Ch.3: Functions and branching

Joakim Sundnes[1,2]

[1]Simula Research Laboratory
[2]University of Oslo, Dept. of Informatics

Aug 30, 2019

This chapter introduces two fundamental programming concepts; *functions* and *branching*. We are used to *functions* from mathematics, were we typically define a function $f(x)$ as some mathematical expression of x, and then we can evaluate the function for different values of x, plot the curve $y = f(x)$, solve equations of the kind $f(x) = 0$ etc. A similar function concept exists in programming, where a function is a piece of code that takes some variables as input, does some processing, and gives some output in return. The function concept in programming is more general than in mathematics, and is not restricted to numbers or mathematical expressions, but the general idea is exactly the same.

Branching, or if-tests, is another fundamental concept that exists in all common programming languages. The idea is that we make decisions in the code based on the value of some Boolean expression or variable. If the expression evaluates to True we perform one set operations, and if it is False we do something different. Such tests are essential for controlling the flow of a computer program.

1 Programming with functions

We have already used a number of Python functions in the previous chapters. The mathematical functions from the `math` module are essentially the same as we are used to from mathematics or from pushing buttons on a calculator:

```
from math import *
y = sin(x)*log(x)
```

We have also used a few non-mathematical functions, such as `len` and `range`

```
n = len(somelist)
for i in range(5, n, 2):
    (...)
```

and we have used methods that were bound to specific objects, and accessed with the dot syntax, for instance `append` to add elements to a list:

```
C = [5, 10, 40, 45]
C.append(50)
```

This latter type of function is quite special since it is bound to a specific object, and operates directly on that object (`C.append` will change `C`). These bound functions are also referred to as *methods*, and will be considered in more detail in Chapter 7. In the present chapter we will focus on regular, un-bound, functions In Python, such functions give easy access to already existing program code written by others (such as `sin(x)`). There is plenty of such code in Python, and nearly all programs involve importing one or more modules and using pre-defined functions from these modules. A good thing about functions is that we can use them without knowing anything about how they are implemented. All we need to know is what goes in and what comes out, the function can be used as a black box.

Functions also provide a way of reusing code we have written ourselves, either in previous projects or as part of the current code, and this is the main focus of this chapter. Functions let us delegate responsibilities and split a program into smaller tasks, which is essential for solving all problems of some complexity. As we shall see later in this chapter, splitting a program into smaller functions is also convenient for testing and verifying that a program works as it should. We can write small pieces of code that test all the individual functions and ensures that work correctly, before putting the functions together to a complete program. If such tests are done properly, we can have some confidence that our main program works as expected. We will come back to this topic towards the end of the chapter.

So how do we write a function in Python? Starting with a simple example, consider the mathematical function considered previously

$$F(C) = \frac{9}{5}C + 32.$$

We can implement this in Python as follows:

```
def F(C):
    return (9.0/5)*C + 32
```

This code has just two lines, but contains a few new concepts that are worth taking note of. Starting with the first line, `def F(C):`, this is called the function *header*, defines the function's interface, i.e. how it is to be used. All function definitions in Python start with the word `def`, which is simply how we tell Python that the following code defines a function. After `def` comes the name of the function, followed by parentheses containing the function's *arguments* (sometimes called parameters). This simple function takes a single argument, but we can easily define functions that take multiple arguments by separating the arguments with commas. The partheses need to be there even if we won't want the function to take any arguments, in that case we would just leave it empty. The line following the function header is the *function body*, which is usually multiple lines but in this case just one. The lines defining the function body are indented, which serves the same purpose as with the loops in the previous chapter. The indentation is the way we specify which code lines that belong inside the function, to the function body. The line inside the function

also starts with the keyword `return`, which is new to us. The return statement specifies the output returned by the function. It is important not to confuse this return statement with the print statements we have used previously. The use of `print` will simply output something to the screen, while `return` makes the function provide an output, which one can think of as a variable being passed back to the code that called the function. Consider for instance the example `n = len(somelist)` used in the previous chapter, where `len` returned an integer which was assigned to a variable `n`.

Another important thing to note about the code above is that it does not do much. In fact, a function definition does essentially nothing before it is *called*.[1] The analogue to the function definition in mathematics is to simply write down a function $f(x)$ as a mathematical expression. This defines the function, but we don't get any output until we start evaluating it for some specific values of x. In programming, we say that we *call* the function when we use it. When programming with functions, it is common to refer to the *main program*, basically being every line of code that is not inside a function. When running the program, only the statements in the main program are executed. Code inside function definitions is not run until we put a call to the function in the main program. We have already called pre-defined functions like `sin, len`, etc, in previous chapters, and a function we have written ourselves is called in exactly the same way:

```
def F(C):
    return (9.0/5)*C + 32

a = 10
F1 = F(a)                                # call
temp = F(15.5)                           # call
print(F(a+1))                            # call
sum_temp = F(10) + F(20)                 # two calls
Fdegrees = [F(C) for C in [0, 20, 40]] # multiple calls
```

The call `F(C)`, for some argument `C`, returns a `float` object, which essentially means that `F(C)` is replaced by this `float` object. We can therefore make the call `F(C)` everywhere a `float` can be used.

