Chapter 1
Python Basics

Abstract This chapter explains basic programming concepts. After an overview of common Python distributions, we show how to use Python as a simple calculator. As a first step toward programming, variables and expressions are introduced. The arithmetic series and Fibonacci numbers illustrate the concepts of iteration and branching. We conclude this chapter with a program for the computation of a planet's orbital velocity around the Sun, using constants and functions from libraries and giving a small glimpse at objects in Python.

1.1 Using Python

There is quite a variety of Python installations. Depending on the operating system of your computer, you might have some basic Python preinstalled. Typically, this is the case on Linux computers. However, you might find it rather cumbersome to use, especially if you are not well experienced with writing source code in elementary text editors and executing the code on the command line. What is more, installing additional packages typically requires administrative privileges. If you work, for example, in a computer lab it is likely that you do not have the necessary access rights. Apart from that, Python version 2.x (that is major version 2 with some subversion x) is still in use, while this book is based on version 3.x.

Especially as a beginner, you will probably find it convenient to work with a a GUI (graphical user interface). Two popular choices for Python programming are Spyder and Jupyter. Spyder (www.spyder-ide.org) is a classical IDE (integrated development environment) which allows you to edit code, execute it and view the output in different frames. Jupyter (jupyter.org) can be operated via an arbitrary web browser. It allows you to run an interactive Python session with input and output cells (basically, just like the console-based `ipython`). Apart from input cells for typing Python source code, there are so-called markdown cells for writing headers and explanatory text. This allows you to use formatting similar to elementary HTML for webpages. A valuable feature is the incorporation of LaTeX to display mathematical expressions. The examples in this book can be downloaded as Jupyter notebooks and Python source code in zipped archives from uhh.de/phy-hs-pybook.

W. Schmidt and M. Völschow, *Numerical Python in Astronomy and Astrophysics*,
Undergraduate Lecture Notes in Physics,
https://doi.org/10.1007/978-3-030-70347-9_1

Since it depends on your personal preferences which software suits you best, we do not presume a particular GUI or Python distribution here. If you choose to work with Spyder or Jupyter, online documentation and tutorials will help you to install the software and to get started (browse the official documentation under docs.spyder-ide.org and jupyter-notebook.readthedocs.io/en/stable). For a comprehensive guideline, see also [1, appendices A and B]. A powerful all-in-one solution is Anaconda, a Python distribution and package manager that can be installed under Windows, macOS or Linux by any user (see docs.anaconda.com for more information). Anaconda provides a largely autonomous environment with all required components and libraries on a per-user basis. Of course, this comes at the cost of large resource consumption (in particular, watch your available disk space).

As a first step, check if you can run the traditional "Hello, World!" example with your favorite Python installation. Being astronomers, we use a slightly modified version:

```
1 print("Hello, Universe!")
```

In this book Python source code is listed in frames with lines numbered on the left (in the above example, there is just one line). Although these line numbers are not part of the source code (that's why they are shown outside of the frame), they are useful for referring to particular parts of a code example. You might be able to display line numbers in your code editor (in Jupyter notebooks, for example, line numbering can be switched on and off in the View menu), but you should not confuse these numbers with the line numbers used in this book. Usually we will continue the numbering over several frames if the displayed pieces of code are related to each other, but we also frequently reset line numbers to 1 when a new program starts or a new idea is introduced. Whenever you encounter a code line with number 1 it should alert you: at this point something new begins.

After executing the print statement in line 1 above, you should see somewhere on your screen the output[1]

```
Hello, Universe!
```

The quotes in the source code are not shown in the output. They are used to signify that the enclosed characters form a string. As you might have guessed, the `print()` function puts the string specified in parentheses on the screen (more precisely, in a window or frame that is used by Python for output).[2]

[1]How to execute Python code depends on the software you are using (consult the documentation). In a notebook, for example, all you need to do is to simultaneously press the enter and shift keys of your keyboard in the cell containing the code.

[2]Enclosing the string in parentheses is obligatory in Python 3. You may find versions of "Hello, World!" without parentheses on the web, which work only with Python 2.

