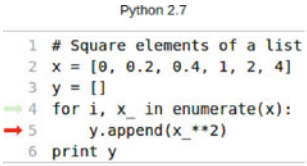
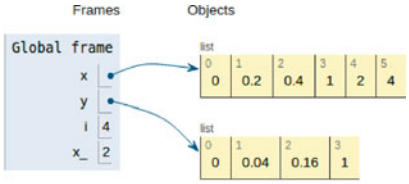
Basic Constructions

2





2.1 If Tests, Colon and Indentation

Very often in life, and in computer programs, the next action depends on the outcome of a question starting with "if". This gives the possibility to branch into different types of action depending on some criterion. Let us as usual focus on a specific example, which is the core of so-called random walk algorithms used in a wide range of branches in science and engineering, including materials manufacturing and brain research. The action is to move randomly to the north (N), east (E), south (S), or west (W) with the same probability. How can we implement such an action in life and in a computer program?

We need to randomly draw one out of four numbers to select the direction in which to move. A deck of cards can be used in practice for this purpose. Let the four suits correspond to the four directions: clubs to N, diamonds to E, hearts to S, and spades to W, for instance. We draw a card, perform the corresponding move, and repeat the process a large number of times. The resulting path is a typical realization of the path of a diffusing molecule.

In a computer program, we need to draw a random number, and depending on the number, update the coordinates of the point to be moved. There are many ways to draw random numbers and translate them into (e.g.) four random directions, but the technical details usually depend on the programming language. Our technique here is universal: we draw a random number in the interval $[0, 1)$ and let $[0, 0.25)$ correspond to N, $[0.25, 0.5)$ to E, $[0.5, 0.75)$ to S, and $[0.75, 1)$ to W. Let `x` and `y`

S. Linge, H.P. Langtangen, *Programming for Computations – Python*,
Texts in Computational Science and Engineering 15, DOI 10.1007/978-3-319-32428-9_2

hold the coordinates of a point and let `d` be the length of the move. A pseudo code (i.e., not "real" code, just a "sketch of the logic") then goes like

```
r = random number in [0,1)
if 0 <= r < 0.25
   move north: y = y + d
else if 0.25 <= r < 0.5
   move east:  x = x + d
else if 0.5 <= r < 0.75
   move south:  y = y - d
else if 0.75 <= r < 1
   move west:  x = x - d
```

Note the need for first asking about the value of `r` and then performing an action. If the answer to the "if" question is positive (true), we are done and can skip the next `else if` questions. If the answer is negative (false), we proceed with the next question. The last test if $0.75 \leq r < 1$ could also read just `else`, since we here cover all the remaining possible `r` values.

The exact code in Python reads

```
import random
r = random.random()        # random number in [0,1)
if 0 <= r < 0.25:
   # move north
   y = y + d
elif 0.25 <= r < 0.5:
   # move east
   x = x + d
elif 0.5 <= r < 0.75:
   # move south
   y = y - d
else:
   # move west
   x = x - d
```

We use `else` in the last test to cover the different types of syntax that is allowed. Python recognizes the reserved words `if`, `elif` (short for `else if`), and `else` and expects the code to be compatible with the rules of if tests:

- The test reads `if condition:`, `elif condition:`, or `else:`, where `condition` is a *boolean expression* that evaluates to `True` or `False`. Note the closing colon (easy to forget!).
- If `condition` is `True`, the following *indented* block of statements is executed and the remaining `elif` or `else` branches are skipped.
- If `condition` is `False`, the program flow jumps to the next `elif` or `else` branch.

The blocks after `if`, `elif`, or `else` may contain new if tests, if desired. Regarding colon and indent, you will see below that these are required in several other programming constructions as well.

Working with if tests requires mastering boolean expressions. Here are some basic boolean expressions involving the *logical operators* `==`, `!=`, `<`, `<=`, `>`, and

>=. Given the assignment to `temp`, you should go through each boolean expression below and determine if it is true or false.

```
temp = 21       # assign value to a variable
temp == 20  # temp equal to 20
temp != 20  # temp not equal to 20
temp <  20  # temp less than 20
temp >  20  # temp greater than 20
temp <= 20  # temp less than or equal to 20
temp >= 20  # temp greater than or equal to 20
```

2.2 Functions

Functions are widely used in programming and is a concept that needs to be mastered. In the simplest case, a function in a program is much like a mathematical function: some input number x is transformed to some output number. One example is the $\tanh^{-1}(x)$ function, called `atan` in computer code: it takes one real number as input and returns another number. Functions in Python are more general and can take a series of variables as input and return one or more variables, or simply nothing. The purpose of functions is two-fold:

1. to *group statements* into separate units of code lines that naturally belong together (a strategy which may dramatically ease the problem solving process), and/or
2. to *parameterize* a set of statements such that they can be written only once and easily be re-executed with variations.