Note that, unlike many other programming languages, Python does not require us to specify the type of function arguments. Judging from the function header only, the argument to `F(C)` above could be any kind of variable. However, by looking at how `C` is used inside the function, we can tell that it must be a number (int or float). If we write complex functions where the types of the arguments are not obvious, we may insert a comment right after the header, a so-called *doc string*, to tell users what the arguments should be. More about this below.

[1] This is not entirely true, as defining the function creates a function object, whcih we can see by defining a dummy function in the Python shell and then calling `dir()` to get a list of defined variables. However, no visible output is produced until we actually call the function, and forgetting to call the function is a common mistake when starting to program with functions.

Functions can have as many arguments as you like. Just as in mathematics, we can define Python functions with more than one argument. Say we want to make a Python function of the physics formula we looked at earlier;

$$y(t) = v_0 t - \frac{1}{2} g t^2.$$

Here, we could define a function with only t as argument, as indicated by the use of $y(t)$ in the definition. But what if we want to evaluate the formula for many different values of v_0? It could be useful to include v_0 as an argument as well. The function definition then becomes

```
def yfunc(t, v0):
    g = 9.81
    return v0*t - 0.5*g*t**2

# sample calls:
y = yfunc(0.1, 6)
y = yfunc(0.1, v0=6)
y = yfunc(t=0.1, v0=6)
y = yfunc(v0=6, t=0.1)
```

In addition to the two function arguments, we have here defined `g` as a *local variable* inside the function. Inside the function, there is no distinction between such a local variable and the arguments passed to the function. The arguments also become local variables, and are used in exactly the same way as any variable we define inside the function. However, there is an important distinction between *local* and *global* variables. Variables defined in the main program become global variables, while variables define inside functions are local.The local variables are only defined and available inside the function, while global variables can be used everywhere in the program. If we tried to access `v0, t`, or `g` (for instance by `print(v0)`) from outside the function, we will simply get an error message stating that the variable is not defined.

Notice also the alternative ways of calling the function. We can either specify the argument names in the call, as in `t=0.1`, or simply pass the values. If we specify the names the order of the arguments becomes arbitrary, as in the last call above. Arguments that are passed without specifying the name are called *positional arguments*, because their position in the argument list determines which variable they are assigned to. Arguments that are passed including the name are called *keyword arguments*. The keyword arguments need to match the definition of the function, i.e., calling the function above with `yfunc(t=0.6,v=5)` would cause an error message because `v` is not defined as an argument to the function. Another rule worth taking note of is that a positional argument cannot follow a keyword argument; a call such as `yfunc(0.1, v0=6)` is fine, but `yfunc(t=0.1,6)` is not and the program will stop with an error message. This rule is quite logical, since a complete mix of positional and keyword arguments would make the call very confusing.

The distinction between local and global variables is generally important in programming, and may be confusing. As stated above, the arguments passed to a function, as well as variables we define inside the function, become local

variables. These variables behave exactly like we are used to inside the function, but are not visible outside it. The potential source of confusion is that global variables are also accessible inside a function, just as everywhere else in the code. We could have put the definition `g=9.81` above outside the function, anywhere before the first call to `yfunc`, and the code would still work. However, sometimes one may define local and global variables with the same name, such as

```
g = 10.0

def yfunc(t, v0):
    g = 9.81
    return v0*t - 0.5*g*t**2

print(yfunc(0.1, 6))
```

Which value of `g` is used in the function call here? The answer is that local variable names always takes precedence over the global names. When the mathematical formula is encountered in the code above, Python will look for values of the variables `v0, g`, and `t` that appear in the formula. First, the so-called *local namespace* is searched, i.e. Python looks for local variables with the given names. If local variables are found, as in the example above, these values are used. If some variables are not found in local namespace, Python will move to the *global namespace*, and look for global variables that match the given names. If a variable with the right name is found among the global variables, i.e. it has been defined in the main program, then the corresponding value is used. If no global variable with the right name is found there are no more places to search and the program ends with an error message. This sequence of searching for variables is quite natural and logical, but is still a potential source of confusion and programming errors. Additional confusion may occur if we attempt to change a global variable inside a function. Consider the code

```
v0 = 5

def yfunc(t):
    g = 9.81
    v0 = 7.5
    print(v0)
    return v0*t - 0.5*g*t**2

print(yfunc(0.6))
print(v0)
```

