Prealgebra via Python Programming: First steps to perform large scale computational tasks in the Sciences and Engineerings

Sergio Rojas
Simon Bolívar University
37 PUBLICATIONS **266** CITATIONS

PREALGEBRA VIA PYTHON PROGRAMMING

First steps to perform large scale computational tasks in the Sciences and Engineerings

SERGIO ROJAS (AUTOR-EDITOR)

Physics Department

Universidad Simón Bolívar (USB)

Venezuela

CARACAS - VENEZUELA - 2018

Prealgebra via Python Programming:

First steps to perform large scale computational tasks in the Sciences and Engineerings

Sergio Rojas
Physics Department
Universidad Simón Bolívar
Venezuela

Caracas-Venezuela
May 31, 2018

Prealgebra via Python Programming: First steps to perform large scale computational tasks in the Sciences and Engineerings
by Sergio Rojas

Typesetting by the Author and Editor (Sergio Rojas) using the *memoir* LaTeX package.

Editor: Sergio Rojas

Legal Deposit/Depósito Legal: MI2018000516
ISBN: 978-980-18-0123-8

Draft copy as of May 31, 2018
Electronic publication:
`https://github.com/rojassergio/`

About the Author

Sergio Rojas (**`http://prof.usb.ve/srojas/`**)

Sergio Rojas is currently a Full Professor of Physics at the Universidad Simón Bolívar, Venezuela. Regarding his formal studies, he earned in 1991 a B.S in Physics with Thesis on Numerical Relativity from the Universidad de Oriente, Estado Sucre, Venezuela, and then, in 1998, he earned a Ph.D. in Physics from the Physics Department of the City College of the City University of New York, where he worked on the applications of Fluid Dynamics in the flow of fluids in porous media, gaining and developing since then a vast experience in programming as an aid to scientific research via fortran77/90 and C/C++. In 2001, he also earned a Master's degree in computational finance from The Oregon Graduate Institute of Science and Technology.

Sergio's teaching activities involve lecturing undergraduate and graduated physics courses at his home university, Universidad Simón Bolívar, Venezuela, including a course on Monte Carlo Methods and other on Computational Finance. His research interests include physics education research, fluid flow in porous media, and the application of the theory of complex systems and statistical mechanics in Financial Engineering. More recently, Sergio has been involved in Machine Learning and its applications in Science and Engineering via the Python programming language.

Sergio's is coauthor of the book *Learning SciPy for Numerical and Scientific Computing - Second Edition* (2015) [`https://www.packtpub.com/big-data-and-business-intelligence/learning-scipy-numerical-and-scientific-computing-second-edition`] and coauthor Editor of the self-published book (in Spanish) *Aprendiendo a programar en Python con mi computador: Primeros pasos rumbo a cómputos de gran escala en las Ciencias e Ingenierías* (2016) [`https://www.researchgate.net/publication/301293668`] and the author of the video course *Numerical and Scientific Computing with SciPy* (2017) [`https://www.packtpub.com/big-data-and-business-intelligence/numerical-and-scientific-computing-scipy-video`].

Preface

"O Inventamos o Erramos."

Simón Rodriguez
Maestro de América
1771-1854

This book was written for students and instructors who want to learn how to use a computer for other than the most common uses, such as web browsing, document creation, or paying bills online. This book is for anyone who wants to perform computational tasks that they design. In other words, if you wish to learn how to program a computer, this book is for you.

Because prealgebra is a subject that practically everyone is supposed to learn in grade school, it provides a platform to introduce basic computer programming concepts. Consequently, this book should also be of interest to students in middle or high school who want to learn how to program, and who are willing to invest the time and effort in learning a programming language that they could continue using throughout their schooling and in their professional life. Similarly, this book could also be of interest to pre-service and in-service mathematics teachers wishing to have at their disposal a complementary tool to assist in fostering understanding, competency, and interest in mathematics among their students. This book can be integrated with the teachers' curriculum as way to tackle non-traditional math problems using an inexpensive modern computer language. By the end of the book, a reader will have learned enough to be able to write a preliminary, step-by-step one variable equation solver that can be expanded in the future to use with more complex equations. In other words, by the end of the book, you will be able to write code that programs their machines to solve equations. This code is foundational and readers are ecouraged to learn on their own how to build on it to suit their mathematics learning needs.

Accordingly, the presentation of topics in the book, as seen in the Table of Contents, will conform to most of the standard course work covered by prealgebra textbook authors, and will consider topics such as whole numbers, integers, fractions, and decimals. Nevertheless, this book is not yet another prealgebra textbook. Think of this book as a complement to a prealgebra textbook. In other words, this book is not meant to replace the teaching of concepts of prealgebra in the classroom, but to help reinforce the learning of those topics by learning how to write programs (in *Python*) that students can use to practice prealgebra concepts discussed in their classes. This book could even lead you to rethink some of these topics, even if you already know them very well.

At its most basic level, programming is basically the craft of writing instructions that a computer can interpret and execute. There are many computer languages, and to learn programming, one needs to choose a particular computer language. In writing this book, we choose the **free and open source** *Python* (scripting) programming language, because it works in practically every operating system in use (including the also **free and open source** *Linux* operating system, a version of which we used to write this book).

To get a bit more technical, *Python* is a *high level computing language* which involves a translation of the user's instructions before those instructions can be understood by the computer. In this case, the interpreter is *Python*, and the instructions are expressed in standard alphanumeric English characters following a rigorous syntax based on such set of characters. Although this is also true for any other *high level computing language* like C/C++ or Fortran, a nice feature that makes the language *Python* suitable for prealgebra courses is that it comes with a symbolic algebraic computational system module (*SymPy*), which permits the execution of algebraic operations symbolically. That is, as required in many prealgebra operations, we can use symbols, instead of numbers, to make computations and obtain analytical results.

But the module *SymPy* also allows the carrying out of numerical computations. So we will be making heavy use of the module *SymPy* to facilitate and enhance prealgebra learning, mainly by performing algebraic operations and numerical computations. Please note that *Python* comes with other modules (namely *NumPy* and *SciPy*) that are the standard modules to perform numerical computations whenever efficiency is an issue. In this introductory book we will not make extensive use of those modules. We will introduce the *Matplotlib* module for visualizations. You will learn more about this in the introductory remarks of the first chapter, starting on page 1.

As mentioned, *Python* is compatible with practically every operating system currently in use. In writing the *Python* programs (also called scripts) that you will learn about in this book, we used the **free and open source** version of the Linux operating system called *Ubuntu*. However, the scripts should work as given in any other *Python* set up.

Free and open source programming languages make it easy and inexpensive for any school to pursue the idea that everyone needs to learn how to program early on. The task of programming helps to develop the ability to focus attention as well as to the ability to think consciously and critically, which reduces the tendency toward learning by rote mechanical memorization. Accordingly, this book can be used by anyone desiring to awaken and develop the skills of active deep thinkers.

Beware, however, that since simplicity and clarity drive the write-up of this book, we are not going to use expert preferred programming constructs, which are efficient, but can seem obscure to new learners. You can consult more advanced texts covering these kinds of expressions or scripts once you learn the basics of programming in *Python* in this book.

We encourage readers to send us comments, ask questions, and provide information on errors or broken web links via email to `srojas@usb.ve` or to alternative contacts found at the companion web site for this book at [`https://github.com/rojassergio/Python-`

`Prealgebra`]. More importantly, if you find yourself in the mood to collaborate to improve the readability and/or enhance the content of the book, feel free to contact me at the above email address.

Sergio Rojas
May 31, 2018

Contents

1

Getting, installing, and testing the programming python environment

"Learning is not attained by chance, it must be sought for with ardor and attended to with diligence."

Abigail Adams
1744 - 1818

1.1 Motivational remarks

In the context of this book, we'll understand programming as the craft of writing instructions that a computer will understand and execute literally (without complaint, unless something is wrong in the given instructions). In its own, a computer is unable to do anything. We, the programmer, are responsible for telling the computer what task to perform and how to do it. Accordingly, when it is told to do so, the computer execute instructions written (following a rigorous syntax) in a language that it is able to understand. If the given instructions do not conform with the allowed syntax, only error messages are dumped on the computer screen. If there are logical errors (like writing a multiplication sign instead of the adding one) no complaint will be given by the computer. Thus, the programmer (or the programming team) is responsible for the logic, efficiency, and rightness of the sequence of instructions a computer should execute.

Why learn how to program at an early stage of our formal educational track.? Following the aforementioned facts, in order to be able to write meaningful instructions that computer will understand (and do what we want) it is necessary that we understand what it is we would like the computer to do. Then we need to think about how to write such a task in the limited set of valid keywords available in the computer language of our choice to interact with the computer. It is a fact that performing such steps requires the activation and use of high order thought processes to accomplish the fascinating task of making the computer to do work for us. Reaching such level of precision requires discipline to devote many hours of enjoyable hard work, having as intellectual reward the satisfaction to have made the computer to do (perhaps efficiently) the task that we want, that perhaps no body else has done yet.

Accordingly, in answering the posed question partially, we can point out that many studies (for details, in case you are curious, see references at the end of this chapter on page 20) has shown that learning how to program helps students to (whatever it means) develop high order thinking skills, which are necessary to approach (via well formed reasoning) quantitative and non quantitative subjects with confidence, that in turns is a desired key outcome of the many mathematical course work training programs, like for instance the *Common Core State Standards* [http://www.corestandards.org/Math/] of mathematical practice in the United States (US). In other words, knowing how to program gives you another dimension to think about *problem-solving*, a critical skill to perform well in the Sciences and Engineerings that is stressed in many courses (including the Prealgebra one). That is, via programming you can easily explore (via the computer) solution to non-routine problems, performing little mathematical experiments to foster your understanding.