Examples will be given to illustrate how functions can be written in various contexts.

If we modify the program `ball.py` from Sect. 1.2 slightly, and include a function, we could let this be a new program `ball_function.py` as

```
def y(t):
    v0 = 5                    # Initial velocity
    g = 9.81                  # Acceleration of gravity
    return v0*t - 0.5*g*t**2

time = 0.6       # Just pick one point in time
print y(time)
time = 0.9       # Pick another point in time
print y(time)
```

When Python reads and interprets this program from the top, it takes the code from the line with `def`, to the line with `return`, to be the definition of a function with the name `y` (note colon and indentation). The *return statement* of the function `y`, i.e.

```
return v0*t - 0.5*g*t**2
```

will be understood by Python as *first compute the expression, then send the result back (i.e., return) to where the function was called from*. Both `def` and `return` are reserved words. The function depends on `t`, i.e., one variable (or we say that it takes one *argument* or *input parameter*), the value of which must be provided when the function is called.

What actually happens when Python meets this code? The `def` line just tells Python that here is a function with name `y` and it has one argument `t`. Python does not look into the function at this stage (except that it checks the code for syntax errors). When Python later on meets the statement `print y(time)`, it recognizes a function call `y(time)` and recalls that there is a function `y` defined with one argument. The value of `time` is then transferred to the `y(t)` function such that `t = time` becomes the first action in the `y` function. Then Python executes one line at a time in the `y` function. In the final line, the arithmetic expression `v0*t - 0.5*g*t**2` is computed, resulting in a number, and this number (or more precisely, the Python object representing the number) *replaces* the call `y(time)` in the calling code such that the word `print` now precedes a number rather than a function call.

Python proceeds with the next line and sets `time` to a new value. The next `print` statement triggers a new call to `y(t)`, this time `t` is set to `0.9`, the computations are done line by line in the `y` function, and the returned result replaces `y(time)`. Instead of writing `print y(time)`, we could alternatively have stored the returned result from the `y` function in a variable,

```
h = y(time)
print h
```

Note that when a function contains `if-elif-else` constructions, `return` may be done from within any of the branches. This may be illustrated by the following function containing three `return` statements:

```
def check_sign(x):
    if x > 0:
        return 'x is positive'
    elif x < 0:
        return 'x is negative'
    else:
        return 'x is zero'
```

Remember that only one of the branches is executed for a single call on `check_sign`, so depending on the number `x`, the return may take place from any of the three return alternatives.

To return at the end or not

Programmers disagree whether it is a good idea to use `return` inside a function where you want, or if there should only be one single `return` statement *at the end of the function*. The authors of this book emphasize readable code and think that `return` can be useful in branches as in the example above when the function is short. For longer or more complicated functions, it might be better to have

one single `return` statement. Be prepared for critical comments if you return wherever you want...

An expression you will often encounter when dealing with programming, is *main program*, or that some code is *in main*. This is nothing particular to Python, and simply refers to that part of the program which is *outside* functions. However, note that the `def` line of functions is counted into main. So, in `ball_function.py` above, all statements outside the function `y` are in main, and also the line `def y(t):`.

A function may take no arguments, or many, in which case they are just listed within the parentheses (following the function name) and separated by a comma. Let us illustrate. Take a slight variation of the ball example and assume that the ball is not thrown straight up, but at an angle, so that two coordinates are needed to specify its position at any time. According to Newton's laws (when air resistance is negligible), the vertical position is given by $y(t) = v_{0y}t - 0.5gt^2$ and the horizontal position by $x(t) = v_{0x}t$. We can include both these expressions in a new version of our program that prints the position of the ball for chosen times. Assume we want to evaluate these expressions at two points in time, $t = 0.6s$ and $t = 0.9s$. We can pick some numbers for the initial velocity components `v0y` and `v0x`, name the program `ball_position_xy.py`, and write it for example as

```
def y(v0y, t):
    g = 9.81                    # Acceleration of gravity
    return v0y*t - 0.5*g*t**2

def x(v0x, t):
    return v0x*t

initial_velocity_x = 2.0
initial_velocity_y = 5.0

time = 0.6       # Just pick one point in time
print x(initial_velocity_x, time), y(initial_velocity_y, time)
time = 0.9       # ... Pick another point in time
print x(initial_velocity_x, time), y(initial_velocity_y, time)
```