As revealed by the print statements, `v0` is set to 7.5 inside the function, but the global `v0` remains unchanged after the function has been called. In fact, since the line `v0 = 7.5` occurs inside a function, Python will treat this as the defnition of a new local variable, rather than trying to change a global one. As such, we define a new local `v0` with value 7.5, while there is still another `v0` defined in the global namespace. After the function call has ended, the local variable no longer exists (in programming terms, it *goes out of scope*), while the global `v0` is still there and has its original value. If we actually want to change a global variable inside a function, we must explicitly state this using the keyword `global`. Consider the following code, which is nearly identical to the one above:

```
v0 = 5

def yfunc(t):
    g = 9.81
    global v0 = 7.5
    print(v0)
    return v0*t - 0.5*g*t**2

print(yfunc(0.6))
print(v0)
```

We see that in this case the global `v0` is in fact changed. The keyword `global` tells Python that we do in fact want to change a global variable, and not define a new local one. As a general rule, to avoid confusion of this kind, one should minimize the use of global variables. In particular, all variables used inside a function should either be local variables or passed as arguments to the function. Similarly, if we want the function to change a global variable then this variable should be returned from the function, instead of using the keyword `global`. It is difficult to think of a single example where using `global` is the best solution, and in practice, it should never be used. If we actually wanted the function above to change the global `v0`, this would be a better way to do it

```
v0 = 5

def yfunc(t):
    g = 9.81
    v0 = 7.5
    y = v0*t - 0.5*g*t**2
    return y, v0

y,v0 = yfunc(0.6)
print(y)
print(v0)
```

Note that we here return two values from the function, separated by commas just as in the list of arguments. Notice also the line with the function call, where we assign the returned values to the global variables `y,v0`, using commas to separate the values also on the left hand side of the assignment operator. Although this simple example may not be the most practically relevant one, there are many cases where it is useful for a function call to change a global variable. In those cases it should always be done in this way, by returning the variable from the function and assigning this return value to the global variable, rather than using the `global` keyword inside the function. Sticking to this rules ensures that each function is a self-contained entity, with a clearly defined interface to the rest of the code through the list of arguments and return values.

Multiple return values are returned as a tuple. To consider a more practically relevant example of multiple return values, say we want to compute both the function and its derivative, $y(t)$ and $y'(t) = v_0 - gt$:

```
def yfunc(t, v0):
    g = 9.81
    y = v0*t - 0.5*g*t**2
```

```
    dydt = v0 - g*t
    return y, dydt

# call:
position, velocity = yfunc(0.6, 3)
```

As above, the return arguments are separated by comma, and we assign the values to the two global variables `position` and `velocity`, also separated with commas. When a function returns multiple values like this, it actually returns a tuple, the non-mutable list type defined in the previous chapter. So we could replace the call above with something like this

```
pos_vel = yfunc(0.6,3)
print(pos_vel)
print(type(pos_vel))
```

We see that the function returns a tuple with two elements. In the previous call, when we included a comma-separated list of variable names on the left hand side (i.e. `position, velocity`) Python will *unpack* the elements in the tuple into the corresponding variables. For this to work the number of variables must match the length of the tuple, otherwise we get an error message stating that there are too many or not enough values to unpack.

A function can return any number arguments, separated by commas exactly as above. Here we have three

```
def f(x):
    return x, x**2, x**4

s = f(2)
print(type(s), s)
x, x2, x4 = s
```

Notice the last line, where a tuple of length 3 is unpacked into three individual variables.

1.1 Example: Function for computing a sum

To look at a more relevant function example, of a kind that will occur frequently in this course, consider the sum

$$L(x;n) = \sum_{i=1}^{n} \frac{1}{i} \left(\frac{x}{1+x} \right)^i,$$

which is an approximation to $\ln(1+x)$ for a finite n and $x \geq 1$. The corresponding Python function for $L(x;n)$ looks like

```
def L(x, n):
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1+x))**i
    return s

#example use
x = 5
from math import log as ln
print(L(x, 10), L(x, 100), ln(1+x))
```

