Chapter 4
Functions and Branching

This chapter introduces two fundamental programming concepts: *functions* and *branching*. We are used to *functions* from mathematics, where we typically define a function $f(x)$ as some mathematical expression of x, and then we can then evaluate the function for different values of x, plot the curve $y = f(x)$, solve equations of the kind $f(x) = 0$, and so on. A similar function concept exists in programming, where a function is a piece of code that takes one or more variables as input, carries out some operations using these variables, and produces 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 decisions are made in the code based on the value of some Boolean expression or variable. If the expression evaluates to true, one set of operations is performed, and if the expression is false, a different set of operations is. Such tests are essential for controlling the flow of a computer program.

4.1 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)
```

Additionally, we used a few non-mathematical functions, such as `len` and `range`

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


J. Sundnes, *Introduction to Scientific Programming with Python*, Simula SpringerBriefs on Computing 6,
https://doi.org/10.1007/978-3-030-50356-7_4

```
    (...)
```

and we also used functions 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 last type of function is quite special, since it is bound to an object, and operates directly on that object (`C.append` changes `C`). These bound functions are also referred to as *methods*, and will be considered in more detail in Chapter 8. In the present chapter we will primarily consider regular, unbound, functions. In Python, such functions provide easy access to already existing program code written by others (e.g., `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 them. One advantage of 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, and the function can thus 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 individual functions and ensure that they work correctly before putting the functions together into a complete program. If such tests are done properly, we can have some confidence that our main program works as expected. We will return 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 previously considered mathematical function

$$A(n) = P(1 + r/100)^n.$$

For given values $P = 100$ and $r = 5.0$, we can implement this in Python as follows:

```
def amount(n):
    P = 100
    r = 5.0
    return P*(1+r/100)**n
```

These two lines of code are very similar to the examples from Chapter 3, but they contain a few new concepts that are worth noting. Starting with the first line, `def amount(n):` is called the function *header*, and defines the function's interface. 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 define functions that take multiple arguments by separating the arguments with commas. The parentheses need to be there, even if we do not want the function to take any arguments, in which case we would just leave the parentheses empty.

The lines following the function header are the *function body*, which need to be indented. The indentation serves the same purpose as for the loops in Chapter 3: to specify which lines of code belong inside the function, or to the function body. The two first lines of the function body are regular assignments, but since they occur inside a function, they define *local variables* `P` and `r`. Local variables the argument `n` are used inside the function just as regular variables. We will return to this topic in more detail later. The last line of the function body starts with the keyword `return`, which is also new in this chapter and is used to specify the output returned by the function. It is important not to confuse this return statement with the print statements we used previously. The use of `print` will simply output something to the screen, while `return` makes the function provide an output, which can be thought 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 that 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 there is no output until we start evaluating the function 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* as basically 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 include 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 amount(n):
    P = 100
    r = 5.0
    return P*(1+r/100)**n

year1 = 10
a1 = amount(year1)                    # call
a2 = amount(5)                        # call
```

[1] This is not entirely true, since defining the function creates a function object, which we can see by defining a dummy function in the Python shell and then calling `dir()` to obtain 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.

```
print(a1, a2)
print(amount(6))                          # call
a_list = [amount(year) for year in range(11)] #multiple calls
```

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

Note that, unlike many other programming languages, Python does not require the type of function arguments to be specified. Judging from the function header only, the argument of `amount(n)` above could be any kind of variable. However, by looking at how `n` is used inside the function, we can tell that it must be a number (integer or float). If we write complex functions where the argument types are not obvious, we can insert a comment immediately after the header, a so-called *doc string*, to tell users what the arguments should be. We will return to the topic of doc strings later in this chapter.

4.2 Function Arguments and Local Variables

Just as in mathematics, we can define Python functions with more than one argument. The formula above involves both P and r in addition to n, and including them all as arguments could be useful. The function definition could then look like

```
def amount(P, r, n):
    return P*(1+r/100.0)**n

# sample calls:
a1 = amount(100, 5.0, 10)
a2 = amount(10, r= 3.0, n=6)
a3 = amount(r= 4, n = 2, P=100)
```

Note that we are using the arguments `P`, `r`, and `n` inside the function exactly as in the previous example, where we defined `P` and `r` inside the function. Inside a function, there is no distinction between such *local variables* 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, whereas variables defined inside functions are local. The local variables are only defined and available inside a function, whereas global variables can be used everywhere in a program. If we tried to access `P, r`, or `n` (e.g., by `print(P)`) from outside the function, we will simply obtain an error message stating that the variable is not defined.