To further your interest in learning how to program, perhaps, at this point, you would like to read the headline story back of 2002 related to the computer scientists guys who used the computer to find out via an efficient algorithm whether a given number is prime [http://www.ams.org/notices/200305/fea-bornemann.pdf] [https://mathoverflow.net/questions/12085/experimental-mathematics], a topic you'll be studying in your Prealgebra course work.

In addition to what we have just said, learning how to program fits beautifully within the framework of many *Computer Science* (CS) programs for young students, like the recently created (in the US) *Computer Science for All* (CSforAll) Consortium [http://www.csforall.org/] which, recognizing the importance of being knowledgeable on the aspects of dealing with computers internally, is dedicated "*to enable students to achieve CS literacy as an integral part of their educational experience both in and out of school.*"[http://www.csforall.org/] Moreover, perhaps widening the question and answering the commonly asked one *when am I ever going to need this*?, a recent study [https://doi.org/10.1063/PT.3.3763] shows that programming is one preponderant skills used on daily, weekly, or monthly basis by physics graduates working at private-sector jobs in engineering or Computer Science. Other endeavors giving preponderance to learning how to program are the RasberryPi [http://bit.ly/1Jua4qn] and Arduino [https://www.arduino.cc/] movements.

Thus, without doubt, knowing how to program will be a required (like reading and writing) skill in the near future. The quick development of Artificial Intelligence nowadays is also a factor demanding such skills (many more devices will require some sort of programming to operate at its optimum powerfulness).

Now, because of the great variety of possibilities, finding a computer language suitable to learn how to program at an early stage of our formal education is a difficult task. This is where *Python* comes to mind.

Complementing what was said in the Preface, *Python*, as mentioned there, is a **free and open source** (scripting) programming language having the functionality required by developers and users to any modern programming language, some of which are possessing an efficient **high level data structure** and allows **object oriented programming**. These requirements might not mean much to anyone starting to learn how program, but it is good to know from the start that we are making a good use of our time by learning to program in a long lasting programming language, that since very recently has been a common programming language at top US Universities (shown via research studies in 2014 and 2017. For details see the web references section, on page 20).

As a very sophisticated programming language, *Python* also provides functionality to program in parallel using both the standard multiprocessing capabilities provided by the Central Processing Units (CPUs) of any modern computer and the power of the Graphical Processing Units (GPUs) available via the many existing video cards. Hence the subtitle of this book: *First steps to perform large scale computational tasks in the Sciences and Engineerings*. In addition, *Python* has (and continues) growing to provides support to handle problems in practically all areas of Science and Engineering, ranging from Astronomy [`http://www.astropython.org/`] to Molecular Biology [`http://biopython.org/DIST/docs/tutorial/Tutorial.html`] (an abridged list of the covered subjects can be found at [`https://pypi.python.org/pypi/`]).

Now, in the same way as it is impossible to learn swimming by just reading a book, neither one learns how to program by only reading a textbook. It is thus necessary to practice writing and executing computer programs. This helps on keeping a mental library of what works a what does not, as well as where to look to correct compilation (in case of working with a compiled language) and runtime crashes.

Accordingly, assuming that *we learn by doing*, a lemma that anyone reading this book should adopt is found in a thought of the Nobel Prize Herbert Simon regarding that

Learning results from what the student does and think and only from what the student does and think. The teacher can advance learning only by influencing what the student does to learn.

Thus, taking for granted that you (the reader) are well interested to learn how to program or using your programming skills in doing your Prealgebra tasks, we are confident that you are willing to read this book carefully enough, executing the programming activities presented all over the content of the text. You can steep up your learning curve by taking advantage of the many resources available on the internet for sharing on the topic of *Python* programming, among them the resources listed at the *Python* community page [`https://www.python.org/community/`].

1.2 The terminal, system shell or console of commands

Before continuing, let's us mention that in this book we are going to be executing commands by typing them (instead of using the computer mouse). Accordingly, you need to be acquainted with how to open a window (*system shell*) to type commands in your operating system. In *Linux* such a window is called a *terminal* or *console of commands*. and to open one depends on the flavor of *Linux* you are using (some standard *Linux* commands are presented in the

Appendix-A.1 of this chapter, on page 17). For instance, in *Linux Ubuntu* [`http://www.ubuntu.com/`], the flavor of *Linux* we are using, a *terminal* is opened by hitting the key **T** while keeping pressed down simultaneously the keys **CTRL** and **ALT**. From now on, we will assume you know how to open a *terminal* in your system.

1.3 Installing *Python*

As pointed out previously, *Python* is our choice of the computational language to learn how to program via the content of a standard Prealgebra course work.

As mentioned, *Python* is a **free and open source** distributed software. In case it is already available in your computer, you can go straight to verify its functionality for this course by executing the steps given in section 1.4 below.

Python can be obtained in source form via The Python Software Foundation [`https://www.python.org/`]. Installing *Python* from sources is the hard way to go and it will take sometime to do it correctly (you'll need to install *Python* this way in case you are using an operating system different from Windows, Mac, or *Linux*. As there are alternatives to have *Python* installed without restrictive licenses for educational purposes, we are skipping this way of installing *Python* directly from sources and let you on your own in case you choose to follow it.

Before continuing with any of the alternatives to have *Python* installed in your system, you need to find out whether your computer is 32 bits, 64 bits or both (in the last case assume 64 bits). One way to do that in *Linux* is by typing in a *terminal* the command (here the $ sign is the *terminal's* prompt in our computer, after which the command **`lscpu | grep CPU`** is typed, hitting **`RETURN`** or **`ENTER`** after it):

Chapter 1, System shell command 1

```
$ lscpu | grep CPU
CPU op-mode(s):   32-bit, 64-bit
CPU(s):           8
On-line CPU(s) list: 0-7
CPU family:       6
CPU MHz:          800.000
NUMA node0 CPU(s): 0-7
$
```

The output of the command (all the stuff after the line containing the $ sign) shows that the computer we are using is capable of running both 32 and 64 bit applications. Thus, we are using a 64 bit *Python*. If you are curios about this terminology, you could find more about it on the internet. A couple of sites are listed in the web reference section of this chapter, on page 20.

Continuing we our *Python* installation, one alternative to have it installed on our computer is to choose the *Python Anaconda* distribution available (for Windows, Mac, and *Linux*) at [`http://continuum.io/downloads`]. The installation instructions can be found at [`http://docs.continuum.io/anaconda/install.html`].

In short, to install the *Python Anaconda* distribution, we need to go to the *Anaconda* download page [`http://continuum.io/downloads`] and find the available version for our operating system. In our case we downloaded *Python* via the "*Linux installers*" option, for a 64 bits system of *Python*-3.x (x is a number fine tuning the current *Python* version). Two versions of *Python* are in use at the moment of writing: *Python*-2.x and *Python*-3.x. We suggest the use of *Python*-3.x, although the *Python* scripts in this book, however, will also work as given (without any changes) in both versions of *Python*. In the following internet entry you can read about the reasons for this coexistence [`https://wiki.python.org/moin/Python2orPython3`] of two versions of *Python*.

Once we have downloaded the chosen *Python Anaconda* distribution, to install it on the computer we type in a *Linux* terminal the command:

Chapter 1, System shell command 2

```
$ bash PythonAnacondaFilename
```

An screen shot of this process is as follows:

Chapter 1, System shell command 3

```
$ bash Anaconda-1.9.1-Linux-x86_64.sh

Welcome to Anaconda 1.9.1 (by Continuum Analytics, Inc.)

In order to continue the installation process, please review the
   license
agreement.
Please, press ENTER to continue
>>> (preionar ENTER o RETURN)
===================================
Anaconda END USER LICENSE AGREEMENT
===================================
...
```

```
...
...
...
Do you approve the license terms? [yes|no]
[no] >>> yes (preionar ENTER o RETURN)

Anaconda will now be installed into this location:
/home/miusuario/anaconda

  - Press ENTER to confirm the location
  - Press CTRL-C to abort the installation
  - Or specify an different location below

[/home/miusuario/anaconda] >>> (preionar ENTER o RETURN)
PREFIX=/home/miusuario/anaconda
installing: python-2.7.6-1 ...
installing: conda-3.0.6-py27_0 ...
...
...
...
...
installing: anaconda-1.9.1-np18py27_0 ...
installing: _cache-0.0-x0 ...
Python 2.7.6 :: Continuum Analytics, Inc.
creating default environment...
installation finished.
Do you wish the installer to prepend the Anaconda install location
to PATH in your /home/miusuario/.bashrc ? [yes|no]
[no] >>> yes (preionar ENTER o RETURN)

Prepending PATH=/home/miusuario/anaconda/bin to PATH in
   /home/miusuario/.bashrc
A backup will be made to: /home/miusuario/.bashrc-anaconda.bak


For this change to become active, you have to open a new terminal.

Thank you for installing Anaconda!
$
```

Notice that the installation process ends by asking if you want to have *Python Anaconda* as your default *Python*. You'll get it by answering "y" (for yes).

Another choice to have *Python* in our computer is to instal the *Enthought Canopy* distribution available at [`https://www.enthought.com/products/epd/free/`]. We let you follow the installation instructions on your own in case you choose the Canopy distribution. They are essentially the same as the one we explained for the *Python Anaconda* distribution.

Both (*Python Anaconda* and *Enthought Canopy*) distributions brings to your computer more power than require to learn how to program using *Python* via a Prealgebra course work. In particular, these distributions comes with modules for scientific computation already installed like *NumPy* [`http://www.numpy.org/`] and *SciPy* [`http://www.scipy.org/`] helpful to perform (among others) numerical computations and computational statistics; *Matplotlib* [`http://matplotlib.org/`] to satisfy graphing needs; *SymPy* [`http://sympy.org/en/index.html`] to perform symbolic computations; and *IPython* [`http://ipython.org/`] which is a nice console to execute *Python* instructions iteratively and some of its nice features will be introduced as we use the *IPython* console along the development of this book (other alternatives to *IPython* that we are not going to be using in this book, like the native *Python idle* console, are also available). Another nice environment is provided by the *Jupyter Notebook* [`http://jupyter.org/`] (formerly called the *IPython Notebook* [`http://ipython.org/notebook.html`]), which provides a web browser like presentation of the *IPython* console with some extra features.