The output from the print statement indicates that the approximation improves as the number of terms n is increased, as usual for such approximating series. For many purposes, it would be useful if the function returned the first neglected term in the sum, since this gives an indication of the error in the approximation, as well as the actual error $\ln(1+x) - L(x;n)$. We can make the function return these values by including them in the return statement, just like we did above:

```
def L2(x, n):
    x = float(x)
    s = 0
    for i in range(1, n+1):
        s += (1.0/i)*(x/(1+x))**i
    value_of_sum = s
    first_neglected_term = (1.0/(n+1))*(x/(1+x))**(n+1)
    from math import log
    exact_error = log(1+x) - value_of_sum
    return value_of_sum, first_neglected_term, exact_error

# typical call:
x = 1.2; n = 100
value, approximate_error, exact_error = L2(x, n)
```

A function does not need a return statement. All the functions considered so far have included a return statement. While this will be the case for most of functions we write in this course, there will be exceptions, and a function does not need to have a return statement. For instance, some functions only serve the purpose of printing information to the screen, as in this course, a function

```
def somefunc(obj):
    print(obj)

return_value = somefunc(3.4)
```

Here, the last line does not make much sense, although it is actually valid Python code and will run without errors. We do not return anything from `somefunc`, how can we then call the function and assign the result to a variable? The answer is that if we do not include a return statement in a function, Python will automatically return a variable with value `None`. So the variable `return_value` variable in this case will be `None`, which is not very useful, but serves to illustrate the behavior of a function with no return statement. Most functions we will write in this course will either return some variables, or print or plot something to the screen. One typical use of a function without a return value is to print information in a table-like format to the screen. This is useful in many contexts, including studying the convergence of series approximations like the one above. The following function will call the `L2(x,n)` function above, and uses a for-loop to print relevant information in a nicely formatted table.

```
def table(x):
    print(f'x={x}, ln(1+x)={log(1+x)}')
    for n in [1, 2, 10, 100, 500]:
        value, next, error = L2(x, n)
        print('n={n:4d} approx: {value:6.2f}, next term: {next:6.2f} error: {error:6.2f}')

print(table(10))
```

```
x=10, ln(1+x)=2.3979
n=1    0.909091    (next term: 4.13e-01  error: 1.49e+00)
n=2    1.32231     (next term: 2.50e-01  error: 1.08e+00)
n=10   2.17907     (next term: 3.19e-02  error: 2.19e-01)
n=100  2.39789     (next term: 6.53e-07  error: 6.59e-06)
n=500  2.3979      (next term: 3.65e-24  error: 6.22e-15)
```

There is no need to return anything from this function, since the entire purpose is to print information to the screen.

Default arguments can be used to simplify function calls. When we used the `range`-function in the previous chapter we saw that we could vary the number of arguments in the function call from one to three, and the non-specified arguments would get default values. We can achieve the same funtionality in our own functions, by defining *default arguments* in the function definition:

```
def somefunc(arg1, arg2, kwarg1=True, kwarg2=0):
    print(arg1, arg2, kwarg1, kwarg2)
```

A function defined in this way can be called with two, three, or four arguments. The first two have no default value and therefore must be included in the call, while the last two are optional and will be set to the default value if not specified in the call. In texts on Python programming, *default arguments* is often referred to as keyword arguments, although these terms do not mean exactly the same thing. They are, however, closely related, and this is why the terms are sometimes interchanged. Just as we cannot have keyword arguments preceding positional arguments in a function call, we cannot have default arguments preceding non-default arguments in the function header. The following code demonstrates some use of the alternative function calls for a completely useless but illustrative function. Testing a simple function like this, which does nothing but print out the argument values, is a good way to understand the implications of default arguments and the resulting flexibility in argument lists.

```
>>> def somefunc(arg1, arg2, kwarg1=True, kwarg2=0):
>>>     print(arg1, arg2, kwarg1, kwarg2)

>>> somefunc('Hello', [1,2])   # drop kwarg1 and kwarg2
Hello [1, 2] True 0            # default values are used

>>> somefunc('Hello', [1,2], 'Hi')
Hello [1, 2] Hi 0              # kwarg2 has default value

>>> somefunc('Hello', [1,2], 'Hi', 6)
Hello [1, 2] Hi 0              # kwarg2 has default value

>>> somefunc('Hello', [1,2], kwarg2='Hi') #kwarg2
Hello [1, 2] True Hi           # kwarg1 has default value

>>> somefunc('Hello', [1,2], kwarg2='Hi', kwarg1=6)
Hello [1, 2] 6 Hi              # specify all args
```

Using what we now know about default argument, we may improve the function considered above, which implemented the formula

$$y(t) = v_0 t - \frac{1}{2} g t^2.$$

Here, it is natural to think of t as the primary argument to the function, which should always be provided, while it could be natural to define v_0 and maybe also g as default arguments. The function definiton in Python could read

```
def yfunc(t, v0=5, g=9.81):
    return v0*t-0.5*g*t**2

#example calls:
y1 = yfunc(0.2)
y2 = yfunc(0.2,v0=7.5)
y3 = yfunc(0.2,7.5,10.0)
```

