number:  124351

 'outside this block'
```

#### 1.1.2 If else example

```
Please input an integer:  4

4 is an even number.
```

### 1.1.3 If, elif, else example

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.

```
What is the coefficients a?  4
What is the coefficients b?  5
What is the coefficients c?  6

Solutions are complex
Finished!
```

### 1.1.4 Combining conditions

Multiple conditions may be combined with the logical operators `and`, `or`.

```
Both parts are true
```

```
At least one test is true
```

**Let's return to our example about making decisions on a rainy day .** Imagine that we consider not only the rain, but also whether or not it is windy. If it is windy or raining, we'll just stay at home. If it's not windy or raining, we can go out and enjoy the weather! Let's try to solve this problem using Python:

```
Just stay at home
```

## 1.2 Loops

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.

### 1.2.1 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.
```

When iterating through lists like in the example before, it is sometimes useful to have the index of the list element also available. Therefore the `enumerate` function is very useful.

```
0 . Max
 Arf, arf!
1 . Molly
 Arf, arf!
2 . Buster
 Arf, arf!
3 . Maggie
 Arf, arf!
4 . Lucy
 Arf, arf!
All done.
```

```
2500

 2500
```

### 1.2.2 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
```

### 1.2.3 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.

```
[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.

### 1.2.4 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]
```