To end this section, let's point out that in case you are unable (or do not want) to install *Python* on your computer, you still can use it over the internet. A good alternative is the *IPython* console available at [`https://www.python.org/shell/`] and the *SymPy* console available at [`http://live.sympy.org/`] (the former also has available *SymPy*). A major limitation of these two options is that you might not be able to see graphs on your computer screen generated from *Python* instructions in these consoles. Another attractive alternative (allowing the shown of graphs but requiring a good internet connection) is the *Sage Math* project [`https://cloud.sagemath.com/`].

1.4 Checking that we have what is needed for our journey with *Python* and Prealgebra

Once *Python* is installed, one should make sure it contains the required setup for this book. To do that we need to start a *terminal* (as explained in section 1.2).

In the opened terminal we will start or activate the *IPython* console, which is where we are going to type *Python* commands. This is done by executing (remember: a) the $ sign is the *terminal* prompt in our case b) hit the **ENTER** or **RETURN** key after typing the command `ipython --pylab`. Notice also the two dashes(--) before the keyword `pylab`):

Chapter 1, System shell command 4

```
$ ipython --pylab
```

After executing this command, your *terminal* window should look similar to:

Chapter 1, IPython session 1

```
$ ipython --pylab
Python 3.6.2 |Anaconda, Inc.| (default, Sep 30 2017, 18:42:57)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.1.0 -- An enhanced Interactive Python. Type '?' for help.
Using matplotlib backend: Qt5Agg

In [1]:
```

The executed command (`ipython --pylab`) activates the *Python* computational environment in the *IPython* console. The passed option --pylab load the graphing capabilities of *Matplotlib* so we can see graphs on our computer screen. If the *IPython* console does not appear on your computer screen, you need to reinstall *Python* as described earlier.

Here we need to notice the prompt **`In [1]:`**, ready to receive the first *Python* instructions that we are going to type and execute in this *IPython* console. We'll be calling the numbered **`In [n]:`** prompt an input cell **`In [n]:`**. Eventually, if required by the command or not suppressed by the user, there will also be a corresponding **`Out[n]:`** prompt that we'll be calling output cell **`Out[n]:`**. Let's advice that, in general, no space should be typed at the beginning of any *Python* instruction written on any **`In [n]:`** prompt (*IPython* already includes an space after it).

Before continuing, let's mention that once we have finished doing computations interactively in the *IPython* console, we can quit from it by typing at the input cell **quit** or **exit** and hit **RETURN** or **ENTER**. An alternative way to quit the *IPython* console is by hitting simultaneously the keys **CTRL** and **D**. The system will request confirmation for which you should type **`y`** and then hit **RETURN** or **ENTER**.

Now, the first check of our computational *Python* environment setup comes by typing on the *IPython* console the following lines of code (these are actually two *Python* instructions as indicated by the semicolon separator) and hit **ENTER** or **RETURN**:

Chapter 1, IPython session 2

```
In [1]: x = randn(10000) ; hist(x, bins=40, color='white',
   edgecolor='black') ;
In [2]:
\end{lstlisting}
% hist(x, bins=40, color='w', edgecolor='black', linewidth=1.2) ;
% x = randn(10000) ; hist(x, bins=40, color='w') ;
%\$ savefig('foo.png', bbox_inches='tight')
```

Before describing these lines of *Python* code, let's point out that the equal (=) sign in programming does not has the same meaning it has in standard mathematical operations. In a crude, not very precise form, the equal sign in programming indicates to store in the computer's memory location temporarily labeled by the name at the left side of the equal sign whatever valid expression is written to the right of the equal sign. This will make more sense to you when discussing variables in *Python* in the next chapter.

After executing the just presented sequence of *Python* instructions, in your computer screen should appear the figure 1.1, shown on page 10). Since we are checking our *Python* setup, it is not necessary to understand those *Python* instructions yet. Nevertheless, a brief description of them does not hurt.

The first line of code (or *Python* instruction) typed on the input cell **`In [1]:`** (`x = randn(10000)` defines a variable x and assign to it ten thousand random numbers drawn from a normal or Gaussian distribution (if you are unfamiliar with such terms, do not worry because they will make sense to you after covering statistics during the Prealgebra course work). Then, the following *Python* instruction (`hist(x, bins=40, color='w')`) makes a histogram plot (see figure 1.1 on page 10) of the set of data stored in the "variable" x (again, if you are not familiar with what this term, histogram, means, it will make sense to you after covering statistics in the Prealgebra course work). If everything has gone right (meaning that you get displayed on your computer screen the plot shown in figure 1.1, on page 10) congratulate yourself because you have written and successfully executed your first *Python* lines of code (program) interactively. This also means that *Matplotlib* is working as expected in your *Python* setup. If for some reason you don't get the plot of figure 1.1 on page 10, revise that you typed correctly the given sequence of commands (copying and pasting might not work correctly).

As *learning by doing* is our lemma, here comes your first practice exercise. Make a color plot of the histogram (this is exercise 1.1, on page 19). This is done by executing the *Python* instruction `hist(x, bins=40, color='g', edgecolor='k');`, on the *IPython* input cell **`In [2]:`** (if the *IPython* window is gone, just start a new one and redo what you did before). Also try writing the lines of code one per line without the semicolon at the end of each line. In addition, after executing the given lines of code, on an *IPython* input cell just type x and hit **`ENTER`** or **`RETURN`** and see what happen.

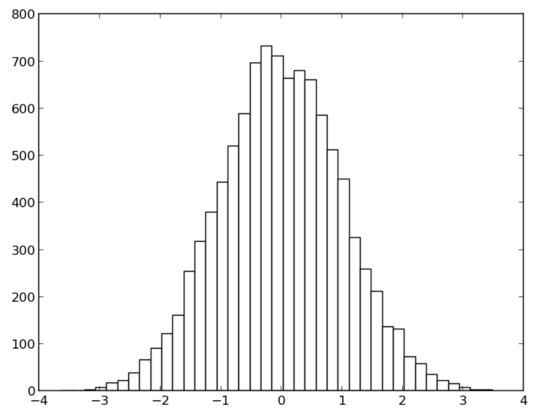


Figure 1.1: Checking out *Matplotlib*

Let's now check that *SymPy* is working as expected. For that will be computing the solution of the following set of equations, the rightness of which can easily be checked via manual computation. The exercise is to find x and y in terms of the other symbols (a, b, c, d, e, and f) considered as known.

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned} \tag{1.1}$$

In case the previous *IPython* session is still active you can type in there the following sequence of *Python* instructions (otherwise you could start a new *IPython* session, as explained earlier, and type in there the *Python* instructions that follows):

Chapter 1, IPython session 3

```
In [8]: from sympy import *

In [9]: a, b, c, d, e, f, x, y = symbols('a b c d e f x y')

In [10]: eqs = [a*x + b*y - e, c*x + d*y -f]

In [11]: eqs
Out[11]: [a*x + b*y - e, c*x + d*y - f]

In [12]: sol = solve(eqs, x, y)

In [13]: sol
Out[13]: {x: (-b*f + d*e)/(a*d - b*c), y: (a*f - c*e)/(a*d - b*c)}

In [14]: sol[x]
```

```
Out[14]: (-b*f + d*e)/(a*d - b*c)

In [15]: sol[y]
Out[15]: (a*f - c*e)/(a*d - b*c)

In [16]: sol[x].subs({a:1, b:1, c:1, d:-1, e:2, f:0})
Out[16]: 1

In [17]: sol[y].subs({a:1, b:1, c:1, d:-1, e:2, f:0})
Out[17]: 1

In [18]: sol
Out[18]: {x: (-b*f + d*e)/(a*d - b*c), y: (a*f - c*e)/(a*d - b*c)}

In [19]:
```

A brief explanation of these *Python* instruction follows. If you are typing the code, remember that after finishing typing each instruction on the *IPython* input cells `In [n]:`, one needs to hit **ENTER** or **RETURN** to execute the respective entry. The instruction on the input line `In [8]:` **from sympy import *** loads into memory of the current *Python* session the functionality (methods and/or functions) of the module *SymPy* (a better choice for doing this will be presented presented shortly and covered in deep next chapter). In the next line `In [9]:`, the variables on the left side of the equal sign are defined as symbolic variables via the *SymPy* function *symbols*. The right hand side of input line `In [10]:` set in a *Python list* (defined by the pair of square brackets $[\cdots]$) the equations to be solved separated by a comma and assigned to the variable `eqs`. Notice that only the left hand side of the equations 1.1 is given, after moving everything to that side. Input line `In [11]:` confirms the equations were stored correctly.

Continuing with the task of solving the two equations 1.1 for x and y, this is done on input line `In [12]:` via the *SymPy* function *solve*, and the result is assigned to the variable `sol` on the left hand side of the equal sign. The solution is then displayed on the output cell `Out [13]:`, and for each unknown x and y, respectively, on the output cells `Out [14]:` and `Out [15]:` (pay attention to the way of extracting them from the returned solution by *SymPy*. The rightness of the obtained solution can be verified by manual computation. That is left as an exercise to you (when covering equations in your Prealgebra course work).

Nevertheless, a simple test of the general result (that can be done via mental computations) is provided on input cells `In [16]:` and `In [17]:`, where numerical values for the constants a, b,$\cdots$, and f are given to find the solution of the simple problem of finding two numbers such that when adding them the number two is obtained, while when subtracting them the result is zero.