1.2 Understanding Expressions and Assignments

Apart from printing messages on the screen, which is not particularly exciting by itself, Python can be used as a scientific calculator. Let us begin right away with an example from astronomy. Suppose we want to calculate the velocity at which Earth is moving along its orbit around the Sun. For simplicity, we treat the orbit as circular (in fact, it is elliptical with a small eccentricity of 0.017). From the laws of circular motion it follows that we can simply calculate the velocity as the circumference $2\pi r$ of the orbit divided by the period P, which happens to be one year for Earth. After having looked up the value of π, the orbital radius r (i.e. the distance to the Sun) in km, and the length of a year in seconds,[3] we type

```
1 2*3.14159*1.496e8/3.156e7
```

and, once evaluated by Python, we obtain

```
29.783388086185045
```

for the orbital velocity in km/s. Line 1 is an example for a Python expression consisting of literal numbers and the arithmetic operators `*` and `/` for multiplication and division, respectively. The factor of two in the formula for the circumference is simply written as the integer `2`, while the number π is approximately expressed in fixed-point decimal notation as `3.14159`.[4] The radius $r = 1.496 \times 10^8$ km is expressed as `1.496e8`, which is a so-called floating point literal . The character `e` followed by an integer indicates the exponent of the leading digit in the decimal system. In this case, `e8` corresponds to the factor 10^8. Negative exponents are indicated by a minus sign after `e`. For example, 10^{-3} can be expressed as `1.0e-3` or just `1e-3` (inserting + for positive exponents is optional).

Of course, there is much more to Python than evaluating literal expressions like a calculator. To get an idea how this works, we turn the example shown above into a little Python program:

```
1 radius = 1.496e8 # orbital radius in km
2 period = 3.156e7 # orbital period in s
3
4 # calculate orbital velocity
5 velocity = 2*3.14159*radius/period
```

Lines 1, 2, and 5 are examples for assignments. Each assignment binds the value of the expression on the right-hand side of the equality sign = to a name on the left-hand side. A value with a name that can be used to refer to that value is in essence what is

[3] Strictly speaking, the time needed by Earth to complete one revolution around the Sun is the *sidereal year*, which has about 365.256 d. One day has 86400 s.

[4] In many programming languages, integers such as `2` are treated differently than floating point numbers. For example, using `2.0` instead of the integer `2` in a division might produce a different result. In Python 3, it is usually not necessary to make this distinction. Alas, Python 2 behaves differently in this respect.

called a variable in Python.[5] In line 5, the variables `radius` and `period` are used to compute the orbital velocity of Earth from the formula

$$v = \frac{2\pi\, r}{P} \tag{1.1}$$

and the result is in turn assigned to the variable `velocity`. Any text between the hash character # and the end of a line is not Python code but a comment explaining the code to someone other than the programmer (once in a while, however, even programmers might be grateful for being reminded in comments about code details in long and complex programs). For example, the comments in line 1 and 2 provide some information about the physical meaning (radius and period of an orbit) and specify the units that are used (km and s). Line 4 comments on what is going on in the following line.

Now, if you execute the code listed above, you might find it surprising that there is no output whatsoever. Actually, the orbital velocity is computed by Python, but assignments do not produce output. To see the value of the velocity, we can append the print statement

```
6 print(velocity)
```

to the program (since this line depends on code lines 1–5 above, we continue the numbering and will keep doing so until an entirely new program starts), which results in the output[6]

```
29.783388086185045
```

This is the same value we obtained with the calculator example at the beginning of this section.

However, we can do a lot better than that by using further Python features. First of all, it is good practice to print the result of a program much in the same way as, hopefully, you would write the result of a handwritten calculation: It should be stated that the result is a velocity in units of km/s. This can be achieved quite easily by using string literals as in the very first example in Sect. 1.1:

```
7 print("orbital velocity =", velocity, "km/s")
```

producing the output

```
orbital velocity = 29.783388086185045 km/s
```

[5]The concept of a variable in Python is different from variables in programming languages such as C, where variables have a fixed data type and can be declared without assigning a value. Basically, a variable in C is a placeholder in memory whose size is determined by the data type. Python variables are objects that are much more versatile.

[6]In interactive Python, just writing the variable name in the final line of a cell would also result in its value being displayed in the output.

It is important to distinguish between the word ‘velocity’ appearing in the string `"orbital velocity ="` on the one hand and the variable `velocity` separated by commas on the other hand. In the output produced by **`print`** the two strings are concatenated with the value of the variable. Using such a print statement may seem overly complicated because we know, of course, that the program computes the orbital velocity and, since the radius is given in km and the period in seconds, the resulting velocity will be in units of km/s. However, the meaning of a numerical value without any additional information might not be obvious at all in complex, real-world programs producing a multitude of results. For this reason, we shall adhere to the practice of precisely outputting results throughout this book. The simpler version shown in line 6 may come in useful if a program does not work as expected and you want to check intermediate results.