Now we compute and print the two components for the position, for each of the two chosen points in time. Notice how each of the two functions now takes *two* arguments. Running the program gives the output

```
1.2  1.2342
1.8  0.52695
```

A function may also have no return value, in which case we simply drop the return statement, or it may return more than one value. For example, the two functions we just defined could alternatively have been written as one:

```
def xy(v0x, v0y, t):
    g = 9.81                        # acceleration of gravity
    return v0x*t, v0y*t - 0.5*g*t**2
```

Notice the two return values which are simply separated by a comma. When calling the function (and printing), arguments must appear in the same order as in the function definition. We would then write

```
print xy(initial_x_velocity, initial_y_velocity, time)
```

The two returned values from the function could alternatively have been assigned to variables, e.g., as

```
x_pos, y_pos = xy(initial_x_velocity, initial_y_velocity, time)
```

The variables `x_pos` and `y_pos` could then have been printed or used in other ways in the code.

There are possibilities for having a variable number of function input and output parameters (using `*args` and `**kwargs` constructions for the arguments). However, we do not go further into that topic here.

Variables that are defined inside a function, e.g., `g` in the last `xy` function, are *local variables*. This means they are only known inside the function. Therefore, if you had accidentally used `g` in some calculation outside the function, you would have got an error message. The variable `time` is defined outside the function and is therefore a *global variable*. It is known both outside and inside the function(s). If you define one global and one local variable, both with the same name, the function only sees the local one, so the global variable is not affected by what happens with the local variable of the same name.

The arguments named in the heading of a function definition are by rule local variables inside the function. If you want to change the value of a global variable inside a function, you need to declare the variable as global inside the function. That is, if the global variable was `x`, we would need to write `global x` inside the function definition before we let the function change it. After function execution, `x` would then have a changed value. One should strive to define variables mostly where they are needed and not everywhere.

Another very useful way of handling function parameters in Python, is by defining parameters as *keyword arguments*. This gives default values to parameters and allows more freedom in function calls, since the order and number of parameters may vary.

Let us illustrate the use of keyword arguments with the function `xy`. Assume we defined `xy` as

```
def xy(t, v0x=0, v0y=0):
    g = 9.81                        # acceleration of gravity
    return v0x*t, v0y*t - 0.5*g*t**2
```

Here, `t` is an *ordinary* or *positional argument*, whereas `v0x` and `v0y` are *keyword arguments* or *named arguments*. Generally, there can be many positional arguments and many keyword arguments, but the positional arguments must *always* be listed before the keyword arguments in function definition. Keyword arguments are given default values, as shown here with `v0x` and `v0y`, both having zero as de-

fault value. In a script, the function xy may now be called in many different ways. For example,

```
print xy(0.6)
```

would make xy perform the computations with `t = 0.6` and the default values (i.e zero) of v0x and v0y. The two numbers returned from xy are printed to the screen. If we wanted to use another initial value for v0y, we could, e.g., write

```
print xy(0.6,v0y=4.0)
```

which would make xy perform the calculations with `t = 0.6`, `v0x = 0` (i.e. the default value) and `v0y = 4.0`. When there are several positional arguments, they have to appear in the same order as defined in the function definition, unless we explicitly use the names of these also in the function call. With explicit name specification in the call, any order of parameters is acceptable. To illustrate, we could, e.g., call xy as

```
print xy(v0y=4.0, v0x=1.0, t=0.6)
```

In any programming language, it is a good habit to include a little explanation of what the function is doing, unless what is done by the function is obvious, e.g., when having only a few simple code lines. This explanation is called a *doc string*, which in Python should be placed just at the top of the function. This explanation is meant for a human who wants to understand the code, so it should say something about the purpose of the code and possibly explain the arguments and return values if needed. If we do that with our xy function from above, we may write the first lines of the function as

```
def xy(v0x, v0y, t):
    """Compute the x and y position of the ball at time t"""
```

Note that other functions may be called from within other functions, and function input parameters are not required to be numbers. Any object will do, e.g., string variables or other functions.