1.2 Documentation of Python functions

An important Python convention is to document the purpose of a function, its arguments, and its return values in a *doc string* - a (triple-quoted) string written right after the function header. The doc string can be long or short, depending on the complexity of the function and its inputs and outputs. The following two examples show how it can be used:

```
def C2F(C):
    """Convert Celsius degrees (C) to Fahrenheit."""
    return (9.0/5)*C + 32

def line(x0, y0, x1, y1):
    """
    Compute the coefficients a and b in the mathematical
    expression for a straight line y = a*x + b that goes
    through two points (x0, y0) and (x1, y1).

    x0, y0: a point on the line (floats).
    x1, y1: another point on the line (floats).
    return: a, b (floats) for the line (y=a*x+b).
    """
    a = (y1 - y0)/(x1 - x0)
    b = y0 - a*x0
    return a, b
```

Doc strings do not take much time to write, and are very useful for others who want to use the function. Being a widely accepted convention in the Python community, doc strings are also used by various tools for automatically generating nicely formatted software documentation. Much of the online documentation of Python libraries and modules is automatically generated from doc-strings included in the code.

1.3 Python functions as arguments to Python functions

Arguments to Python functions can be any Python object, including another function. This functionality is quite useful for many scientific applications, where we need to define mathematical functions that operate on or make use of other mathematical functions. For instance, we can easily write Python functions for numerical approximations of integrals $\int_a^b f(x)dx$, derivatives $f'(x)$, and roots $f(x) = 0$. For such functions to be general and useful, they should work with an

arbitrary function $f(x)$. This is most conveniently accomplished by passing a Python function `f(x)` as an argument to the function.

Consider the example of approximating the second derivative $f''(x)$ by centered finite differences,

$$f''(x) \approx \frac{f(x-h) - 2f(x) + f(x+h)}{h^2}.$$

The corresponding Python function looks like

```
def diff2(f, x, h=1E-6):
    r = (f(x-h) - 2*f(x) + f(x+h))/float(h*h)
    return r
```

We see that the function `f` is passed to the function just like any other argument, and called as a regular function inside `diff2`. Of course, for this to work we need to actually send a callable function as the first argument to `diff2`. If we send something else, like a number or a string, the code will stop with an error when it tries to make the call `f(x-h)` in the next line. Such potential errors are part of the price we pay for Python's flexibility. We can pass any argument to a function, but the object we pass must be possible to use as intended inside the function. As noted above, for more complex functions it is useful to include a doc string that specifies what types of arguments the function expects.

Lambda functions for compact inline function definitions. In order to use the function `diff2` above, the standard way would be to define our `f(x)` as a Python function, and then pass it as an argument to `diff2`. The following code shows an example.

```
def f(x):
    return x**2 - 1

df2 = diff2(f,1.5)
print(df2)
```

A concept known as a *lambda function* offers a compact way to define functions, which can be convenient for the present application. Using the keyword `lambda`, we can define our `f` on a single line:

```
f = lambda x: x**2 - 1
```

More generally, a lambda function defined by

```
somefunc = lambda a1, a2, ...: some_expression
```

is equivalent to

```
def somefunc(a1, a2, ...):
    return some_expression
```

One may ask if anything is really gained here, and whether it is useful to introduce a new concept just to reduce a function definition from two lines to one line. One answer is that the lambda function definition can be put directly into the argument list of the other function. Instead of first defining `f(x)` and then passing it as an argumen, as in the code above, we combine these tasks into one line:

```
df2 = diff2(lambda x: x**2-1,1.5)
print(df2)
```

Using lambda functions in this way can be quite convenient in cases where we need to pass a simple mathematical expression as an argument to a Python function. We save some typing, and may also improve the readability of the code.

2 If-tests for branching the program flow

In computer programs we commonly want to perform different actions depending on a condition. As usual, we can find a similar concept in mathematics, which should be familiar to most readers of this book. Consider a function defined in a piecewise manner, for instance

$$f(x) = \begin{cases} \sin x, & 0 \leq x \leq \pi \\ 0, & \text{otherwise} \end{cases}$$

The Python implementation of such a function needs to test on the value of the input x, and choose to return either 0 or $\sin(x)$ depending on the outcome. Such a decision taking place in the program code is called *branching* and is obtained using an if-test, or more generally an if-else block. The code looks like

```
from math import sin, pi

def f(x):
    if 0 <= x <= pi:
        return sin(x)
    else:
        return 0

print(f(0.5))
print(f(5*pi))
```

The new item here is the if-else block. An if-test is simply constructed by the keyword `if` followed by a Boolean variable or expression, and then a block of code which is to be executed if the condition is true. When the if-test is reached in the function above, the Boolean condition is tested, just as for the while loops of the previous chapter. If it is True, the following block of indented code is executed (here just one line). If not, the indented code block after `else` is executed. One may also notice that unlike the functions seen so far, this function has two return statements. This is perfectly valid, and is quite common in functions with if-tests. When a return statement is executed, the function call is over and any following lines in the function are simply ignored. Therefore, there is usually no point in having multiple return statements unless they are combined with some if-tests, since if the first one is always executed the others are enver reached.