An important issue in numerical computations is the precision of the result. By default, Python displays a floating point value with machine precision (i.e. the maximal precision that is supported by the way a computer stores numbers in memory and performs operations on them). However, not all of the digits are necessarily significant. In our example, we computed the orbital velocity from parameters (the radius and the period) with only four significant digits, corresponding to a relative error of the order 10^{-4}. Although Python performs arithmetical operations with machine precision, the inaccuracy of our data introduces a much larger error. Consequently, it is pointless to display the result with machine precision. Insignificant digits can be discarded in the output by appropriately formatting the value:

```
print("orbital velocity = {:5.2f} km/s".format(velocity))
```

Let us see how this works:

- The method **`format`**`()` inserts the value of the variable in parentheses into the preceding string (mind the dot in between). You will learn more about methods in Sect. 1.4.
- The placeholder `{:5.2f}` controls where and in which format the value of the variable `velocity` is inserted. The format specifier `5.2f` after the colon `:` indicates that the value is to be displayed in fixed-point notation with 5 digits altogether (including the decimal point) and 2 digits after the decimal point. The colon before the format specifier is actually not superfluous. It is needed if several variables are formatted in one print statement (examples will follow later).

Indeed, the output now reads

```
orbital velocity = 29.78 km/s
```

Optionally, the total number of digits in the formatting command can be omitted. Python will then just fill in the leading digits before the decimal point (try it; also change the figures in the command and try to understand what happens). A fixed number of digits can be useful, for example, when printing tabulated data.

As the term variable indicates, the value of a variable can be changed in subsequent lines of the program by assigning a new value. For example, you might want to calculate the orbital velocity of a hypothetical planet at ten times the distance of the Earth from the Sun, i.e. $r = 1.496 \times 10^9$ km. To that end, we could start with the assignment `radius=1.496e9`. Alternatively, we can make use of the of the current value based on the assignment in line 1 and do the following:

```
 9  radius = 10*radius
10  print("new orbital radius = {:.3e} km".format(radius))
```

Although an assignment may appear as the equivalent of a mathematical equality, it is of crucial importance to understand that it is not. Transcribing line 9 into the algebraic equation $r = 10r$ is nonsense because one would obtain $1 = 10$ after dividing through r, which is obviously a contradiction. Keep in mind:

The assignment operator = in Python means *set to*, not *is equal to*.

The code in line 9 thus encompasses three steps:

(a) Take the value currently assigned to `radius`,
(b) multiply this value by ten
(c) and reassign the result to `radius`.

Checking this with the print statement in line 10, we find that the new value of the variable `radius` is indeed 10 times larger than the original value from line 1:

```
new orbital radius = 1.496e+09 km
```

The radius is displayed in exponential notation with three digits after the decimal point, which is enabled by the formatting type `e` in place of `f` in the placeholder `{:.3e}` for the radius (check what happens if you use type `f` in line 10). You must also be aware that repeatedly executing the assignment `radius = 10*radius` in interactive Python increases the radius again and again by a factor of 10, which is possibly not what you might want. However, repeated operation on the same variable is done on purpose in iterative constructions called loops (see Sect. 1.3).

After having defined a new radius, it would not be correct to go straight to the computation of the orbital velocity since the period of the orbit changes, too. The relation between period and radius is given by Kepler's third law of planetary motion, which will be covered in more detail in Sect. 2.2. For a planet in a circular orbit around the Sun, this relation can be expressed as[7]

$$P^2 = \frac{4\pi^2}{GM} r^3 \,, \tag{1.2}$$

[7]Here it is assumed that the mass of the planet is negligible compared to the mass of the Sun. For the general formulation of Kepler's third law see Sect. 2.2.

where M=1.989 × 10^{30} kg is the mass of the Sun and G=6.674 × 10^{-11} N kg^{-2} m^2 is the gravitational constant. To calculate P for given r, we rewrite this equation in the form

$$P = 2\pi\,(GM)^{-1/2}\,r^{3/2}.$$

This formula can be easily turned into Python code by using the exponentiation operator `**` for calculating the power of an expression:

```
11 # calculate period in s from radius in km (Kepler's third law)
12 period = 2*3.14159 * (6.674e-11*1.989e30)**(-1/2) * \
13          (1e3*radius)**(3/2)
14 # print period in yr
15 print("new orbital period = {:.1f} yr".format(period/3.156e7))
16
17 velocity = 2*3.14159*radius/period
18 print("new orbital velocity = {:.2f} km/s".format(velocity))
```

The results are

```
new orbital period = 31.6 yr
new orbital velocity = 9.42 km/s
```

Hence, it would take more than thirty years for the planet to complete its orbit around the Sun, as its orbital velocity is only about one third of Earth's velocity. Actually, these parameters are quite close to those of the planet Saturn in the solar system. The backslash character \ in line 12 is used to continue an expression that does not fit into a single line in the following line (there is no limitation on the length of a line in Python, but code can become cumbersome to read if too much is squeezed into a single line). An important lesson taught by the code listed above is that you always need to be aware of physical units when performing numerical calculations. Since the radius is specified in km, we obtain the orbital velocity in km/s. However, the mass of the Sun and the gravitational constants in the expression for the orbital period in lines 12–13 are defined in SI units. For the units to be compatible, we need to convert the radius from km to m. This is the reason for the factor 10^3 in the expression `(1e3*radius)**(3/2)`. Of course, this does not change the value of the variable `radius` itself. To avoid confusion, it is stated in the comment in line 11 which units are assumed. Another unit conversion is applied when the resulting period is printed in units of a year in line 15, where the expression `period/3.156e7` is evaluated and inserted into the output string via **`format`**. As you may recall from the beginning of this section, a year has 3.156 × 10^7 s.

Wrong unit conversion is a common source of error, which may have severe consequences. A famous example is the loss of NASA's Mars Climate Orbiter due to the inconsistent use of metric and imperial units in the software of the spacecraft.[8] As a result, more than $100 million were quite literally burned on Mars. It is therefore extremely important to be clear about the units of all physical quantities in a

[8]See mars.jpl.nasa.gov/msp98/orbiter/.

program. Apart from the simple, but hardly foolproof approach of using explicit conversion factors and indicating units in comments, you will learn different strategies for ensuring the consistency of units in this book.

1.3 Control Structures

The computation of the orbital velocity of Earth in the previous section is a very simple example for the implementation of a numerical algorithm in Python.[9] It involves the following steps:

1. Initialisation of all data needed to perform the following computation.
2. An exactly defined sequence of computational rules (usually based on mathematical formulas), unambiguously producing a result in a finite number of steps given the input from step 1.
3. Output of the result.

In our example, the definition of the variables `radius` and `period` provides the input, the expression for the orbital velocity is a computational rule, and the result assigned to the variable `velocity` is printed as output.

A common generalization of this simple scheme is the repeated execution of the same computational rule in a sequence of steps, where the outcome of one step is used as input for the next step. This is called iteration and will be explained in the remainder of this section. The independent application of the same operations to multiple elements of data is important when working with arrays, which will be introduced in Chap. 2.

Iteration requires a control structure for repeating the execution of a block of statements a given number of times or until a certain condition is met and the iteration terminates. Such a structure is called a loop. For example, let us consider the problem of summing up the first 100 natural numbers (this is a special case of an arithmetic series, in which each term differs by the previous one by a constant):

$$s_n \equiv \sum_{k=1}^{n} k = 1 + 2 + 3 + \ldots + n\,. \tag{1.3}$$

[9]The term algorithm derives from the astronomer and mathematician al-Khwarizmi whose name was transcribed to *Algoritmi* in Latin (cf. [2] if you are interested in the historical background). al-Khwarizmi worked at the *House of Wisdom*, a famous library in Bagdad in the early 9th century. Not only was he the founder of the branch of mathematics that became later known as algebra, he also introduced the decimal system including the digit 0 in a book which was preserved until the modern era only in a Latin translation under the title *Algoritmi de numero Indorum* (this refers to the origin of the number zero in India). The digit 0 is quintessential to the binary system used on all modern computers.

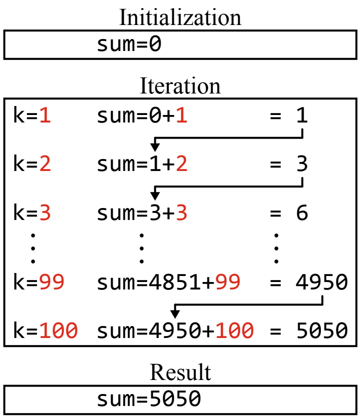

Fig. 1.1 Illustration of the computation of the sum s_{100} defined by Eq. (1.3) via a **for** loop. The box on the top of the figure contains the initialization statement prior to the loop (`sum` is set to zero). The middle box shows iterations of `sum` in the body of the **for** loop with the counter `k`, resulting in the sum shown at the bottom. The arrows indicate how values from one iteration are used in the next iteration. Values of the loop counter are shown in red

where $n = 100$. In Python, we can perform the summation using a **for** loop:

```
1  sum = 0   # initialization
2  n = 100 # number of iterations
3
4  for k in range(1,n+1): # k running from 1 to n
5      sum = sum + k      # iteration of sum
6
7  print("Sum =", sum)
```