Functions are straightforwardly passed as arguments to other functions, as illustrated by the following script function_as_argument.py:

```
def sum_xy(x, y):
    return x + y

def prod_xy(x, y):
    return x*y

def treat_xy(f, x, y):
    return f(x, y)

x = 2;  y = 3
print treat_xy(sum_xy, x, y)
print treat_xy(prod_xy, x, y)
```

When run, this program first prints the sum of `x` and `y` (i.e., 5), and then it prints the product (i.e., 6). We see that `treat_xy` takes a function name as its first parameter. Inside `treat_xy`, that function is used to actually *call* the function that was given as input parameter. Therefore, as shown, we may call `treat_xy` with either `sum_xy` or `prod_xy`, depending on whether we want the sum or product of `x` and `y` to be calculated.

Functions may also be defined *within* other functions. It that case, they become *local functions*, or *nested functions*, known only to the function inside which they are defined. Functions defined in main are referred to as *global functions*. A nested function has full access to all variables in the *parent function*, i.e. the function within which it is defined.

Short functions can be defined in a compact way, using what is known as a *lambda function*:

```
f = lambda x, y: x + 2*y

# Equivalent
def f(x, y):
    return x + 2*y
```

The syntax consists of `lambda` followed by a series of arguments, colon, and some Python expression resulting in an object to be returned from the function. Lambda functions are particularly convenient as function arguments:

```
print treat_xy(lambda x, y: x*y, x, y)
```

Overhead of function calls

Function calls have the downside of slowing down program execution. Usually, it is a good thing to split a program into functions, but in very computing intensive parts, e.g., inside long loops, one must balance the convenience of calling a function and the computational efficiency of avoiding function calls. It is a good rule to develop a program using plenty of functions and then in a later optimization stage, when everything computes correctly, remove function calls that are quantified to slow down the code.

Here is a little example in IPython where we calculate the CPU time for doing array computations with and without a helper function:

```
In [1]: import numpy as np

In [2]: a = np.zeros(1000000)

In [3]: def add(a, b):
   ...:     return a + b
   ...:

In [4]: %timeit for i in range(len(a)): a[i] = add(i, i+1)
The slowest run took 16.01 times longer than the fastest.
This could mean that an intermediate result is being cached
1 loops, best of 3: 178 ms per loop

In [5]: %timeit for i in range(len(a)): a[i] = i + (i+1)
10 loops, best of 3: 109 ms per loop
```

We notice that there is some overhead in function calls. The impact of the overhead reduces quickly with the amount of computational work inside the function.

2.3 For Loops

Many computations are repetitive by nature and programming languages have certain *loop structures* to deal with this. Here we will present what is referred to as a *for loop* (another kind of loop is a *while* loop, to be presented afterwards). Assume you want to calculate the square of each integer from 3 to 7. This could be done with the following two-line program.

```
for i in [3, 4, 5, 6, 7]:
    print i**2
```

Note the colon and indentation again!

What happens when Python interprets your code here? First of all, the word `for` is a reserved word signalling to Python that a `for` loop is wanted. Python then sticks to the rules covering such constructions and understands that, in the present example, the loop should run 5 successive times (i.e., 5 *iterations* should be done), letting the variable `i` take on the numbers $3, 4, 5, 6, 7$ in turn. During each iteration, the statement inside the loop (i.e. `print i**2`) is carried out. After each iteration, `i` is automatically (behind the scene) updated. When the last number is reached, the last iteration is performed and the loop is finished. When executed, the program will therefore print out $9, 16, 25, 36$ and 49. The variable `i` is often referred to as a *loop index*, and its name (here `i`) is a choice of the programmer.

Note that, had there been several statements within the loop, they would all be executed with the same value of `i` (before `i` changed in the next iteration). Make sure you understand how program execution flows here, it is important.