In its simplest form, we may skip `else` and define only an if-test:

```
if condition:
    <block of statements, executed if condition is True>
```

```
<next line after if-block, always executed>
```

Here, whatever is inside the if-block is executed if `condition` is True, otherwise the program simply moves to the next line after the block. As we did above, we may add an else-block to ensure that exactly one of two code blocks is executed

```
if condition:
    <block of statements, executed if condition is True>
else:
    <block of statements, executed if condition is False>
```

For mathematical functions of the form considered above we usually want to include an else-block, since we want the function to return a meaningful value for all input arguments. Forgetting the else-block in the definition `f(x)` above would make the function return $\sin(x)$ (a `float`) for $0 \leq x \leq \pi$, and otherwise `None`, which is obviously not what we want. Finally, we may combine multiple if-else statements with different conditions

```
if condition1:
    <block of statements>
elif condition2:
    <block of statements>
elif condition3:
    <block of statements>
else:
    <block of statements>
<next statement>
```

Notice the keyword `elif`, short for *else if*, which ensures that that subsequent conditions are only tested if the preceding ones are False. The conditions are checked one by one, and as soon as one evaluates to True the corresponding block is executed and the program moves to the first statement after the else-block. The remaining conditions are not checked. If none of the conditions are True, the code inside the else-block is executed.

Multiple branching has useful applications in mathematics, since we often see piecewisely defined functions defined on multiple intervals. Consider for instance the piecewise linear function

$$N(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \leq x < 1 \\ 2 - x, & 1 \leq x < 2 \\ 0, & x \geq 2 \end{cases}.$$

which in Python is implemented with multiple if-else-branching

```
def N(x):
    if x < 0:
        return 0
    elif 0 <= x < 1:
        return x
    elif 1 <= x < 2:
        return 2 - x
    elif x >= 2:
        return 0
```

In later chapters we will see multiple examples of more general use of branching, not restricted to mathematics and piecewisely defined functions.

Inline if-tests for shorter code. The list comprehensions in Chapter 2, and the lambda functions above, offered more compact alternatives to the standard way of defining lists or functions. Not surprisingly, a similar concept exists for if-tests. A common use of if-else blocks is to assign a value to a variable, where the value depends on some condition, just as we did in the examples above. The general form looks like

```
if condition:
    variable = value1
else:
    variable = value2
```

This code can be replaced by the following one-line if-else block:

```
variable = (value1 if condition else value2)
```

With such compact

```
def f(x):
    return (sin(x) if 0 <= x <= pi else 0)
```

3 Test functions

In the discussion above we introduced the idea of writing tests to verify that functions work as intended. This approach to programming may be very effective, and although we spend some time writing the tests we often save much more time by the fact that we discover errors early, and can build our program from components that are known to work. The process is often referred to as *unit testing*, since each test verifies that a small unit of the program works as expected. Many programmers even take the approach one step further, and write the test before they write the actual function. This is often referred to as test-driven development, and is an increasingly popular method for software development.

The tests we write to test our functions will also be functions; a special type of function known as *test functions*. Writing good test functions, which test the functionality of our code in a reliable manner, can be quite challenging. However, the overall idea is very simple. For a given function, which often takes some arguments, we choose arguments where we can calculate the result of the function by hand. Inside the test function we will then simply call our function with the right arguments and compare the result returned by the function with the expected (hand-calculated) result. The following example illustrates how we can write a test function to test that the (very) simple function `double(x)` works as it should:

```
def double(x):              # some function
    return 2*x

def test_double():          # associated test function
```

```
"""Call double(x) to check that it works."""
x = 4                 # some chosen x value
expected = 8          # expected result from double(x)
computed = double(x)
success = computed == expected  # boolean value: test passed?
msg = f'computed {computed}, expected {expected}'
assert success, msg
```

In this code, the only Python keyword that we have not seen previously is `assert` which is used in place of `return` whenever we write a test function. Test functions should not return anything, so a regular return statement would not make sense. The only purpose of the test function is to compare the value returned by a function with the value we expect it to return, and write an error message if the two are different. This task is precisely what `assert` does. The assert keyword should always be followed by a condition, `success` in the code above, which is True if the test passes and False if it fails. The code above follows the typical recipe; we compare the the expected with the returned result with `computed == expected`, which is a Boolean expression returning True or False. This value is then assigned to the variable `success`, which is included in the `assert` statement. The last part of the assert statement, the text string `msg`, is optional and is simply included to give a more meaningful error message if the test fails. If we leave this out, and only write `assert success`, we will get a general message stating that the test failed (a so-called *assertion error*), but without much information about what actually went wrong.

There are some rules to observe when writing test functions:

- The test function must have at least one statement of the type `assert success`, where `success` is a Boolean variable or expression, which is `True` if the test passed and `False` otherwise. We can include more than one assert statement if we want, but we always need at least one.
- The test function should take no arguments. The function to be tested will typically be called with some arguments, but these should be defined as local variables inside the test function.
- The name should always be `test_`, followed by the name of the function we want to test. Following this convention is useful because it makes it obvious to anyone reading the code that the function is a test function, and it is also used by tools that can automatically run all test functions in a given file or directly. More about this below.

If we follow these rules, and remember the fundamental idea that a test function simply compares the returned result with the expected result, writing test functions does not have to be complicated. In particular, many of the functions we write in this course will evaluate some kind of mathematical function and then return either a number or a list/tuple of numbers. For this type of function the recipe for test functions is quite rigid, and the structure is usually exactly the same as in the simple example above.

If you are new to programming, it may be confusing to be faced with a general task like "write a test function for the Python function ‘somefunc(x,y)‘",

and it is natural to ask questions about what arguments the function should be tested for, and how you can know what the expected values are. In such cases it is important to remember the overall idea of test functions, and also that these are choices that must be made by the programmer. You have to choose a set of suitable arguments, then calculate or otherwise predict by hand what the function *should* return for these arguments, and write the comparison into the test function.

A test function can include mulitple tests. We can have multiple assert statements in a single test function. This can be useful if we want to test a function with different arguments. For instance, if we write a test function for one of the piecewisely defined mathematical functions considered earlier in this chapter it would be natural to test all the separate intervals on which the function is defined. The following code illustrates how this can be done.

```
from math import sin, pi

def f(x):
    if 0 <= x <= pi:
        return sin(x)
    else:
        return 0

def test_f():
    x1, exp1 = -1.0, 0.0
    x2, exp2 = pi/2, 1.0
    x3, exp3 = 3.5, 0.0

    tol = 1e-10
    assert abs(f(x1)-exp1) < tol, f'Failed for x = {x1}'
    assert abs(f(x2)-exp2) < tol, f'Failed for x = {x2}'
    assert abs(f(x3)-exp3) < tol, f'Failed for x = {x3}'
```

Note here that since we compare floating point numbers, which have finite precision on a computer, we compare with a tolerance rather than the equality `==`. The tolerance `tol` is some small number, chosen by the programmer, which should be small enough that we would consider a difference of this magnitude insignificant, but larger than the machine precision ($\approx 10^{-16}$). In practice it will quite often work to compare floats using `==`, but sometimes it fails and it is impossible to predict when it fails. The code therefore becomes unreliable, and it is much safer to compare with a tolerance. On the other hand, when we work with integers we can always use `==`.

One may argue that the test function code above is quite inelegant and repetitive, since we repeat the same lines multiple times with very minor changes. Since we only repeat three lines it may not be a big deal in this case, but if we included more assert statements it would certainly be both boring and error-prone to write code in this way. In the previous chapter we introduced loops as a much more elegant tool for perforoming such repetitive tasks. Using lists and a for-loop, the example above can be written as follows:

```
from math import sin, pi
```

```
def f(x):
    if 0 <= x <= pi:
        return sin(x)
    else:
        return 0

def test_f():
    x_vals = [-1, pi/2, 3.5]
    exp_vals = [0.0, 1.0, 0.0]

    tol = 1e-10
    for x, exp in zip(x_vals, exp_vals):
            assert abs(f(x)-exp) < tol, \
               f'Failed for x = {x}, expected {exp}, but got {f(x)}'
```

Python tools for automatic testing. An advantage of following the naming convention for test functions defined above is that there are tools that can be used to automatically run *all* test functions in a file or folder, and report if any bug has sneaked into the code. The use of such automatic testing tools is essential in larger development projects with multiple people working on the same code, but may also be quite useful for your own projects. The recommended and most widely used tool is called `pytest` or `py.test`. (`pytest` is simply the new name for `py.test`, they are the same tool.) We can run `pytest` from the terminal window, and pass it either a file name or a folder name as argument, as in

```
Terminal> py.test .
Terminal> py.test my_python_project.py
```

If we pass it a file name, `pytest` will look for functions in this file with the a name starting with `test_`, as specified by the naming convention above. All these functions will be identified as test functions and called by `pytest`, regardless of whether the test functions are actually called from inside the code. After execution, `pytest` will print a small summary of how many tests it found, and how many that passed and failed. For larger software projects it may be more relevant to give a folder name as argument to `pytest`, as in the first line above. In this case the tool will search the given folder (here `.`, the current folder) and all sub-folders for Python files with names starting or ending with `test` (for instance `test_math.py`, `math_test.py`, etc.). All these files will be searched for test functions following the naming convention, and these will be run as above. Large software projects typically have thousands of test functions, and collecting them in separate file and using automatic tools like `pytest` is very convenient. For the smaller programs we write in this course it may be just as easy to write the test functions in the same file as the functions we are testing.

It is important to remember that test functions run *silently* if the test passes. That is, we only get an output if there is an assertion error, otherwise nothing is printed to the screen. When using `pytest` we always get a summary specifying how many tests were run, but if we include calls to the test functions in the `.py` file, and run this file as normal, we will get no output if the test passes. This can be confusing, and one is sometimes left wondering if the test was called at all.

When first writing a test function it may be useful to include a print-statement inside the function simply to verify that the function is actually called. This statement should be removed once we know the function works correctly, and as we get used to how test functions work.

4 A summarizing example for Chapter 3

The following example illustrates most of the ideas from this chapter, and may be useful to read through to make sure you understand the new topics. An integral on the form

$$\int_a^b f(x)dx$$

can be approximated by *Simpson's rule*:

$$\int_a^b f(x)dx \approx \frac{b-a}{3n}\Big(f(a)+f(b)+4\sum_{i=1}^{n/2} f(a+(2i-1)h) + 2\sum_{i=1}^{n/2-1} f(a+2ih)\Big)$$

Problem: make a function `Simpson(f, a, b, n=500)` for computing an integral of `f(x)` by Simpson's rule. Call `Simpson(...)` for $\frac{3}{2}\int_0^\pi \sin^3 x dx$ (exact value: 2) for $n = 2, 6, 12, 100, 500$.

Function, simplest version.

```
def Simpson(f, a, b, n=500):
    """
    Return the approximation of the integral of f
    from a to b using Simpson's rule with n intervals.
    """

    h = (b - a)/float(n)

    sum1 = 0
    for i in range(1, n/2 + 1):
        sum1 += f(a + (2*i-1)*h)

    sum2 = 0
    for i in range(1, n/2):
        sum2 += f(a + 2*i*h)

    integral = (b-a)/(3*n)*(f(a) + f(b) + 4*sum1 + 2*sum2)
    return integral
```

Improved function, with test for possible errors. Simpson's rule only works for even values of n, and the function above will fail if we provide it an

uneven argument `n`. We can improve the function by checking if `n` is an even integer, and make a correction if it is not.

```
def Simpson(f, a, b, n=500):

    if a > b:
        print 'Error: a=%g > b=%g' % (a, b)
        return None

    # Check that n is even
    if n % 2 != 0:
        print 'Error: n=%d is not an even integer!' % n
        n = n+1  # make n even

    # as before...
    ...
    return integral
```

The main program. Here we actually apply the function to compute the given integral:

```
def h(x):
    return (3./2)*sin(x)**3

from math import sin, pi

print('Integral of 1.5*sin^3 from 0 to pi:')
for n in 2, 6, 12, 100, 500:
    approx = Simpson(h, 0, pi, n)
    print(f'n={n}, approx={approx}, error={2-approx}')
```

A test function to test the `Simpson` function. Finally, we write a test function to verify that our `Simpson` function works as it should. To create the test, we utilize the known property of Simpson's rule that 2nd degree polynomials are integrated exactly.

```
def test_Simpson():       # rule: no arguments
    """Check that quadratic functions are integrated exactly."""
    a = 1.5
    b = 2.0
    n = 8
    g = lambda x: 3*x**2 - 7*x + 2.5        # test integrand
    G = lambda x: x**3 - 3.5*x**2 + 2.5*x   # integral of g
    exact = G(b) - G(a)
    approx = Simpson(g, a, b, n)
    success = abs(exact - approx) < 1E-14  # tolerance for floats
    msg = f'exact={exact}, approx={approx}'
    assert success, msg
```

We can now either include a call to `test_Simpson()` in the `.py`-file, or use pytest:

```
Terminal> pytest Simpson.py
...
Ran 1 test in 0.005s

OK
```