Arguments can be *positional arguments* or *keyword arguments*. Notice also the alternative ways of calling a function. We can either specify the

argument names in the call, as in `r=3.0, n=6`, 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 the variable to which they are assigned. Arguments that are passed including the name are called *keyword arguments*. Keyword arguments need to match the definition of the function; that is, calling the function above with `amount(100, 5.0, year=5)` would cause an error message because `year` is not defined as an argument to the function. Another rule worth noting is that a positional argument cannot follow a keyword argument; a call such as `amount(100, 5.0, n=5)` is fine, but `amount(P=100, 5.0, 5)` is not and the program will stop with an error message. This rule is quite logical, since a random mix of positional and keyword arguments would make the call very confusing.

The difference between local and global variables. The distinction between local and global variables is generally important in programming, and can be confusing at first. 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 as 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 assigned a value to the variables `P` and `r` outside the function, anywhere before the first call to `amount`, and the code would still work:

```
P = 100
r = 5.0

def amount(n):
    return P*(1+r/100)**n

print(amount(7))
```

Here `n` is passed as an argument, while, for `P` and `r`, the values assigned outside the function is used. However, it is also possible to define local and global variables with the same name, such as

```
P = 100
r = 5.0

def amount(n):
    r = 4.0
    return P*(1+r/100)**n
```

Which value of `r` is used in the function call here? Local variable names always take precedence over the global names. When the mathematical formula is encountered in the code above, Python will look for the values of the variables `P, r`, and `n` that appear in the formula. First, the so-called *local namespace* is searched, that is, Python looks for local variables with the given names. If

local variables are found, as for `r` and `n` in this case, these values are used. If some variables are not found in the 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, that is, 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 sequential search for variables is quite natural and logical, but still a potential source of confusion and programming errors. Additional confusion can arise if we attempt to change a global variable inside a function. Consider, for instance, this small extension of the code above:

```
P = 100
r = 5.0

def amount(n):
    r = 4.0
    return P*(1+r/100)**n

print(amount(n=6))
print(r)
```

```
126.53190184960003
5.0
```

As revealed by the print statements, `r` is set to 4.0 inside the function, but the global variable `r` remains unchanged after the function has been called. Since the line `r = 4.0` occurs inside a function, Python will treat this as the definition of a new local variable, rather than trying to change a global one. We thus define a new local `r` with value 4.0, while there is still another `r` defined in the global namespace. After the function has ended, the local variable no longer exists (in programming terms, it *goes out of scope*), whereas the global `r` is still there and has its original value. If we actually want to change a global variable inside a function, we must explicitly state so by using the keyword `global`. Consider this minor change of the code above:

```
P = 100
r = 5.0

def amount(n):
    global r
    r = 4.0
    return P*(1+r/100)**n

print(amount(n=6))
print(r)
```

```
126.53190184960003
4.0
```

In this case, the global `r` is changed. The keyword `global` tells Python that we do want to change a global variable, and not define a new local one. As a general rule, one should minimize the use of global variables inside functions and, instead, define all the variables used inside a function either as local variables or as arguments passed to the function. to the function. Similarly, if we want the function to change a global variable then we should make the function return this variable, 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 `r`, the following is a better way:

```
P = 100
r = 5.0

def amount(n,r):
    r = r - 1.0
    a = P*(1+r/100)**n
    return a, r

a0, r = amount(7)
print(a0, r)
```

Notice that, here, we return two values from the function, separated by a comma, just as in the list of arguments, and we also assign the returned values to the global variables `a0, r` in the line where the function is called. Although this simple example might not be the most useful in practice, there are many cases in which it is useful for a function call to change a global variable. In such cases the change should always be performed in this way, by passing the global variable in as an argument, returning the variable from the function, and then assigning the returned value to the global variable. Following these steps is far better than using the `global` keyword inside the function, since it 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. For a more practically relevant example of multiple return values, say we want to implement a mathematical function so that both the function value and its derivative are returned. Consider, for instance, the simple physics formula that describes the height of an object in vertical motion; $y(t) = v_0 t + (1/2)gt^2$, where v_0 is the initial velocity, g is the gravitational constant, and t is time. The derivative of the function is $y'(t) = v_0 - gt$, and we can implement a Python function that returns both the function value and the derivative:

```
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 a comma, and we assign the values to the two global variables `position` and `velocity`, also separated by a comma. When a function returns multiple values like this, it actually returns a tuple, the immutable list type defined in the previous chapter. We could therefore replace the call above with something like the following:

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

```
(0.034199999999999786, -2.886)
<class 'tuple'>
```

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 would *unpack* the elements in the tuple into the corresponding variables. For this unpacking to work, the number of variables must match the length of the tuple; otherwise, we obtain an error message stating that there are too many or not enough values to unpack.

A function can return any number of 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.

Example: A function to compute a sum. For a more relevant function example, of a kind that will arise frequently in this book, consider the sum

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

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

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

    return s
```

```
#example use
x = 0.5
from math import log
print(L(x, 3), L(x, 10), -log(1-x))
```

The output from the print statement indicates that the approximation improves as the number of terms n is increased, as is usual for such approximating series. For many purposes, it would be useful if the function returned the error of the approximation, that is, $-\ln(1-x) - L(x;n)$, in addition to the value of the sum:

```
from math import log

def L2(x, n):
    s = 0
    for i in range(1,n+1):
        s += x**i/i
    value_of_sum = s

    error = -log(1-x) - value_of_sum
    return value_of_sum, error

# typical call:
x = 0.8; n = 10
value, 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 the 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

```
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. If `somefunc` does not return anything, how can we then call the function and assign the result to a variable? If we do not include a return statement in a function, Python will automatically return a variable with value `None`. The value of the variable `return_value` in this case will therefore 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 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 tabular format to the screen. This is useful in many contexts, including studying the convergence of series approximations such as the one above. The following function calls the `L2(x,n)` function defined 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]:
        value, error = L2(x, n)
        print(f'n={n:4d} approx: {value:7.6f}, error: {error:7.6f}')

table(0.5)
```

```
x=0.5, -ln(1-x)=0.6931471805599453
n=   1 approx: 0.500000, error: 0.193147
n=   2 approx: 0.625000, error: 0.068147
n=  10 approx: 0.693065, error: 0.000082
n= 100 approx: 0.693147, error: 0.000000
```

This function does not need to return anything, since entire purpose is to print information to the screen.

4.3 Default Arguments and Doc Strings

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 be assigned default values. We can achieve the same functionality 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 must therefore 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* are often referred to as keyword arguments, although these terms do not mean exactly the same thing. They are, however, closely related, which is why the terms are sometimes used interchangeably. 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 uses of the alternative function calls for a useless but illustrative function. Testing a simple function such as the following, 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 arguments, we can improve the function considered above, which implements the formula

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

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

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

#example calls:
y1, dy1 = yfunc(0.2)
y2, dy2 = yfunc(0.2,v0=7.5)
y3, dy3 = yfunc(0.2,7.5,10.0)
```

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 immediately 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 a doc string can be used:

```
def amount(P, r, n):
    """Compute the growth of an investment over time."""
    a = P*(1+r/100.0)**n
    return a

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. 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.

4.4 If-Tests for Branching the Program Flow

In computer programs we often want to perform different actions depending on a condition. As usual, we can find a similar concept in mathematics that 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 the value of the input x, and return either zero or $\sin(x)$ depending on the outcome. Such a decision 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 in the previous chapter. If the condition is true, the following block of indented code is executed (in this case, just one line); if not, the indented code block after `else` is executed. You might 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 if-tests, since, if the first one is always executed the others will never be reached.

Sometimes we just want a piece of code to be executed if a condition is true, and to do nothing otherwise. In such cases, we can skip the `else` block 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 above, we can 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 cans 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 only if the preceding ones are False. The conditions are checked one by one and, as soon as one is evaluated as 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 is true, the code inside the else block is executed.

Multiple branching has useful applications in mathematics, since we often see piecewise 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 can be 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 or piecewise-defined functions.

Inline if-tests for shorter code. The list comprehensions in Chapter 3 offered a more compact alternative to the standard way of defining lists, and a similar alternative 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 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)
```

Using this compact notation, we can write the example from the start of this section as

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

4.5 Functions as Arguments to 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 $f(x)$, which 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 as any other argument, and is 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 the types of arguments the function expects.

Lambda functions for compact inline function definitions. In order to use the function `diff2` above, one would standardly 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)
```

The 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, as follows:

```
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
```

It could be natural to ask whether 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 placed directly in the argument list of the other function. Instead of first defining `f(x)` and then passing it as an argument, as in the code above, we can 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 in which we need to pass a simple mathematical expression as an argument to a Python function. We save some typing, and could also improve the code's readability.

4.6 Solving Equations with Python Functions

Solving equations of the form $f(x) = 0$ is a frequently occuring task in all branches of science and engineering. For special cases, such as a linear or quadratic f, we have simple formulas that give us the solution directly. In the general case, however, the equation cannot be solved analytically, and we need to find an approximate solution using numerical methods. We shall see that we can create powerful and flexible tools for equation solving based on the building blocks introduced so far. Specifically, we will combine functions and function arguments with the while loop introduced in Chapter 3.

Finding roots on an interval with the bisection method. One of the simplest algorithms for solving equations of the form $f(x) = 0$ is called the *bisection method*. This method is founded on the intermediate value theorem, which states that, if a continuous function changes sign on an interval $[a,b]$ then there must be a value $x \in [a,b]$ such that $f(x) = 0$. In the bisection method we start by choosing an interval $[a,b]$ on which f changes sign (i.e., $f(a)f(b) < 0$), and then compute the midpoint $m = (a+b)/2$ and check the sign of $f(m)$. If f changes sign on $[a,m]$ then we repeat the process on the interval $[a,m]$; otherwise, we choose $[m,b]$ as our new interval and repeat the process there. These steps are conveniently implemented as a while loop, and we can create a generic tool by placing the while loop inside a function that takes a function as argument:

```
from math import exp

def bisection(f,a,b,tol= 1e-3):
    if f(a)*f(b) > 0:
        print(f'No roots or more than one root in [{a},{b}]')
        return

    m = (a+b)/2

    while abs(f(m)) > tol:
        if f(a)*f(m) < 0:
```

```
            b = m
        else:
            a = m
        m = (a+b)/2
    return m

#call the method for f(x)= x**2-4*x+exp(-x)
f = lambda x: x**2-4*x+exp(-x)
sol = bisection(f,-0.5,1,1e-6)

print(f'x = {sol:g} is an approximate root, f({sol:g}) = {f(sol):g}')
```

We see that the `bisection` function takes four arguments: the mathematical function $f(x)$ implemented as a Python function, the bounds for our initial interval, and the tolerance for the approximate solution. The first if-test of the function simply checks that f changes sign in $[a, b]$, which ensures that the function has at least one root on the interval. We then proceed to define the midpoint `m` and enter the while-loop, which forms the core of the algorithm. This loop will continue running as long as `abs(f(m)) > tol` (otherwise `m` is our solution), repeatedly checking whether f changes sign on $[a, m]$ or $[m, b]$, and then calculating a new `m` to repeat the process on an interval of half the size.

Newton's method gives faster convergence. The bisection method converges quite slowly, and other methods are far more popular for solving non-linear equations. In particular, numerous varieties of Newton's method are widely used in practice. Newton's method is based on a *local linearization* of the non-linear function $f(x)$. Starting with an initial guess x_0, we replaces $f(x)$ by a linear function $g(x)$ that satisfies $g(x) \approx f(x)$ in a small interval around x_0. Then, we solve the equation $g(x) = 0$ to find an updated guess x_1, and repeat the process of linearization around that point. Repeated application of these steps converges quickly towards the true solution, provided that the initial guess x_0 is sufficiently close. In mathematics, one step of the algorithm looks like

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

where x_n is the solution after n iterations, x_{n+1} is the improved approximation, and $f'(x_n)$ is the derivative of f in x_n.

Just as the bisection method, Newton's method is easy to implement in a while loop, and we can implement it as a generic function that takes a Python function implementing $f(x)$ as argument. The function will also need $f'(x)$, since this is used in the algorithm, as well as an initial guess x_0 and a tolerance:

```
from math import exp

def Newton(f, dfdx, x0, tol= 1e-3):
    f0 = f(x0)
    while abs(f0) > tol:
```

```
        x1 = x0 - f0/dfdx(x0)
        x0 = x1
        f0 = f(x0)
    return x0

#call the method for f(x)= x**2-4*x+exp(-x)
f = lambda x: x**2-4*x+exp(-x)
dfdx = lambda x: 2*x-4-exp(-x)

sol = Newton(f,dfdx,0,1e-6)

print(f'x = {sol:g} is an approximate root, f({sol:g}) = {f(sol):g}')
```

Notice how the `x0` variable is updated inside the loop. The algorithm only needs to know the value at one iteration to compute the next one, so for each iteration we update `x0` to hold the most recent approximation, and then use this to compute the next one. Note also that the implementation provided here is not very robust, and if the method does not converge, it will simply continue in an infinite loop. One simple way to improve the implementation is to stop the method after a given number of iterations:

```
from math import exp

def Newton2(f, dfdx, x0, max_it=20, tol= 1e-3):
    f0 = f(x0)
    iter = 0
    while abs(f0) > tol and iter < max_it:
        x1 = x0 - f0/dfdx(x0)
        x0 = x1
        f0 = f(x0)
        iter += 1

    converged = iter < max_it
    return x0, converged, iter

#call the method for f(x)= x**2-4*x+exp(-x)
f = lambda x: x**2-4*x+exp(-x)
dfdx = lambda x: 2*x-4-exp(-x)

sol, converged, iter = Newton2(f,dfdx,0,tol=1e-3)

if converged:
    print(f'Newtons method converged in {iter} iterations')
else:
    print(f'The method did not converge')
```

Newton's method usually converges much faster than the bisection method, but has the disadvantage the function f needs to be manually differentiated. In Chapter 8 we will see some examples of how this step can be avoided.

4.7 Writing Test Functions to Verify our Programs

In the first part of this chapter, we mentioned the idea of writing tests to verify that functions work as intended. This approach to programming can 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 approach 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 are also 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 of test functions is very simple. For a given function, which often takes one or more arguments, we choose arguments such that we can calculate the result of the function by hand. Inside the test function, we 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
    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 instead 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 to write an error message if the two are different. This task is precisely what `assert` does. The keyword `assert` should always be followed by a condition, `success` in the code above, that is true if the test passes and false if it fails. The code above follows the typical recipe; we compare the expected with the returned result in `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 see a general message stating that the test has failed (a so-called *assertion error*), but without much information about what actually went wrong.

Some rules should be observed 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 one or more arguments, but these should be defined as local variables inside the test function.
- The name of the function 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 directory. More about this is discussed 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 can be confusing to be faced with a general task such as "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 in the test function.

A test function can include multiple 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 piecewise-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, that is small enough that we would consider a difference of this magnitude insignificant, but greater than the machine precision ($\approx 10^{-16}$). In practice, comparing floats using `==` will quite often work, but sometimes it fails and it is impossible to predict when this will happen. 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 could 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 might 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 f or performing 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* the 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 can also be quite useful for your own projects. The recommended and most widely used tool is called `pytest` or `py.test`, where `pytest` is simply the new name for `py.test`. We can run `pytest` from the terminal window, and pass it either a file name or a folder name as an argument, as in

Terminal

```
Terminal> pytest .
Terminal> pytest my_python_project.py
```

If we pass it a file name, `pytest` will look for functions in this file with 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 elsewhere in the code. After execution, `pytest` will print a short summary of how many tests it found, and how many that passed and failed.

For larger software projects, it might be more relevant to give a directory name as argument to `pytest`, as in the first line above. In this case, the tool will search the given directory (here `.`, the directory we are currently in) and all its sub-directories for Python files with names starting or ending with `test` (e.g., `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 it is very convenient to collect them in a separate file and use automatic tools such as `pytest`. For the smaller programs we write in this course, it can be just as easy to write the test functions in the same file as the functions being tested.

It is important to remember that test functions run *silently* if the test passes; that is, we only obtain an output if there is an assertion error, otherwise nothing is printed to the screen. When using `pytest` we are always given a summary specifying how many tests were run, but if we include calls to the test functions directly in the `.py` file, and run this file as normal, there will be 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 can 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 become used to how the test functions work.