The result is

```
Sum = 5050
```

The keywords **for** and **in** indicate that the loop counter `k` runs through all integers defined by `range(1,n+1)`, which means the sequence $1, 2, 3, \dots, n$ in mathematical notation. It is a potential source of confusion that Python includes the start value `1`, but excludes the stop value `n+1` in `range(1,n+1)`.[10]

The *indented* block of code following the colon is executed subsequently for each value of the loop counter. You need to be very careful about indentations in Python! They must be identical for all statements in a block, i.e. you are not allowed to use more or less white space or mix tabs and white spaces. We recommend to use one tab per indentation. The block ends with the first non-indented statement. In the example above, only line 5 is indented, so this line constitutes the body of the loop, which adds the value of the loop counter to the variable `sum`. The initial value of `sum` must be defined prior to the loop (see line 1). Figure 1.1 illustrates how the variables are iterated in the loop.

[10]There is a reason for the stop value being excluded. The default start value is `0` and `range(n)` simply spans the n integers $0, 1, 2, \dots, n-1$.

Actually, our Python program computes the sum exactly in the way intended by the teacher of nine-year old Carl Friedrich Gauss[11] in school, just by summing up the numbers from 1 to 100. However, Gauss realized that there is a completely different solution to the problem and he came up with the correct answer much faster than expected by his teacher, while his fellow students were still tediously adding up numbers. The general formula discovered by Gauss is (the proof can be found in any introductory calculus textbook or on the web):

$$s_n = \sum_{k=1}^{n} k = \frac{n(n-1)}{2}\,. \tag{1.4}$$

We leave it as an exercise to check with Python that this formula yields the same value as direct summation.

A slightly more demanding example is the calculation of the Fibonacci sequence[12] using the recursion formula

$$F_{n+1} = F_n + F_{n-1} \quad \text{for } n \geq 1, \tag{1.5}$$

with the first two elements

$$F_1 = 1\,, \quad F_0 = 0\,. \tag{1.6}$$

The meaning of Eq. (1.5) is that any Fibonacci number is the sum of the two preceding ones, starting from 0 and 1. The following Python program computes and prints the Fibonacci numbers $F_1, F_2, \ldots, F_{10}$ (or as many as you like):

```python
# how many numbers are computed
n_max = 10

# initialize variables
F_prev = 0 # 0. number
F = 1      # 1. number

# compute sequence of Fibonacci numbers
for n in range(1,n_max+1):
    print("{:d}. Fibonacci number = {:d}".format(n,F))

    # next number is sum of F and the previous number
    F_next = F + F_prev

```

[11] German mathematician, physicist, and astronomer who is known for the Gauss theorem, the normal distribution, and many other import contributions to algebra, number theory, and geometry.

[12] Named after Leonardo de Pisa, also known as Fibonacci, who introduced the sequence to European mathematics in the early 13th century. However, the Fibonacci sequence was already known to ancient Greeks. It was used to describe growth processes and there is a remarkable relation to the golden ratio.

```
15     # prepare next iteration
16     F_prev = F # first reset F_prev
17     F = F_next # then assign next number to F
```

The three variables F_prev, F, and F_next correspond to F_{n-1}, F_n, and F_{n+1}, respectively. The sequence is initialized in lines 5 and 6 (definition of F_0 and F_1). The recursion formula (1.5) is implemented in line 13. Without lines 16 and 17, however, the same value (corresponding to $F_0 + F_1$) would be assigned again and again to F_next. For the next iteration, we need to re-assign the values of F (F_n) and F_next (F_{n+1}) to F_prev (F_{n-1}) and F (F_n), respectively. The loop counter n merely controls how many iterations are executed. Figure 1.2 shows a schematic view of the algorithm (you can follow the values of the variables in the course of the iteration by inserting a simple print statement into the loop). The output produced by the program is[13]

```
1. Fibonacci number = 1
2. Fibonacci number = 1
3. Fibonacci number = 2
4. Fibonacci number = 3
5. Fibonacci number = 5
6. Fibonacci number = 8
7. Fibonacci number = 13
8. Fibonacci number = 21
9. Fibonacci number = 34
10. Fibonacci number = 55
```

Since two variables are printed, we need two format fields where the values of the variables are inserted (see line 10). As an alternative to using **format**(), the same output can be produced by means of a formatted string literal (also called f-string)[14]:

```
print(f"{n:d}. Fibonacci number = {F:d}")
```

Here, the variable names are put directly into the string. The curly braces indicate that the values assigned to the names n and F are to be inserted in the format defined after the colons (in this example, as integers with arbitrary number of digits). This is a convenient shorthand notation. Nevertheless, we mostly use **format**() in this book because the syntax maintains a clear distinction between variables and expressions on the one hand and formatted strings on the other hand. If you are more inclined to f-strings, make use of them as you please.

Suppose we would like to know all Fibonacci numbers smaller than, say, 1000. We can formally write this as $F_n < 1000$. Since it is not obvious how many Fibonacci numbers exist in this range, we need a control structure that repeats a block of code

[13]As you can see from Fig. 1.2, F_{11} is computed as final value of F_next. But it is not used. You can try to modify the program such that only 9 iterations are needed to print the Fibonacci sequence up to F_{10}.

[14]This feature was introduced with Python 3.6.

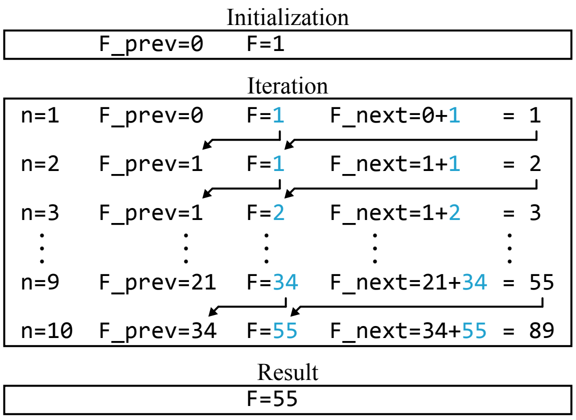
Initialization
F_prev=0 F=1
Iteration
n=1 F_prev=0 F=1 F_next=0+1 = 1
n=2 F_prev=1 F=1 F_next=1+1 = 2
n=3 F_prev=1 F=2 F_next=1+2 = 3
n=9 F_prev=21 F=34 F_next=21+34 = 55
n=10 F_prev=34 F=55 F_next=34+55 = 89
Result
F=55

Fig. 1.2 Illustration of the recursive computation of the Fibonacci sequence (see Eq. 1.5). In each iteration of the loop, the sum of `F_prev` and `F` is assigned to the variable `F_prev`. The resulting number is shown in the rightmost column. For the next iteration, this number is re-assigned to `F`, and the value of `F` to `F_prev`, as indicated by the arrows

as long as a certain condition is fulfilled. In such a case, it is preferable to work with a **`while`** loop. This type of loop enables us to modify our program such that all Fibonacci numbers smaller than 1000 are computed, without knowing the required number of iterations:

```python
# initialize variables
F_prev = 0 # 0. number
n,F = 1,1  # 1. number

# compute sequence of Fibonacci numbers smaller than 1000
while F < 1000:
    print("{:d}. Fibonacci number = {:d}".format(n,F))

    # next number is sum of F and the previous number
    F_next = F + F_prev

    # prepare next iteration
    F_prev = F # first reset F_prev
    F = F_next # then assign next number to F
    n += 1     # increment counter
```

The resulting numbers are:

```
1. Fibonacci number = 1
2. Fibonacci number = 1
3. Fibonacci number = 2
4. Fibonacci number = 3
5. Fibonacci number = 5
6. Fibonacci number = 8
7. Fibonacci number = 13
```

```
8. Fibonacci number = 21
9. Fibonacci number = 34
10. Fibonacci number = 55
11. Fibonacci number = 89
12. Fibonacci number = 144
13. Fibonacci number = 233
14. Fibonacci number = 377
15. Fibonacci number = 610
16. Fibonacci number = 987
```

Of course, the first ten numbers are identical to the numbers from our previous example. If you make changes to a program, always check that you are able to reproduce known results!

The loop header in line 6 of the above listing literally means: Perform the following block of code *while* the value of `F` is smaller than 1000. The expression `F<1000` is an example of a Boolean (or logical) expression. The operator < compares the two operands `F` and `1000` and evaluates to `True` if the numerical value of `F` is smaller than `1000`. Otherwise, the expression evaluates to `False` and the loop terminates. Anything that is either `True` or `False` is said to be of Boolean type. In Python, it is possible to define Boolean variables.

A **`while`** loop does not come with a counter. To keep track of how many Fibonacci numbers are computed (in other words the index n of the sequence F_n), we initialize the counter `n` along with `F` in the multiple assignment in line 3. This is equivalent to

```
n = 1
F = 1
```

Python allows you to assign multiple values separated by commas to multiple variables (also separated by commas) in a single statement, where the ordering on the left corresponds to the ordering on the right. We will make rarely use of this feature. While it is useful in some cases (for example, to swap variables[15] or for functions returning multiple values), multiple assignments are rather difficult to read and prone to errors, particularly if variables are interdependent.

While the loop counter of a **`for`** loop is automatically incremented, we need to explicitly increase our counter in the example above at the end of each iteration. In line 15, we use the operator += to increment `n` in steps of one, which is equivalent to the assignment `n=n+1` (there are similar operators -=, *=, etc. for the other basic arithmetic operations).

Let us try to be even smarter and count how many even and odd Fibonacci numbers below a given limit exist. This requires branching, i.e. one block of code will be executed if some condition is met and an alternative block if not (such blocks are also called clauses). This is exactly the meaning of the **`if`** and **`else`** statements in the following example:

[15] Another application in our Fibonacci program would be the merging of lines 13 and 14 into the multiple assignment `F_prev,F = F,F_next`.

```
 1 # initialize variables
 2 F_prev = 0 # 0. number
 3 F = 1      # 1. number
 4 n_even = 0
 5 n_odd = 0
 6 
 7 # compute sequence of Fibonacci numbers smaller than 1000
 8 while F < 1000:
 9     # next number is sum of F and the previous number
10     F_next = F + F_prev
11 
12     # prepare next iteration
13     F_prev = F # first reset F_prev
14     F = F_next # then assign next number to F
15 
16     # test if F is even (divisible by two) or odd
17     if F%2 == 0:
18         n_even += 1
19     else:
20         n_odd += 1
21 
22 print("Found {:d} even and {:d} odd Fibonacci numbers".\
23       format(n_even,n_odd))
```

Instead of a single counter, we need two counters here, `n_even` for even Fibonacci numbers and `n_odd` for the odd ones. The problem is to increment `n_even` if the value of `F` is an even number. To that end the modulo operator `%` is applied to get the remainder of division by two. If the remainder is zero, then the number is even. This is tested with the comparison operator == in the Boolean expression following the keyword **`if`** in line 17. If this expression evaluates to `True`, then the counter for even numbers is incremented (line 18). If the condition is `False`, the **`else`** branch is entered and the counter for odd numbers is incremented (line 20). You must not confuse the operator ==, which *compares* variables or expressions *without changing* them, with the assignment operator =, which sets the value of a variable. The program reports (we do not bother to print the individual numbers again):

```
Found 5 even and 11 odd Fibonacci numbers
```

Altogether, there are $5 + 11 = 16$ numbers. You may check that this is in agreement with the listed numbers.

1.4 Working with Modules and Objects

Python offers a collection of useful tools in the Python Standard Library (see docs.python.org/3/library). Functions such as **`print`**`()` are part of the Standard Library. They are called *built-in* functions. Apart from that, many more optional libraries (also called packages) are available. Depending on the Python distribution you use, you will find that some libraries are included and can be imported as shown below, while you might need to install others.[16] Python libraries have a hierarchical modular structure. This means that you do not necessarily have to load a complete library, but you can access some part of a library, which can be a module, a submodule (i.e. a module within a module) or even individual names defined in a module. To get started, it will be sufficient to consider a module as a collection of definitions. By importing a module, you can use variables, functions, and classes (see below) defined in the module.

For example, important physical constants and conversion factors are defined in the `constants` module of the SciPy library (for more information, see www.scipy.org/about.html). A module can be loaded with the **`import`** command:

```python
import scipy.constants
```

To view an alphabetically ordered list of all names defined in this module, you can invoke `dir(scipy.constants)` (this works only after a module is imported). By scrolling through the list, you might notice the entry `'gravitational_constant'`. As the name suggests, this is the gravitational constant G. Try

```python
print(scipy.constants.gravitational_constant)
```

which displays the value of G in SI units:

```
6.67408e-11
```

The same value is obtained via `scipy.constants.G`. Even so, an identifier composed of a library name in conjunction with a module and a variable name is rather cumbersome to use in programs. Alternatively, a module can be accessed via an alias:

```python
import scipy.constants as const

print(const.G)
```

also displays the value of G. Here, `const` is a user-defined nickname for `scipy.constants`.

It is also possible to import names from a module directly into the global namespace of Python. The variables we have defined so far all belong to the global namespace. The syntax is as follows:

```python
from scipy.constants import G

print(G)
```

[16]See, for example, packaging.python.org/tutorials/installing-packages
and docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-pkgs.html.

In this case, only `G` is imported, while in the examples above *all* names from `scipy.constants` are made available. Importing names via the keyword **`from`** should be used with care, because they can easily conflict with names used in assignments elsewhere. Python does not treat this as an error. Consequently, you might accidentally overwrite a module variable such as `G` with some other value.

By using constants from Python libraries, we can perform computations without looking up physical constants on the web or in textbooks and inserting them as literals in the code. Let us return to the example of a planet at 10 times the distance of Earth from the Sun, i.e. $r = 10$ au (see Sect. 1.2). Here is an improved version of the code for the computation of the orbital period and velocity:

```
 1  from math import pi,sqrt
 2  from astropy.constants import M_sun
 3  from scipy.constants import G,au,year
 4
 5  print("1 au =", au, "m")
 6  print("1 yr =", year, "s")
 7
 8  radius = 10*au
 9  print("\nradial distance = {:.1f} au".format(radius/au))
10
11  # Kepler's third law
12  period = 2*pi * sqrt(radius**3/(G*M_sun.value))
13  print("orbital period = {:.4f} yr".format(period/year))
14
15  velocity = 2*pi * radius/period # velocity in m/s
16  print("orbital velocity = {:.2f} km/s".format(1e-3*velocity))
```

The output of this program is

```
1 au = 149597870691.0 m
1 yr = 31536000.0 s

radial distance = 10.0 au
orbital period = 31.6450 yr
orbital velocity = 9.42 km/s
```

We utilize the value of π and the square-root function defined in the `math` module, which is part of the standard library. The function `sqrt()` imported from `math` is called in line 12 with the expression `radius**3/(G*M_sun.value)` as argument. This means that the number resulting from the evaluation of this expression is passed as input to `sqrt()`, which executes an algorithm to compute the square root of that number. The result returned by the function and is then multiplied with `2*pi` to obtain the orbital period. Moreover, we use constants and conversion factors from the SciPy and Astropy libraries. For instance, `au` is one astronomical unit in m and `year` is one year in s. The values are printed in lines 5 and 6. These conversion factors enable us to conveniently define the radius in astronomical units (line 8) and, after apply Kepler's third law in SI units, to print the resulting orbital period in years (lines 12 and 13). When printing the radius in line 9, the newline character `'\n'` at the beginning of the string inserts a blank line. Other than in Sect. 1.2, the orbital velocity assigned to `velocity` is in m/s. So we need to multiply by a factor of 10^{-3} to obtain the velocity in units of km/s in the final print statement.

In contrast to the gravitational constant `G`, which is simply a floating point number, the mass of the Sun defined in `astropy.constants` is a more complex object. In computer science, the term object has a specific meaning and refers to the object-oriented programming (OOP) paradigm. You can go a long way in Python without bothering about object-oriented programming. Nevertheless, you will find it helpful if you are aware of a few basic facts:

1. Everything in Python is an object.
2. An object contains a particular kind of data.
3. Objects have methods to manipulate the object's data in a controlled way.
4. A method can change an object or create a new object.

This implies that `G` is also an object, albeit a rather simple one. If you print `M_sun`, you will find quite a bit more information in there, such as the uncertainty of the value and its physical unit:

```
Name   = Solar mass
Value  = 1.9884754153381438e+30
Uncertainty  = 9.236140093538353e+25
Unit  = kg
Reference = IAU 2015 Resolution B 3 + CODATA 2014
```

Particular data items are called object attributes. For example, the value of the solar mass is an attribute of `M_sun`. You can fetch an attribute by joining the names of the object and the attribute with a dot. We refer to the attribute `value` in line 12 to obtain a pure number that can be combined with other numbers in an arithmetic expression. To list attributes belonging to an object, you can use **`dir`**`()`, just like for modules, or search the documentation. Attributes and methods are defined in classes. While objects belonging to the same class may contain different data, they have the same attributes and methods. For example, `M_earth` from `astropy.constants` has a `value` attribute just like `M_sun`, but the value behind this attribute is Earth's mass instead of the mass of the Sun. Both objects belong to the class `Quantity`. You can take a glimpse behind the curtain in Appendix A, where you are briefly introduced to writing your own classes.

Since everything in Python is an object, so is a string. Now you are able to better understand the meaning of **`format`**`()` being a method. It is a method allowing you to insert formatted numbers into a string.[17] While methods are relatives of Python functions, a method always has to be called in conjunction with a particular object. In the print statements in lines 9, 13, and 16, the objects are string literals. The

[17]Since strings are immutable objects, the method does not change the original string with placeholders. It creates a new string object with the formatted numbers inserted.

syntax is similar to accessing object attributes, except for the arguments enclosed in parentheses (here, the variables holding the numbers to be inserted). In general, you can call methods on names referring to objects – in other words, Python variables. This will be covered in more detail in the next chapter.