In Python, integer values specified for the loop variable are often produced by the built-in function `range`. The function `range` may be called in different ways, that either explicitly, or implicitly, specify the start, stop and step (i.e., change) of the loop variable. Generally, a call to `range` reads

```
range(start, stop, step)
```

This call makes `range` return the integers from (and including) `start`, up to (but excluding!) `stop`, in steps of `step`. Note here that `stop-1` is the last integer included. With `range`, the previous example would rather read

```
for i in range(3, 8, 1):
    print i**2
```

By default, if `range` is called with only two parameters, these are taken to be `start` and `stop`, in which case a step of 1 is understood. If only a single parameter is used in the call to `range`, this parameter is taken to be `stop`. The default step of

1 is then used (combined with the starting at 0). Thus, calling `range`, for example, as

```
range(6)
```

would return the integers `0, 1, 2, 3, 4, 5`.

Note that decreasing integers may be produced by letting `start > stop` combined with a negative step. This makes it easy to, e.g., traverse arrays in either direction.

Let us modify `ball_plot.py` from Sect. 1.4 to illustrate how useful `for` loops are if you need to traverse arrays. In that example we computed the height of the ball at every milli-second during the first second of its (vertical) flight and plotted the height versus time.

Assume we want to find the maximum height during that time, how can we do it with a computer program? One alternative may be to compute all the thousand heights, store them in an array, and then run through the array to pick out the maximum. The program, named `ball_max_height.py`, may look as follows.

```
import matplotlib.pyplot as plt

v0 = 5                    # Initial velocity
g = 9.81                  # Acceleration of gravity
t = linspace(0, 1, 1000)  # 1000 points in time interval
y = v0*t - 0.5*g*t**2     # Generate all heights

# At this point, the array y with all the heights is ready.
# Now we need to find the largest value within y.

largest_height = y[0]          # Starting value for search
for i in range(1, 1000):
    if y[i] > largest_height:
        largest_height = y[i]

print "The largest height achieved was %f m" % (largest_height)

# We might also like to plot the path again just to compare
plt.plot(t,y)
plt.xlabel('Time (s)')
plt.ylabel('Height (m)')
plt.show()
```

There is nothing new here, except the `for` loop construction, so let us look at it in more detail. As explained above, Python understands that a `for` loop is desired when it sees the word `for`. The `range()` function will produce integers from, and including, 1, up to, and including, 999, i.e. $1000 - 1$. The value in `y[0]` is used as the *preliminary* largest height, so that, e.g., the very first check that is made is testing whether `y[1]` is larger than this height. If so, `y[1]` is stored as the largest height. The for loop then updates `i` to 2, and continues to check `y[2]`, and so on. Each time we find a larger number, we store it. When finished, `largest_height`

will contain the largest number from the array `y`. When you run the program, you get

```
The largest height achieved was 1.274210 m
```

which compares favorably to the plot that pops up.

To implement the traversing of arrays with loops and indices, is sometimes challenging to get right. You need to understand the start, stop and step length choices for an index, and also how the index should enter expressions inside the loop. At the same time, however, it is something that programmers do often, so it is important to develop the right skills on these matters.

Having one loop inside another, referred to as a *double loop*, is sometimes useful, e.g., when doing linear algebra. Say we want to find the maximum among the numbers stored in a 4×4 matrix `A`. The code fragment could look like

```
largest_number = A[0][0]

for i in range(4):
    for j in range(4):
        if A[i][j] > largest_number:
            largest_number = A[i][j]
```

Here, all the `j` indices (`0 - 3`) will be covered for *each* value of index `i`. First, `i` stays fixed at `i = 0`, while `j` runs over all its indices. Then, `i` stays fixed at `i = 1` while `j` runs over all its indices again, and so on. Sketch `A` on a piece of paper and follow the first few loop iterations by hand, then you will realize how the double loop construction works. Using two loops is just a special case of using *multiple* or *nested loops*, and utilizing more than two loops is just a straightforward extension of what was shown here. Note, however, that the loop index *name* in multiple loops must be unique to each of the nested loops. Note also that each nested loop may have as many code lines as desired, both before and after the next inner loop.

The vectorized computation of heights that we did in `ball_plot.py` (Sect. 1.4) could alternatively have been done by traversing the time array (`t`) and, for each `t` element, computing the height according to the formula $y = v_0 t - \frac{1}{2} g t^2$. However, it is important to know that vectorization goes much quicker. So when speed is important, vectorization is valuable.

Use loops to compute sums

One important use of loops, is to calculate sums. As a simple example, assume some variable x given by the mathematical expression

$$x = \sum_{i=1}^{N} 2 \cdot i,$$

i.e., summing up the N first even numbers. For some given N, say $N = 5$, x would typically be computed in a computer program as: