Chapter 7

Functions and More Loops

7.1 Function definitions

Common functions such as `sin()` and `sqrt()` are built into the NumPy library. You can define your own functions using the keywords `def` and `return`.

Imagine a code that uses the function $f(x) = \ln(x)\sin(x^3)/(x^2 + xe^x)$ throughout, in many different places. Rather than typing this complicated expression over and over, you can type it once by creating a function definition. The following code defines $f(x)$ with the name `myfxn()`. The main code evaluates `myfxn()` at 3 and prints the result:

```
def myfxn(x):
    return np.log(x)*np.sin(x**3)/(x**2 + x*np.exp(x))

# main code:
y = myfxn(3)
print(f"y = {y:.4}")
```

The output is `y = 0.01517`. This new function can be evaluated throughout the program, as the need arises.

The general form for a function definition is

```
def functionname():
    code to execute
    return value
```

Here,

- *functionname* is the name of the function. The parenthesis `()` contains a list of variables that the function will use.
- *code to execute* is code that is executed as part of the function.
- *value* is the value (or values) that the function returns.

Note that the first line ends with a colon and the remaining lines are indented.

The definition of `myfxn()` given above is particularly simple because there are no lines of "code to execute" between the `def` statement and the `return` statement. Here is another example:

```
def myfxn(x):
    num = np.log(x)*np.sin(x**3)   # compute numerator
    den = x**2 + x*np.exp(x)       # compute denominator
    return num/den

# main code:
y = myfxn(3)
print(f"y = {y:.4}")
```

The "code to execute" can contain loops and control structures, just like any other Python code.

Functions can help streamline your code and make it easier to understand. A function can be defined anywhere, as long as it occurs before the function is used. Most programmers place their function definitions at the beginning of the program, right after the import statements.

Exercise 7.1a

Write a program that defines the function $f(x) = \sin(e^{\sin x})$. Evaluate this function at $x = 0, 1, 2$ and 3, and print the results.

Exercise 7.1b

Create a function definition for

$$f(x, a) = x^2 + x + a \ .$$

Have your program compute $f(f(4, 2), 3)$ and $f(f(4, 3), 2)$. Which of these numbers is larger? (Try to guess before running your code.)

Exercise 7.1c

Use a Python function to define

$$f(n) = \begin{cases} n/2 \ , & \text{if } n \text{ is even} \ , \\ 3n + 1 \ , & \text{if } n \text{ is odd} \ . \end{cases}$$

This function acts on a positive integer n to produce another positive

integer $f(n)$. Your code should start with an initial choice for n, compute $f(n)$, and iterate; that is, use $f(n)$ as the new value of n, evaluate the function again, and repeat. Have your code print out each integer in the sequence, and stop iterating if the integer reaches 1. The *Collatz conjecture* says that for any initial n, the sequence will always reach 1. What is the longest sequence you can find?

7.2 Functions and variables

Let's take a closer look at functions. Consider a function that computes the length of the hypotenuse of a right triangle using the Pythagorean theorem. The program below defines the hypotenuse function, evaluates it, and prints the result:

```
def hyp(a,b):
    return np.sqrt(a**2 + b**2)

# main code:
c = hyp(3,4)
print("c =", c)
```

The output is `c = 5.0`. Note that `hyp()` is a function of two variables, the lengths of the two short sides of the right triangle.

The complete program consists of two parts, the function definition and the main code. Each part plays a distinct role. The role of the main code:

- When the main code reaches the statement `c = hyp(3,4)`, it passes the values `3` and `4` to the function.
- When the main code receives a value for the hypotenuse from the function, it assigns that value to the variable `c`.

The role of the function:

- When the function receives the values `3` and `4`, it assigns these numbers to the variables `a` and `b`.
- The function computes the hypotenuse `np.sqrt(a**2 + b**2)` and returns the result `5.0` to the main code.

The main code and the function are independent of one another, apart from the passing of numerical values.

The following program is equivalent to the one above:

```
def hyp(a,b):
```

```
    return np.sqrt(a**2 + b**2)

a = 3
b = 4
c = hyp(a,b)
print("c =", c)
```

Once again, the main code passes the numbers `3` and `4` to `hyp()`, and `hyp()` returns the result `5.0`.

In the program above, Python does *not* pass the variable names `a` and `b` to the function. It is up to the function to choose variable names for the numbers it receives. The variables `a` and `b` that appear in the function are "local" to the function, and are independent of the variables `a` and `b` that appear in the main code. We could also write this program as

```
def hyp(alpha,beta):
    return np.sqrt(alpha**2 + beta**2)

a = 3
b = 4
c = hyp(a,b)
print("c =", c)
```

The output is the same: `c = 5.0`.

Here is another example:

```
def hyp(a,b):
    return np.sqrt(a**2 + b**2)

a = 17.3
b = 21.8
c = hyp(3,4)
print("c =", c, "a =", a, "b =", b)
```

The function assigns `3` and `4` to its local variables `a` and `b`, computes the hypotenuse, and returns `5.0`. The output of this program is `c = 5.0, a = 17.3, b = 21.8`. The function does not care that the main code has assigned `17.3` and `21.8` to its own variables, also called `a` and `b`. Likewise, the main code does not care that the function has assigned `3` and `4` to its local variables `a` and `b`.

Exercise 7.2a

Write a function that computes the factorial of any positive integer. Use this function in a program that asks the user to input two pos-

itive integers, n and m, then computes the number of combinations of n objects taken m at a time:

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

Test your code to make sure it works correctly.

A function can return more than one number.

Exercise 7.2b

Use a function definition to solve $ax^2+bx+c = 0$ using the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \ .$$

The inputs to the function will be the coefficients `a`, `b` and `c`. Since there are two solutions, there are two return values. Let's call these values `x1` and `x2`, and name the function `quad()`. Then the return statement is `return x1, x2`. The main-code statement `x1, x2 = quad(a,b,c)` will assign the return values to `x1` and `x2`.

A function can accept a NumPy array as input, as long as the *code to execute* can evaluate the array.

Exercise 7.2c

Create a Python program that defines the function $\sin(a/x)$. Have your code produce a graph of $\sin(3/x)$ for $0.1 \leq x \leq 2.0$. The inputs to the function will be $a = 3$ and a NumPy array of x values. The function will return a NumPy array of $\sin(a/x)$ values.

7.3 Variables and parameters

In the previous section we saw that the variables used in the main code and the variables used in a function definition are isolated from one another. Only numerical values are passed between the main code and the function.

This isn't the full story. If a function depends on a variable whose value has *not* been passed from the main code, the function will "look outside" to the main code for the missing value.

Here is an example:

```
def hyp(a):
    return np.sqrt(a**2 + b**2)

a = 3
b = 4
c = hyp(a)
print("c =", c)
```

The main code passes the value `3` to the function `hyp()`. The function assigns this value to its local variable `a`. The next task for the function is to compute `np.sqrt(a**2 + b**2)`. Since the function doesn't have a value for `b`, it "looks outside" to the main code. It sees that `b` equals `4` in the main code, and uses this value to calculate `np.sqrt(a**2 + b**2)`.

You could also exclude `a` from the argument list of the hypotenuse function. It is probably not a good idea to exclude either `a` or `b`, or both. As a general rule, all of the variables used in a function definition should be included in the argument list. This makes the function easier to understand and easier to reuse in another code.

Let's look at another example. A ball is thrown straight up into the air. The ball's height y as a function of time t is

$$y = y_0 + v_0 t - \frac{1}{2} g t^2 \ , \tag{7.1}$$

where y_0 is the initial height, v_0 is the initial velocity, and g is the acceleration due to gravity. The program below uses a function definition to compute the ball's height at various times t:

```
def height(y0,v0,g,t):
    return y0 + v0*t - 0.5*g*t**2

y0 = 2.0       # initial height (m)
v0 = 10.0      # initial velocity (m/s)
g = 9.8        # acceleration (m/s^2)

for t in [0.0, 0.4, 0.8, 1.2, 1.6, 2.0]:
    y = height(y0,v0,g,t)
    print(f"y({t:.1f}) = {y:.3f}")
```

Exercise 7.3a

Run the code above. What is the output?

The function `height()` depends on four variables, y_0, v_0, g and t. The values of these variables are passed from the main code to the function, which returns the height of the ball.

Using terminology that is common in physics and mathematics, we might describe y_0, v_0 and g as *parameters* and t as a *variable*. The difference is this: Parameters have fixed values throughout the program, whereas variables take on multiple values. This is a somewhat loose distinction, but is usually clear in practice. In our "ball in air" program, the parameters y_0, v_0 and g never change, but the variable t takes on multiple values as the program unfolds.

If a function depends on many parameters, it can be tedious to include all of the parameters in the argument list. As a practical matter, it might be preferable to drop the parameters from the argument list. For example, we can drop `y0`, `v0` and `g` from the list of arguments for the height function:

```
def height(t):
    return y0 + v0*t - 0.5*g*t**2

y0 = 2.0       # initial height (m)
v0 = 10.0      # initial velocity (m/s)
g = 9.8        # acceleration (m/s^2)

for t in [0.0, 0.4, 0.8, 1.2, 1.6, 2.0]:
    y = height(t)
    print(f"y({t:.1f}) = {y:.3f}")
```

Only the variable `t` is explicitly passed to `height()`. The function looks to the main code for values of the parameters `y0`, `v0`, and `g`.

Exercise 7.3b

Verify that the program produces the correct output when the parameters are omitted from the argument of `height()`.

From the computer's point of view, there is really no distinction between parameters and variables. Thus, we could modify our code even further by dropping t from the argument list. That's probably not a good idea. As a general rule, all variables used in a function should be included in the argument list. Ideally, parameters should be included as well. However, this might be impractical if the number of parameters is large.

Exercise 7.3c

Newton's universal law of gravitation says that the gravitational force between objects of mass m_1 and m_2 is

$$F = \frac{Gm_1m_2}{r^2} \ ,$$

where $G = 6.67 \times 10^{-11}\,\mathrm{N\,m^2/kg^2}$ is Newton's constant. Here, r is the distance between the two objects. Write a program to compute the force between Earth ($m_2 = 5.97 \times 10^{24}\,\mathrm{kg}$) and the moon ($m_1 = 7.35 \times 10^{22}\,\mathrm{kg}$) at various distances. Use a function definition for F, treating r as a variable and G, m_1 and m_2 as parameters. The distance between Earth and the moon varies from roughly $r = 3.6 \times 10^8\,\mathrm{m}$ to $r = 4.1 \times 10^8\,\mathrm{m}$. How much does the force vary?

7.4 Beyond functions

In mathematics a function is a "machine" that takes an input (or inputs) and produces an output (or outputs). For most applications in scientific computing, we can think of Python functions in the same way. For example, the definition

```
def F(a,x):
    v = a*x
    return v
```

is the programming equivalent of $F(a, x) = ax$. The inputs are a and x, the output is ax.

A Python function definition can go beyond the usual mathematical concept of a function by modifying the inputs. This can be a source of error if not done carefully. Consider the main code

```
a = 2.0
x = np.array([1.1, 2.2, 3.3])
y = F(a,x)
print(f"a = {a}, x = {x}, y = {y}")
```

In the first line `a` is given the value `2.0` and in the second line `x` is assigned to the array `[1.1 2.2 3.3]`. In the third line the function `F()` is used to multiply each element of `x` by `a` and return the result. This program produces the output

```
a = 2.0, x = [1.1  2.2  3.3], y = [2.2  4.4  6.6]
```

Exercise 7.4a

Make the following modifications, one at a time, to the program above.

- Add the statement `a = 3.0` inside the function definition, before `v = a*x`. Does this change the value of `a` in the main code? Does this change the array `y`?
- Add the statement `x = np.array([2.2, 3.3])` inside the function definition, before `v = a*x`. Does this change `x` in the main code? Does this change `y`?
- Add the statement `x[1] = 4.4` inside the function definition, before `v = a*x`. Does this change `x` in the main code? Does this change `y`?

The statements `a = 3.0` and `x = np.array([2.2,3.3)]` inside the function definition do not affect the values of `a` or `x` outside the function, but they do affect the result for `y`. On the other hand, the statement `x[1] = 4.4` inside the function definition changes the array `x` both inside and outside the function.

This leads us to another general rule: The return statement should include all variables and parameters whose values might be changed by the function. This is the best way to insure that changes made inside the function are passed back to the main code. In the example above we should include `a` and `x` in the return statement:

```
def F(a,x):
    statements that modify a and x
    y = a*x
    return y, a, x
a = 2.0
x = np.array([1.1, 2.2, 3.3])
y, a, x = F(a,x)
print(f"a = {a}, x = {x}, y = {y}")
```

Exercise 7.4b

Verify that this code works as expected by placing statements such as `a = 3.0`, `x = np.array([2.2, 3.3])` and `x[1] = 4.4` inside the function definition.

7.5 More `for` loops

In Sec. 5.5 we considered a code that computes the square roots of the numbers from 0 through 10 in steps of 0.5. The code below produces the same result with a more readable output:

```
A = np.linspace(0,10,21)
for i in range(len(A)):
    print(f"The sqrt of {A[i]:.2f} is {np.sqrt(A[i]):.4f}")
```

Here, we are using a `for` loop to create a separate print statement for each element of `A`. As a byproduct, we can compute the square root values "on the fly" inside the `print` command.

In the code above the `for` loop cycles over values of the array index. The following code acheives the same result by letting the `for` loop cycle over the values of the array itself:

```
A = np.linspace(0,10,21)
for x in A:
    print(f"The sqrt of {x:.2f} is {np.sqrt(x):.4f}")
```

The command `for x in A` tells Python to repeat the indented code for each element `x` in the array `A`. In the first pass through the loop, `x` is set equal to `A[0]`; in the second pass through the loop, `x` is set equal to `A[1]`; *etc.*

Exercise 7.5a

Create an array such as `A = np.array([-2.3, 5.4, 7.1, -4.2, 6.8])`. Write a code that will compute the sum of the array elements. Do this in two ways, by looping over the array index and by looping over the array elements. (The NumPy function `sum(A)` will sum the elements of an array `A`. Use this to check your answers.)

The following code computes the exponentials of the numbers 0.0 through 2.0 in increments of 0.1:

```
for n in range(0,21):
    x = n/10
    expx = np.exp(x)
    print(f"exp({x:.1f}) = {expx:.3f}")
```

Here we created the numbers $x = 0.0, 0.1, \ldots, 2.0$ by dividing the integers $n = 0, 1, \ldots, 20$ by 10.

Exercise 7.5b

Create the numbers 0.0 through 2.0 in increments of 0.1 using the NumPy `linspace` command. Use a `for` loop to cycle through those numbers. Compute and print the exponential of each number.

Exercise 7.5c

Write a code that accepts a word as user input and counts the number of vowels in the word. (Vowels are the letters "a", "e", "i", "o" and "u".)

7.6 `while` loops

A `for` loop is useful when we know (or the code can compute ahead of time) the number of times the loop should be executed. A `while` loop is useful when we don't know how many times the loop should be executed. Rather, the number of iterations of the loop depends on the results of calculations within the loop.

The syntax for the `while` loop is

`while` *condition*:
    code to execute and repeat as long as condition is `True`

The code to be executed must be indented.

The following code computes the exponentials of $0.0, 0.1, \ldots$ *etc.* and continues until the result exceeds 1000:

```
x = 0.0
expx = 0.0
while expx < 1000:
    expx = np.exp(x)
    print(f"exp({x:.1f}) = {expx:.3f}")
    x = x + 0.1
```

When Python reaches the `while` statement, it executes the loop repeatedly, as long as the condition `expx < 1000` evaluates to `True`. The program exits the loop when the condition evaluates to `False`.

Some things to note about this code:

- The initial value of `x` is placed before the `while` loop, and `x` is incremented by 0.1 for each pass through the loop.

- The variable `expx` is given a value 0.0 before the `while` loop. This allows Python to evaluate the conditional `expx < 1000` when it first reaches the `while` statement. The initial value for `expx` doesn't matter, as long as it is less than 1000.

Exercise 7.6a

The code above doesn't stop looping until `expx` exceeds 1000. As a result, in the final line from the `print` command, `expx` is greater than 1000. This might not be what we intended. Rewrite the code so that it only prints the results with `expx < 1000`.

Recall that conditions are statements that evaluate to `True` or `False`. Most conditions involve a comparison between two quantities using a comparison operator, such as `==` or `<`. Python also defines certain quantities as intrinsically `True` or `False`. For example, nonzero numbers and the word `True` will evaluate to `True`. The number zero and the word `False` will evaluate to `False`.

It is common to use the conditional `True` with a `while` loop, along with the command `break` to exit from the loop. As an example, let's find the cubes of all positive integers until the result exceeds some maximum value, say, 2000. Here's one way:

```
n = 1
while True:
    x = n**3
    if x <= 2000:
        print(x)
        n = n + 1
    else:
        break
```

Comments:

- The counter `n` is initialized to `1` before the loop begins.
- The condition `True` always evaluates to `True`.
- The `if` statement checks to see if `x` is less than or equal to `2000`.
- If `x <= 2000` then `x` is printed, the counter is incremented, and the loop is repeated.
- If `x` is not `<= 2000` (that is, `x` is greater than `2000`) then the `break` command causes the `while` loop to terminate.

Exercise 7.6b

Write a code to find the largest sum of integers $1 + 2 + 3 + \cdots$ that is less than 10^6 (one million).

Exercise 7.6c

Create an array containing all of the numbers of the form 3^n (where n is a positive integer) that are less than 10000.

Warning: Don't use `while True` without `break`. Since `True` always evaluates to `True`, the `while` loop will continue running forever, in an "infinite loop," if there is no `break` command.

What should you do if your code is stuck in an infinite loop? You can always stop the program while it is running. Look for a button or a drop-down menu item that says something like "interrupt execution" or "interrupt kernel."

7.7 More exercises

Exercise 7.7a

The *floor function* is defined as the largest integer less than or equal to its input. As a function of x, the floor function is often denoted $\lfloor x \rfloor$. For example, $\lfloor 3.26 \rfloor = 3$ and $\lfloor -4.6 \rfloor = -5$. NumPy has a built–in floor function called `floor()`. Create your own floor function and test it using various real numbers, both positive and negative.

Exercise 7.7b

The *Chebyshev polynomials* of the first kind are defined by

$$T_n(x) = \cos(n \arccos(x)) \ ,$$

where n is a nonnegative integer and $-1 \leq x \leq 1$. (You can use multiple–angle trig identities to show that $T_n(x)$ is an nth order polynomial in x.) Create a function definition for the Chebyschev polynomials and plot T_0 through T_5 on the same graph.