One can check that the answer to the posed exercise is $x = 1$ and $y = 1$, as given, respectively, on output cells `Out[16]:` and `Out[17]:`. This action of using easy to compute (special) cases is a common nice way to test the correctness of general results that we should always perform.

Finally, in output cell `Out[18]:` it is shown that the general result is still available (for extra numerical or algebraic computations) in the variable `sol`. It remains unchanged as numerical values has been assigned to the symbols a, b,$\cdots$, and f to find numerical solutions.

If you are a bit confused with what we have done, don't worry. Remember that we are just becoming familiar and checking our installation. Many of the programming terms will be fully explained in the chapters to come.

Now, even though *SymPy* has capabilities to perform numerical computations, it is not the right tool for it. *Python* includes two powerful modules specially designed to perform numerical computations efficiently. They are *NumPy* and *SciPy* (the later is dependent on the former). We will cover some of its power later on in the book. For now, let's just check their functionality by solving the system of equations 1.1 resulting after setting $a = 1$, $b = 1$, $c = 1$, $d = -1$, $e = 2$, and $f = 0$ to get the system (whose solution we already know from above):

$$\begin{aligned} x + y &= 2 \\ x - y &= 0 \end{aligned} \tag{1.2}$$

The solution of the equations 1.2 can be found by typing in an *IPython* console the following lines of code (*Python* instructions):

Chapter 1, IPython session 4

```
In [15]: import numpy as np

In [16]: import scipy

In [17]: A = np.array([[1, 1],[1, -1]])

In [18]: A
Out[18]:
array([[ 1,  1],
       [ 1, -1]])

In [19]: B = np.array([ 2, 0])

In [20]: B
Out[20]: array([2, 0])

In [21]: scipy.linalg.solve(A,B)
Out[21]: array([ 1., 1.])
```

```
In [22]: np.linalg.solve(A,B)
Out[22]: array([ 1., 1.])

In [23]:
```

On input lines `In [15]:` and `In [16]:` the functionality of the modules *NumPy* and *SciPy* are loaded in the computer memory of the current *Python* session. These are the recommended way to bring a module to the current computational *Python* environment. We'll explain it a bit further later on, but a major reason is given on input lines `In [21]:` and `In [22]:`. Both *NumPy* and *SciPy* contains functions to do the same task named the same way in both modules (in this case the function is called *solve*). By prefixing the function with the module's name we ensure we are using the function of that module. Later on will explain the customary way of importing the module *NumPy*, as given on input line `In [15]:` (by the way, the order of importing the modules is irrelevant. The only requirement is importing them before any function in there is used). The output cells `Out [21]:` and `Out [22]:` indicates we get the expected answer from both ways of finding the solution of the posed system of two equations 1.2, as obtained previously using the *SymPy* operations. This gives us some confidence that *NumPy* and *SciPy* were installed correctly.

As mentioned earlier, once we have finished doing computations interactively in the *IPython* console, we can quit from it by typing at the input cell **quit** or **exit** and hit **RETURN** or **ENTER**. An alternative way to quit the *IPython* console is by hitting simultaneously the keys **CTRL** and **D**. The system will request confirmation for which you should type **y** and then hit **RETURN** or **ENTER**.

1.5 Running *Python* packages test suite (optional)

Although the examples we just executed makes us confident that the *Python* modules of interest for this book (namely *SymPy*, *Matplotlib*, *NumPy*, *SciPy* and the *IPython* console) were installed correctly in our system, these modules came packaged each one with a test suit we should execute to have and idea what sort of issues comes with each one (keep in mind that any software system is not perfect and always has issues. Unfortunately, sometimes commercial software vendors does not fix such issues as fast as costumers might expect it to happen [`http://www.ams.org/notices/201410/rnoti-p1249.pdf`]).

When using *Python* for extensive and intensive computational work, knowing any issue reported by the modules (providing) suite tests could save us hours of pain and frustration trying to find out why something (perhaps already reported by the tests suite as such) does not behave as expected.

Here is a way to execute the tests for the basic computational modules we will using in this book. **Beware, however, that executing them could take a few long minutes to finish, depending on your computer speed**. Later on you'll learn how to write *Python* scripts so *Python* instructions (programs) can be executed non-interactively. **The tests can be executed in any order**.

In case errors and/or failures are reported at the summary given at the end, after finishing the test execution, one needs to check them and take note we are not using the respective functions in our *Python* programs.

1.5.1 Executing the *NumPy* test suite

Open a *Linux terminal* and initiates an *IPython* session. The *NumPy* test suite can be run by executing the following *Python* instructions (**avoid executing this test on the same *IPython* session where the *SciPy* test has been executed**):

Chapter 1, IPython session 5

```
$ ipython --pylab
...
...
...
In [1]: import numpy

In [2]: numpy.test("full", verbose=10)
...
...
...
Running unit tests for numpy
NumPy version 1.13.1
NumPy relaxed strides checking option: True
NumPy is installed in
    /home/myProg/Anaconda35001/lib/python3.6/site-packages/numpy
Python version 3.6.2 |Anaconda, Inc.| (default, Sep 30 2017,
   18:42:57) [GCC 7.2.
0]
...
...
...
----------------------------------------------------------------------
Ran 6832 tests in 215.953s

OK (KNOWNFAIL=7, SKIP=14)

In [3]:
```

1.5.2 Executing the *SciPy* test suite

Open a *Linux terminal* and initiates an *IPython* session. The *SciPy* test suite can be run by executing the following *Python* instructions (**avoid executing this test on the same *IPython* session where the *NumPy* test has been executed**):

Chapter 1, IPython session 6

```
$ ipython --pylab
...
...
...
In [1]: import scipy

In [2]: scipy.test("full", verbose=10)
...
...
...
Running unit tests for scipy
NumPy version 1.13.1
NumPy relaxed strides checking option: True
NumPy is installed in
       /home/myProg/Anaconda35001/lib/python3.6/site-packages/numpy
SciPy version 0.19.1
SciPy is installed in
       /home/myProg/Anaconda35001/lib/python3.6/site-packages/scipy
Python version 3.6.2 |Anaconda, Inc.| (default, Sep 30 2017,
   18:42:57) [GCC 7.2. 0]
...
...
...
----------------------------------------------------------------------
Ran 25594 tests in 612.945s

OK (KNOWNFAIL=153, SKIP=1819)

In [3]:
```

1.5.3 Executing the *SymPy* test suite

Open a *Linux terminal* and initiates an *IPython* session. The *SymPy* test suite can be run by executing the *Python* instructions:

Chapter 1, IPython session 7

```
$ ipython --pylab
...
...
...
In [1]: import sympy

In [2]: sympy.test()
...
...
...
------------------------------------------------------------------------
 tests finished: 6984 passed, 1 failed, 193 skipped, 348 expected to
    fail,
11 expected to fail but passed, in 1677.08 seconds
DO *NOT* COMMIT!

In [3]:
```

1.6 Chapter Summary

In this chapter you had a lot of fun getting the *Python* environment ready to start the business of crunching numbers on your computer. A major step was to install the *Anaconda* and/or *Canopy Python* distribution. Then we executed some basic tests checking the functionality of the installation.

In the next chapter we will start using this *Python* environment to help you enhance your knowledge of computing with whole numbers. For that, you will start writing small *Python* programs, right away after working out with us some preliminaries steps on *Python* programming presented in the chapter, including defining *Python* variables and *Python list* objects, and going to basic implementations of the *for* and *while Python* loops which are the standard tools to perform repetitive task in *Python*.

Appendix of Chapter 1

A.1 Some *Linux* commands

As you are already familiar with it, the *Linux terminal* is a window (console) that allows the typing of commands to interact with the computer. In the *Ubuntu* flavor of *Linux*, a *terminal* is opened by hitting simultaneously the key sequence **`CTRL-ALT-T`** (other *Linux* systems has different sequences of keys). A brief list of *Linux* commands that are available in any *Linux* flavor is listed below (they also work in any *Unix terminal* like the Mac *terminal*) :

`cat`: used to dump the contents of a file (by default) to the terminal. It is invoked like **`cat file(s)name`**.

`cd`: used to change between directories. It is invoked like **`cd directory-name`**.

`cp`: used to make a copy of a file to a file with a new name or with the same name if the copy goes to a different directory where the copied file resides. It is invoked like:

```
cp filename new-filename
cp file(s)name directory-filename-path
cp filename directory-filename-path/new-filename.
```

`cp -r`: used to make a copy of a directory. It is invoked like
`cp -r directory-name new-directory-name`.

`chmod -R`: used to change permissions to files or directories. Its basic usage goes like **`chmod -R u+r file-or-directory-name`**.

`chown`: used to change the owner of a file or directory. Its basic usage goes like **`chown new-user file-or-directory-name`**.

`exit`: used to close a *terminal*. It is invoked like **`exit`**.

`file`: used to know the type of a file. It is invoked like **`file filename`**.

`gzip`: used to pack files in the zip format. Its basic usage goes like **`zip file.zip file(s)name`**.

`ls`: used to list the files in a directory. Its basic usage goes like **`ls`**.

`locate`: used to find files in the system. It is invoked like **`locate filename`**.

`man`: used to show on screen a basic help of a command. It is invoked like **`man command-name`**.

`mkdir`: used to create a directory. It is invoked like **`mkdir directory-name`**.

`more`: used to show the contents of a file to the *terminal* screen by screen (meaning that that a large file it shows in the terminal in small pieces of the screen size. The *space bar* needs to be hitting to go between screens). It is invoked like **`more filename`**.

`mv`: used to change a file or directory name to a new one or to move a file or directory to an existing directory. Its basic usage goes like
`mv filename new-filename-or-existing-directory-name`.

`pwd`: used to show the current directory path. It is invoked like **`pwd`**.

`rm`: used to delete files or non empty directories. Its basic usage goes like
`rm -rf file-or-directory-name`.

`rmdir`: used to delete directories. It is invoked like `rmdir directory-name`.

`unzip`: used to unpack zip files. Its basic usage goes like **`unzip full-zip-filename`**.

Exercises of Chapter 1

Exercise 1.1 *Following the instructions given on page 9, make a color graph of the figure 1.1, on page 10.*

References of Chapter 1

Books and/or Articles

- **Marecek, L. and Smith, M. A.** (2017). Prealgebra, Rice University, OpenStax `https://openstax.org`.
  Book available for free at: `http://cnx.org/content/col11756/1.9`

- **Tollervey, N. H.** (2015). Python in Education Teach, Learn, Program, O'Reilly.
  `http://www.oreilly.com/programming/free/python-in-education.csp`

- **Developping high-order thinking skills**:
  Reif, F. and Scott, L. A. (1999) Teaching scientific thinking skills: Students and computers coaching each other. *American journal of physics*, **67** (9), 819--831.
  Reif, F. (2008) Applying Cognitive Science to Education. Thinking and Learning in Scientific and Other Complex Domains. MIT Press.
  Polya, G. (1988) How to solve it. A new aspect of mathematical method. Expanded edition, Princeton University Press.
  Schoenfeld, A. H. (1994) Mathematical thinking and problem solving. Taylor & Francis. (See essay by Andrea A. diSessa on page 248.)
  Rojas, S (2012) Enhancing the process of teaching and learning physics via *dynamic problem solving strategies*: a proposal. *Revista Mexicana de Física E*, **58** (1), 7--17. (Freely available at `http://rmf.fciencias.unam.mx/pdf/rmf-e/56/1/56_1_022.pdf`).
  Rojas, S (2010) On the teaching and learning of physics problem solving. *Revista Mexicana de Física*, **56** (1), 22--28. (Freely available at `http://rmf.fciencias.unam.mx/pdf/rmf-e/56/1/56_1_022.pdf`).
  Rojas, S (2008) On the need to enhance physical insight via mathematical reasoning. *Revista Mexicana de Física E*, **54** (1), 75--80. (Freely available at `http://rmf.fciencias.unam.mx/pdf/rmf-e/54/1/54_1_075.pdf`).

References on the WEB

- **Pedagogical Aspects of Computational Thinking**:
  `http://nap.edu/12840`

http://www.nap.edu/catalog.php?record_id=13170
http://bit.ly/1T7O2Tg
http://bit.ly/1T9iGZt

- **Top programming languages research studies**:
  https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages
  https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities
- **Operating systems: 32-bit-vs-64-bit**:
  https://www.digitaltrends.com/computing/32-bit-vs-64-bit-operating-systems/
  https://www.pcmag.com/article/350934/32-bit-vs-64-bit-oses-whats-the-difference
- **Automate the Boring Stuff with Python**:
  https://automatetheboringstuff.com

2

Whole numbers in *Python*

"Learning results from what the student does and think and only from what the student does and think. The teacher can advance learning only by influencing what the student does to learn."

Herbert Simon
Nobel Laureate in Economic Sciences (1978)

2.1 Introductory remarks

The study of whole numbers (or non-negative integers, including zero) offers a natural environment to start the use of *Python*, first using it as an standard calculator to perform basic numerical operations with whole numbers (this is done in section 2.3, starting on page 24) and then, at a more sophisticated level, while finding the answer to a problem involving a big whole number obtained via an story related to the chessboard (see section 2.6, starting on page 31), we will learn basic *Python* programming notions which will allow you to start writing small programs in *Python* to perform repetitive tasks.

This will happen after introducing the basic and important programming notions of *loop* execution (see section 2.7, starting on page 34). After that, we will be presenting some aspects of the *Python* syntax that we need to know from the very start of using the language to avoid unnecessary, annoying frustrations.

Accordingly, after finishing this chapter, you'll be equipped with some components of the minimal basic set of *Python* tools to fully explore in your computer (via the exploration of routine and non-routine problems) many of the subjects (not only whole numbers) that you will be studying in your prealgebra course work.

Of course, additional important *Python* programming ideas will be presented in the following chapters of this book, in the context of the respective chosen prealgebra topic (as in this chapter the topic of whole numbers was chosen).

Let's recall that *Python* is a (scripting) open-source programming language having the functionality required to any modern programming language: it has an efficient **high level data structure** and allows **object oriented programming**. This remarks might not mean much to anyone starting to program, but it is good to know from the start that we are making a good use of our time by learning to program in a long lasting programming language, that since very recently has been adopted as a common programming language at top US Universities [https://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-u-s-universities].

In addition to the simplicity and elegance of its syntax, *Python* provides in one environment the ability to perform numerical and symbolic (algebraic) computation as well as data visualization or graphing capabilities. Moreover, *Python* codes are portable to practically any existing operating system (Windows, Mac OS X, *Linux*, Unix, etc.). This makes *Python* a very cost-efficient option to develop computational literacy in school, colleges and universities.

Before starting the prealgebra topic of this chapter, we want to mention some extra words on the *IPython* console that we are going to be using to execute *Python* instructions (code).

2.2 Starting the *IPython* console

As we have already mentioned, in this book we will use the *IPython* console in order to execute *Python* instructions interactively (that is the meaning of the *I* in *IPython*). A more sophisticated alternative is the Jupyter Notebook, but we are not using it in this book. Also, as our command of *Python* develops, we will learn how to write *Python* instructions (code) in a file (called *Python* script) and execute them directly from the terminal, non-interactively.

If installed correctly, the *IPython* console can be started by opening a terminal or system shell (which in *Linux* Ubuntu is done by hitting simultaneously the keys **CTRL-ALT-T**) and then typing on it the instruction (or command):

Chapter 2, System shell command 1

```
$ ipython --pylab
```

after which we need to press the **ENTER** or **RETURN** key. This sequence of actions will present in the same terminal or system shell an output similar to:

Chapter 2, IPython session 1

```
$ ipython --pylab
Python 3.6.2 |Anaconda, Inc.| (default, Sep 30 2017, 18:42:57)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.1.0 -- An enhanced Interactive Python. Type '?' for help.
Using matplotlib backend: Qt5Agg
```

```
In [1]:
```

Notice the `In [1]:` prompt. This is where we can start interacting with *Python* by typing lines of code or instructions, as we will be doing shortly. Keep in mind that every typed instruction is executed after you hit **RETURN** or **ENTER**.

Before continuing, let's mention that the instruction `--pylab` is optional, meaning that it is not required to include it when initiating an *IPython* console. This option is useful to display graphs on a separated window. Some other features are initialized, but they are not of interest to us in a basic usage of the *IPython* console.

We will be covering some features of the *IPython* console as they are needed. In case you would like to jump ahead, you could read the firsts chapters of the *IPython* book by Rossant, listed in the reference section for this chapter, starting on page 64. For now let's mention in an *IPython* console one could use the computer's keyboard up and down arrow keys to go through the input history of instructions already entered so you could revise and re-enter them. If you start typing before pressing the arrow keys, only the commands that match what you have typed so far will be shown.

To exit or leave the *IPython* console just type *exit* or *quit* and hit **RETURN** or **ENTER**. Alternatively, you could hit simultaneously **CTRL-D** and then typing **y** (for yes) and hit **RETURN** or **ENTER**.

2.3 Computing with Whole Numbers in *Python*

As learned in your class about whole numbers, these numbers are written down using the ten digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Such set of digits are part of the alpha-numeric characters labeling the keys in your computer's keyboard. Also you learned that large whole numbers are represented in standard form using combination of these ten numerals grouped in sets of three digits (called periods) separated by comma to indicate the named place value of each numeral, as for example $1,234,567,890$ which is read *one billion, two hundred thirty-four million, five hundred sixty-seven thousand, eight hundred ninety.*

As you might already know, in the computer numbers are represented without the comma representing the place value of the digits. The only character allowed when writing numbers in the computer is the period (.) representing decimals and a (preceding) dash (-) representing negative numbers (both topics are covered when studying integers and real numbers, later on in your prealgebra course work). Whole numbers does not have such a period as they do not contain any decimal, neither they are written preceded by the dash as they can not be negative. Thus, 1234567890 is the way to write the previous example of a whole number in the computer.

At this point one should also mention that the equal sign (=) in programming does not have the same meaning as you learned in your prealgebra course work. We will present the meaning of the equal sign in *Python* programming shortly, when studying *variables* in *Python*. An equality, represented by the equal (=) sign in your prealgebra course work, is obtained in *Python* programming by using a double equal sign (==). This will be discussed when studying boolean operations in *Python* and equations via the *SymPy* module.

2.3.1 Basic Whole Numbers operations in *Python*

Standard operations of Addition, Subtraction, Multiplication, Exponentiation, and Division can be readily executed in *Python* as we can do them in a calculator, with the added advantage that in *Python*, when using whole numbers, they can be performed with unlimited digits (precision).

Addition: the symbol +, as in your prealgebra course work and in your calculator, is used to perform the addition operations in *Python*. Type the following example to add 3 and 5 in the just opened *IPython* session (don't forget to hit return after finishing your typing):

Chapter 2, IPython session 2

```
In [1]: 3+5
Out[1]: 8
```

This operation can be done the other way around $(5 + 3)$. We let you do it as a right away exercise and check the result.

Subtraction: the symbol −, as in your prealgebra course work and in your calculator, is used to perform the subtraction operations in *Python*. Type the following example to subtract 2 from 6 in the current *IPython* session:

Chapter 2, IPython session 3

```
In [2]: 6-2
Out[2]: 4
```

This operation can be done the other way around $(2 - 6)$. We let you do it as an straightaway exercise and check the result.

Multiplication: the symbol ∗, as in your calculator, is used to perform the multiplication operations in *Python*. Type the following example to find the product of 2 and 7 in the current *IPython* session:

Chapter 2, IPython session 4

```
In [3]: 2*7
Out[3]: 14
```

This operation can be done the other way around ($7 * 2$). We let you do it at this instant as a quick exercise and check the result.

Exponentiation: the symbol $**$ is used to perform exponentiation (repeated multiplication) operations in *Python*. Type the following example to find two (2) to the third (3rd) power in the current *IPython* session (we need to be careful in extreme when writing exponent operations. The mistake of writing $*$ (multiplication) instead of $**$ (exponent) could take some time to find out in large expressions):

Chapter 2, IPython session 5

```
In [4]: 2**3
Out[4]: 8
```

In general, x to the power of y (x^y) is written in *Python* as $x**y$.

Division: *Python* version 3 (the one we are using in writing this book) contains two symbolic operators / and // to perform the division operation. Either of the symbols separates the numbers in the same way (as you learned in your prealgebra course work) does the symbol ÷ (the dividend goes to the left of the symbol while the divisor goes to the right of the symbol). Type the following example to find the quotient of 6 divided by 2 in the current *IPython* session:

Chapter 2, IPython session 6

```
In [4]: 6/2
Out[4]: 3.0

In [5]: 25/7
Out[5]: 3.5714285714285716

In [6]: 6//2
Out[6]: 3

In [7]: 25//7
```

```
Out[7]: 3
```

In the case of exact division, shown in the output cell `Out[4]:` when using the symbol / to execute the division, *Python* behaves the same way as does your calculator by adding to the quotient (or result of the operation) the extra character .0 (in some other cases it can adds more extra digits as a result of numerical round off errors). In this case of exact division, as shown in the output cell `Out[6]:`, the result does not contain the extra character .0, as correspond to a whole number. Nevertheless, as you learn in your prealgebra course work real numbers (of which the whole numbers are a subset), we will be using the symbol / as the standard division symbol. The symbol // is used to obtain the **quotient** of a division, without information of the **remainder** which can be obtained by other means.

As a right away exercise, try the division examples just given the other way around, interchanging dividend and divisor. In case you are using *Python* version 2 the result will be surprising. To get the same (mathematically correctly) result as given by *Python* version 3, you need to replace either 6 by 6.0 or 2 by 2.0 or do both. Another exercise for you to do at the moment is to execute any division by zero (like 6/0). Take note of the output given by *Python* (we will talk about it at the end of this chapter).

In summary, we have introduced the symbols $+$, $-$, $*$, $**$, and / as operators to perform in *Python* the basic operations of addition, subtraction, multiplication, exponentiation, and division respectively with any set of *Python* objects (not only whole numbers) on which such operations have meaning. A few extra words on the use of the operator // for division will be given in the next section.

2.4 Variables in *Python*

The issue with the symbols / and // as way to execute division operations in *Python*, allow us to introduce the notion of *variables* in *Python*.

Let's first recall that when a division is not exact (the **remainder** is not zero) we can find the **quotient** of the division of the two whole numbers by using // as the division symbol instead of /. The following example of finding the quotient of 25 and 7 illustrates this operation:

Chapter 2, IPython session 7

```
In [1]: 25//7
Out[1]: 3
```

Now to find the remainder of any division we can use the % operator, as follows:

Chapter 2, IPython session 8

```
In [2]: 25%7
Out[2]: 4
```

This result can be checked by using the fact that for any division, it must happen that the **dividend** (in this case 25) must be equal to the **remainder** (in this case 4) plus the product of the **quotient** (in this case 3) and the **divisor** (in this case 7). That is, **dividend** = **remainder** + **quotient** ∗ **divisor** (in numbers $25 = 4 + 3 * 7$). We can set these operations in *Python* as follows:

Chapter 2, IPython session 9

```
In [3]: dividend = 25

In [4]: dividend
Out[4]: 25

In [5]: divisor = 7

In [6]: divisor
Out[6]: 7

In [7]: quotient = dividend//divisor

In [8]: quotient
Out[8]: 3

In [9]: remainder = dividend % divisor

In [10]: remainder
Out[10]: 4

In [11]: result = remainder + quotient * divisor

In [12]: result
Out[12]: 25

In [13]: result - dividend
```

```
Out[13]: 0

In [14]:
```

Let's explore in words what we have done. On input cell `In [3]:` the variable *dividend* is defined and set to whole the value 25. The equal (=) sign in many programming languages means to assign to the variable name to the left of the sign (in this case *dividend*) whatever expression is given to the right of it (in this case the value 25). To check what is contained in a variable, we type the variable's name on an *IPython* input cell and hit **ENTER** or **RETURN** (this is done on input cell `In [4]:` and the result is shown on the corresponding output cell `Out[4]:`).

In standard *Python*, variables must have been defined (assigned a valid value, not necessarily a number) before they can be used. This is illustrated on the input cells `In [7]:`, `In [9]:`, and `In [11]:` on which the name of variables appear on the right hand side of the equal sign. Notice that on the right hand side of the equal sign could appear any valid set of operations or expressions that *Python* first evaluates and assign the result to the variable in the left hand side of the equal sign. The output cells `Out[12]:` and `Out[13]:` confirms that the **dividend** is rightly defined by the expression assigned to the variable *result* on the input cell `In [11]:`.

The name of a variable in *Python* must start with any (upper or lower case) alphabetical character or the underscore symbol (_) and could contain any combination of alpha-numeric characters and the underscore (_) symbol (though valid, it is customary not to start our variable names with the underscore symbol). Thus, *a01_23*, *A01_23*, *_A01_23* (not recommended), *Iam*, and *myname* are all valid variables names. The name of variables should be chosen so we can have a sense of what they represent. A very long variable name is cumbersome to use, while names with non sense are difficult to remember what they represent.

The use of properly chosen variables names allow us to focus attention on organize our thoughts on solve what we are trying to accomplish instead of wasting time understanding how we are doing it because of a wrong choice of variable names.

More importantly, the use of variables is what allows the craft of programming.

2.4.1 Reserved words in *Python*

Python, and any other computer language, has a set of reserved words that can not be used as name of variables. A partial list of such words is shown in Table 2.1:

and	as	assert	break	class	continue	def	del	elif
else	except	exec	finally	for	from	global	if	import
in	is	lambda	not	or	pass	print	raise	return
try	while	with	yield					

Table 2.1: A partial listing of reserved words in *Python*

A completed list of reserved words in *Python* can be obtained by executing in the *IPython* console the following sequence of *Python* instructions:

Chapter 2, IPython session 10

```
In [1]: import keyword
In [2]: keyword.kwlist
```

Try it in your system and see what you get. Also try assigning a value to any of such keywords (i. e. try doing, $else = 9$).

2.5 Grouping basic Whole Number operations in *Python*

Now that we have introduced basic arithmetic operations with whole numbers in *Python*, we can use round parenthesis $(\cdots)$ to group complex computational operations in order to make clear or explicit the order or precedence we want them to be performed. Let's make emphasize that what follows is also valid not only to whole numbers but also to the wider mathematics set of numbers defined in *Python*.

Following with the *IPython* session (or in a new one), let's try the following computation:

Chapter 2, IPython session 11

```
In [7]: ((2 + 7*(234-15)+673)*775)//(5+10+(4+1)*5)
Out[7]: 42780
```

The *Python* computational engine perform first divisions, followed by multiplication and then by sum/subtraction operations. By using parenthesis we tell *Python* to perform first arithmetic operations enclosed in parenthesis. This is a good practice that helps to avoid miss-arrangements of terms. We tell explicitly to *Python* the order in which we want executed the arithmetic operations in an expression.

We can also use extra white spaces in between operations to clarify the operations:

Chapter 2, IPython session 12

```
In [8]: ( (2 + 7*(234 - 15) + 673)*775 )//( 5 + 10 + (4+1)*5 )
Out[8]: 42780
```

In case the computational expression we are writing is too large, we can span it over the next line by using the back slash symbol (\) to tell *Python* that the expression continues in the next line. *IPython* shows it by a symbol like (...:). Some examples are as follows:

Chapter 2, IPython session 13

```
In [9]: ( (2 + 7*(234 - 15) + 673)*775 )//( 5 \
  ...: + 10 + (4+1)*5 )
Out[9]: 42780

In [10]: ( (2 + 7*(234 - 15) + 673)*775 )//( 5 \
   ...:             + 10 + (4+1)*5 )
Out[10]: 42780

In [11]: ( (2 + 7*(234 - 15) + 673)*775 )// \
   ...: ( 5 + 10 + (4+1)*5 )
Out[11]: 42780
```

2.6 The wheat problem: a computational example involving a big Whole Number

At this point it is time to illustrate the computational capabilities of *Python*. We'll do that computing the answer to a problem involving the chessboard, which is given by means of a illuminating story about numbers. To read the full story refers to chapter 16 (*The game plan* of the book *The man who counted*, listed on the reference section for this chapter on page 64.

The story goes as follows. A king wanted to reward a man, Lahur Sessa, who introduced him to the game of chess. Lahur Sessa refused to receive any reward, but after the king's insistence he made the following request to the king:

"... I do not wish either gold or lands or palaces... I want my reward in grains of wheat. You give me one grain of wheat for the first square on the board, two for the second, four for the third, eight for the fourth, and so on, doubling the amount with each square up to the sixty-fourth and last square on the board. I beg you, O King, in accordance with your magnanimous offer, to pay me in grains of wheat in the manner I have indicated."

After a while, the king's wisest mathematicians told him:

"Magnanimous King! We have calculated the number of grains of wheat, and we have reached a sum that is beyond human imagination. With the greatest care, we have calculated the number of ceiras required to hold the appropriate quantity of wheat, and we have arrived at the following conclusion: the wheat that you will have to give to Lahur Sessa is the equivalent of a mountain with a diameter at its base the size of the city of Taligana and a height ten times greater than that of the Himalayas. If all the fields of India were sown with wheat, in two thousand centuries you would not harvest what you have promised young Sessa. "

The number contains 20 digits and it is $18,446,744,073,709,551,615$ (one easy way to obtain such a huge number is by subtracting one from the power 64 of two or $(2^{64} - 1)$. Perhaps you do not understand it yet, but a proof of why is so is given on the Appendix of this chapter, starting on page 59.

Let's first compute it in *Python* following the straightforward instructions given by Sessa to the king (for this you need to know that the chessboard is divided in 64 little squares):

1. For the first (1st) square, Sessa should receive one grain of wheat: $(1 = 2^0 = 2^{1-1})$.
2. For the second (2nd) square, Sessa should receive twice grains of: wheat as received in the previous (last) square: $(2 \times 1 = 2^1 \times 2^0 = 2^{1+0} = 2^1 = 2^{2-1})$
3. For the third (3rd) square, Sessa should receive twice grains of wheat as received in the previous (last) square: $(2 \times 2^1 = 2^1 \times 2^1 = 2^{1+1} = 2^2 = 2^{3-1})$
4. For the fourth (4th) square, Sessa should receive twice grains of wheat as received in the previous (last) square: $(2 \times 2^2 = 2^1 \times 2^2 = 2^{1+2} = 2^3 = 2^{4-1})$
5. For the fifth (5th) square, Sessa should receive twice grains of wheat as received in the previous (last) square: $(2 \times 2^3 = 2^1 \times 2^3 = 2^{1+3} = 2^4 = 2^{5-1})$
6. $\cdots$
7. Hopefully you have captured the pattern of $2^{\text{Number of the square}-1}$
8. $\cdots$
9. For the sixty-fourth (64th) square, Sessa should receive twice grains of wheat as received in the previous (last) square: $(2 \times 2^{62} = 2^1 \times 2^{62} = 2^{1+62} = 2^{63} = 2^{64-1})$

To find the total grains of wheat that Sessa should have received we need to add the amount received for each square. This done in *Python* as follows:

Chapter 2, IPython session 14

```
In [14]: 2**0+2**1+2**2+2**3+2**4+2**5+2**6+2**7+2**8+2**9+2**10 + \
   ....: 2**11+2**12+2**13+2**14+2**15+2**16+2**17+2**18+2**19+2**20
      + \
   ....: 2**21+2**22+2**23+2**24+2**25+2**26+2**27+2**28+2**29+2**30
      + \
   ....: 2**31+2**32+2**33+2**34+2**35+2**36+2**37+2**38+2**39+2**40
      + \
   ....: 2**41+2**42+2**43+2**44+2**45+2**46+2**47+2**48+2**49+2**50
      + \
   ....: 2**51+2**52+2**53+2**54+2**55+2**56+2**57+2**58+2**59+2**60
      + \
   ....: 2**61+2**62+2**63
Out[14]: 18446744073709551615

In [15]: 2**64-1
Out[15]: 18446744073709551615

In [16]: Out[14]-Out[15]
Out[16]: 0
```

To be sure that we have added correctly the terms for each square (typing all of them is prone to error), the result is redone by computing $(2^{64}-1)$. This is shown in the output cell `Out[15]:`. Here we introduce another nice feature of the *IPython* console. The cell names are variable names. Thus, to verify that the output of cells `Out[14]:` and `Out[15]:` are really the same (instead of a visual inspection comparing the digits one by one) we can differentiated them, as has been done in the input cell `In [16]:`. As expected, the result is zero as shown in the corresponding output cell `Out[16]:`.

As already mentioned, a proof that the addition $2^0+2^1+2^2+\cdots+2^n = 2^{n+1}-1$, for any whole number (integer) $n \geq 0$ is given in the Appendix of this chapter, starting on page 59. You can also recheck this result consulting the Wikipedia entry for this problem listed on the web references section for this chapter, on page 64. A good solving problem strategy is to always find alternative ways to verify the rightness of any obtained result. In the next section we will show alternative, more efficient ways to perform the computation of this exercise.

Let's end this section by trying to get an idea of how big is the number $18,446,744,073,709,551,615$ obtained as the answer of this exercise. Such a number can be compared with the number $10^{20} = 100,000,000,000,000,000,000$ (a one followed by twenty zeros), which is about five times the former (as you can verify by finding the quotient of the division of the later number by the former one).

If we considers that the heart beats of a healthy adult is roughly $100,000$ per day, the number

10^{20} is obtained by computing the daily heart beats of one thousand trillion $(1,000,000,000,000,000)$ people. Do we have that number of people in the world today? Certainly the number $18,446,744,073,709,551,615$ is really a big one. In *Python* computing with big whole numbers (or integers in general) is done via a special method called *extended integer precision*. Check about it on the Internet.

We let you as an exercise to execute the above sum by writing 2.0^2 or $2.^2$ in one of the terms. Is there any difference with the already known answer?

2.7 Repetitive computations in *Python*

An straight-forward computation in finding the answer to our exercise of the wheat problem involves the typing on many terms that follow a pattern. The involved amount of typing is prone to error, leading to obtaining a wrong answer. Fortunately, after finding our answer we showed ways to check the obtained answer was the right one. The fact that there is a pattern in the typed terms allows the natural introduction of repetitive computation in *Python*. Previous to that we will need to introduce the notion of *Python* list and *Python relational* operators. Then will be ready to present two ways of looping in *Python*: the *for* and the *while* loop.

2.7.1 *Python* List

An important basic *Python* type object is named *list*, which is an object that can contain zero or more objects in an ordered way separated by comma and enclosed by square brackets:

$$[obj_0, obj_1, \cdots, obj_n]$$

Also, like any other *Python* object, a *list* can be assigned to a variable, and we will be doing so:

$$\text{milista} = [obj_0, obj_1, \cdots, obj_n]$$

Here, the *list* object to the right of the equal sign was assigned to the variable *milista*. The *list* elements are numbered left to right starting from zero, and the way to access any of them is via a pattern containing the name of the variable referencing the *list* followed by a set of squared brackets enclosing the number of the element we ant to list (following our example, the element n is listed like `milista`[n]). Another way to referencing the elements of a list is by using negative integers from $-n$ to -1. Extra details operating with *list* can be found in the documentation [`https://docs.python.org/3/tutorial/datastructures.html`].

Since the only *Python* objects we know so far are numbers (or variables referencing assigned numbers), so in our first encounter with *list* objects they will contain whole numbers (but the elements of a *Python list* can be objects of any type). Take a look at the following *IPython* session illustrating the notion of *list* and how to work with them (in following this example you don't need to type anything after and including the character numeral or hash-tag (#). In *Python*, that character is used to write comments which are ignored by the *Python* computational engine, though it does not hurt if you type everything. Comments are used to document codes. Other ways to write comments will be presented along the development of this book):

Chapter 2, IPython session 15

```
In [1]: # anything written after the numeral (hash-tag) character is a comment

In [2]: [3,4,6,8,1] # Here a list if defined
Out[2]: [3, 4, 6, 8, 1]

In [3]: milist = [3,4,6,8,1] # It is better assign the list to a variable

In [4]: type(milist) # type is a python function showing the object type
Out[4]: list

In [5]: milist[0] # shows the first element in object list milist
Out[5]: 3

In [6]: milist[0] = 34 # replace the first element in object list milist

In [7]: milist # shows the content of milist
Out[7]: [34, 4, 6, 8, 1]

In [8]: len(milist) # the Python function len gives the number of
               # elements in list-type objects
Out[8]: 5

In [9]: range(len(milist)) # the Python function range(n) creates a
                      # list of integers from 0 to n-1
Out[9]: range(0, 5)

In [10]: list(Out[9]) # the Python function list shows explicitly
                # the list assigned to Out[9]
Out[10]: [0, 1, 2, 3, 4]

In [11]: mylist = milist # Another name to milist.
                    # WRONG way of making a copy of milist

In [12]: mylist
Out[12]: [34, 4, 6, 8, 1]

In [13]: CopyOfmilist = list(milist) # RIGHT way of making a copy of milist
In [14]: CopyOfmilist # shows the content of CopyOfmilist
Out[14]: [34, 4, 6, 8, 1]

In [15]: milist[3] = 104 # modify the third element in milist
                      # (it changes 6 by 104)

In [16]: milist # shows the content of milist
Out[16]: [34, 4, 6, 104, 1]

In [17]: mylist # shows the content of mylist
Out[17]: [34, 4, 6, 104, 1]

In [18]: CopyOfmilist # shows the content of CopyOfmilist. It is unchanged
Out[18]: [34, 4, 6, 8, 1]

In [19]: mylist.remove(4) # way to remove an element from a list

In [20]: mylist # shows the content of mylist just modified
Out[20]: [34, 6, 104, 1]

In [21]: CopyOfmilist[-1] # shows last element of CopyOfmilist
Out[21]: 1

In [22]: CopyOfmilist[-3] # shows third to last element of CopyOfmilist
Out[22]: 6

In [23]: CopyOfmilist.append(300) # A way to append an element to a list

In [24]: CopyOfmilist # shows the content of CopyOfmilist just changed
Out[24]: [34, 4, 6, 8, 1, 300]

In [25]: newlist = mylist + CopyOfmilist # Concatenating lists via another
                                      # use of the + operator and
                                      # assign it to the variable newlist

In [26]: newlist # shows the content of newlist
Out[26]: [34, 6, 104, 1, 34, 4, 6, 8, 1, 300]

In [27]: 2*mylist # concatenating the list with itself
Out[27]: [34, 6, 104, 1, 34, 6, 104, 1]
```

When working with *list*, make sure to use the right way of making a copy of it (an alternative to the way shown on the input cell `In [13]:`, is to apply the method *copy* to the list as in `CopyOfmilist = milista.copy()`). This way, changes make to any of the *list* objects are not transmitted to the other. To find out other methods that can be applied to a *list*, type in an *IPython* input cell (after a *list* has been defined) the name of the *list* appending a dot to it (like `milista.`) and hit the keyboard **TAB** key.

There are many other operations that can be done with *Python list* objects that are outside the scope of this book. You are invited to read references listed at the end of this chapter, on page 64.

2.7.2 The wheat problem via the *Python for loop*

To find the answer of the wheat problem we need to implement the addition $2^0+2^1+2^2+\cdots 2^{63}$, the terms of which follows the pattern of two to the power of each one of the terms in the *list* $[0, 1, 2, 3, \cdots, 63]$. We know how to create such a *list* via the *Python* instruction `powers = range(n)` with $n = 64$. We also know how to extract the elements of the *list* via the instruction `powers[m]`, with m the indexing of the elements in the *list* which start from zero, as the power of two in our addition operation. What we are missing is a procedure to iterate over the elements of the *list* and use them as powers of two to be added. Such iterative procedure can be think of as the execution of a repetitive task. For that *Python* offers two basic looping operations the *for loop* (discussed in this section) and the *while loop* (discussed in the next section). Both loops are used to repetitively execute a sequence of *Python* instructions while changing a variable from iteration to iteration.

The *for loop* [`https://wiki.python.org/moin/ForLoop`] has the standard operating form:

```
for iterator in data:
    Body or indented set of Python instructions
```

In the first line of code, **`for`** and **`in`** are *Python* reserved keywords, while *iterator* and *data* are variables defined by the user (programmer). The colon (`:`) at the end of *data* is a required sign, as is also required the indentation spaces in the next line, where `Body or set of Python instructions` of the loop goes (each line that belongs to the body of the loop must be indented the same way. The number of indentation spaces is at least one. It is customary to use a minimum of four spaces).

The *iterator* is usually a single variable, while *data* must be a *Python* objects allowing iterations on it (as is the case of a *list* object).

Using the aforementioned elements, the summation in our wheat exercise could be implemented in the following lines of code (when typing in your *IPython* console the lines of code of input cell `In [23]:`, do not type the dots `...:` starting the lines below `In [23]:`. Once you hit return after typing the first line of code ending with the colon (**:**) *IPython* will type for you these dots and will even add the indentation spaces for the next line of code):

Chapter 2, IPython session 16

```
In [20]: n = 64

In [21]: powers = range(n)

In [22]: lasuma = 0

In [23]: for j in powers:
    ...:     lasuma = lasuma + 2**j
    ...:

In [24]: print(lasuma)
18446744073709551615

In [25]: lasuma
Out[25]: 18446744073709551615

In [26]:
```

A description of these lines of code is as follows: on input cell `In [20]:` the variable n is assigned the value 64, which then is used to build the iterating object `range(n)` assigned to be referenced by the variable `powers`. Then the variable `lasuma` is assigned the value zero (this can be rephrased as "initializing to zero the variable `lasuma`"). As iterator of our *for loop* (on input cell `In [23]:`) we use the variable `j`. Then the body of the *for loop*, executed sixty-four times (from $j = 0$ to $j = 63$), is defined by the indented line containing `lasuma = lasuma + 2**j`. This line of code is executed as follows: takes the power j of two, add it to the current value of the variable `lasuma`, and assign the result to the variable `lasuma` (this could be rephrased as "updating the variable `lasuma` with the new value"). This last action is repeated 64 times. The first time, the `j-iterator` takes the value of zero (the first entry of the list `powers`), and the variable `lasuma` is updated to $0 + 2^0 = 0 + 1 = 1$; the second time, the `j-iterator` takes the value of one (the second entry of the list `powers`, and the variable `lasuma` is updated to $1 + 2^1 = 1 + 2 = 3$; and so for (for a visual inspection of these facts, see exercise 2.5, on page 62).

Finally, the lines of code ends on input line (`In [24]:`), printing on screen the value on the variable `lasuma` after exiting the *for loop* (non indented statements are not part of the body of the loop). Notice that in writing that line of code, we have explicitly instructed *Python* to show the result on the screen by passing to the *Python print* function the variable *lasuma*. We will discuss this function in later chapters. For another way of using it, see exercise 2.5, on page 62. (input line `In [25]:` is another (preferred) way of printing on screen interactively).

Now, for completeness, let's write these lines of code to a file, just to make it our first formal *Python* code. We will start by recalling that the lines of code we just wrote (on page 38) are a bit complex compared with what we had written before. They allow us to show another advantage of using the *IPython* console by writing them to a file. For that will be using the *IPython %save* magic command (to get a listing of the full set of these commands execute in an *IPython* input cell the *%lsmagic* command. Consult the *IPython* reference given at the end of this chapter, on page 64, to understand more about these commands).

Let's start by noting that the lines of the just presented *for loop* code we are interested in were typed in the *IPython* input cells `In [20]:`--`In [25]:`. In your case the numbers of the cells will be different and (to complete this exercise) you need to take note of them. In case you have exited your *IPython* session, please open one and retype the code on page 38. Now type the following instruction on your available input cell of your *IPython* session (use the numbers from your actual *IPython* session at the end of this instruction):

Chapter 2, IPython session 17

```
In [26]: %save myfirstprog.py 24-25
The following commands were written to file `myfirstprog.py`:
n = 64
powers = range(n)
lasuma = 0
for j in powers:
   lasuma = lasuma + 2**j

print(lasuma)
lasuma

In [27]:
```

The just create file named (without the quotes) "myfirstprog.py" resides in the current directory were you started your *IPython* session, which you can find out by executing (they are *Linux* commands, a few of which are listed in the Appendix A.1, on page 17):

Chapter 2, IPython session 18

```
In [27]: pwd
```

```
Out[27]: '/home/srojas/The_prealgebra_book/CH02'

In [28]:
```

That the file is there can be checked via the command (notice that the answer obtained from the instruction executed on the input cell `In [28]:` is not labeled with the usual output cell `Out[28]:`) :

Chapter 2, IPython session 19

```
In [28]: ls -l myfirstprog.py
-rw-rw-r-- 1 srojas srojas 122 Jan 8 14:14 myfirstprog.py

In [29]:
```

The content of the file can be printed to the computer screen by typing:

Chapter 2, IPython session 20

```
In [29]: more myfirstprog.py
# coding: utf-8
n = 64
powers = range(n)
lasuma = 0
for j in powers:
   lasuma = lasuma + 2**j

print(lasuma)
lasuma

In [30]:
```

You should congratulate yourself as the printed lines form your first formal *Python* code (or *Pythonscripts*). We will learn how to write them directly, without using the *IPython* console later in the book. Notice that the filename ends with the extension ".py" (that we wrote explicitly when creating the file). It is a common practice to end *Python* scripts with that extension (not supplying that extension when using the "%save" instruction, *IPython*, by default, would add it to the given base-name).

To execute this code and any *Python* script directly from the *IPython* console, use the *IPython* magic command (%run) by typing:

Chapter 2, IPython session 21

```
%run full_path_of_the_python_script
```

In our case, since the *Python* script we want to execute resides in the same directory were the *IPython* session was started we only need to supply the name of the script to the (%run) *IPython* magic command, as follows:

Chapter 2, IPython session 22

```
In [30]: %run myfirstprog.py
18446744073709551615

In [31]:
```

Notice that only one output is printed to the screen. This why we used the *print Python* function in the code. Just typing a variable name in a *Python* script does not send the output to the screen.

In case you want to delete the file we just created, execute the following (but wait a bit before doing it if you really want to delete it):

Chapter 2, IPython session 23

```
In [31]: rm myfirstprog.py

In [32]: ls -l myfirstprog.py
ls: cannot access myfirstprog.py: No such file or directory

In [36]:
```

As a bonus, before deleting the file we just created, exit the *IPython* console by typing *exit* or *quit* on the available input cell. This will bring you back to the terminal or system shell where you started the *IPython* console. In there execute the following instruction (here $ is the prompt of our terminal or system shell, so you don't need to type it):

Chapter 2, System shell command 2

```
$ python myfirstprog.py
18446744073709551615
```

If things goes smooth and you get printed the answer to our wheat problem on your computer screen (as shown here), congratulate yourself again as you have executed your first *Python* program directly from the terminal or system shell, without need of the *IPython* console. This is the state of affairs were we are hitting. Now you can go back to the *IPython* console and execute the instruction to delete the *Python* script, in case you want to do so.

More advanced users of *Python* could tell you that via *Python list comprehension* programming you could get the answer to our exercise more succinctly, in the form:

Chapter 2, IPython session 24

```
In [39]: sum([ 2**j for j in range(64) ])
Out[39]: 18446744073709551615
```

With some extra practice you'll be able to get there, don't worry.

In doing this exercise we have written our first *Python* program. In general terms, we can say that to program is the craft of writing efficient instructions that, doing work for us, a computer can execute (in our exercise we found a way to avoid typing explicitly in the *IPython* console, as we did before, on page 32, the addition $2^0 + 2^1 + \cdots + 2^{63}$). Remember that meaningful typing mistakes (like typing $*$ instead of $**$) are hard to debug, and they can even pass undetected. By the way, by changing the value of n in the given code, you can check the rightness of the general result proved on the Appendix of this chapter, starting on page 59.

2.7.3 Relational operators in *Python*

Previous to introducing the *Python while loop*, we need to have a clear notion of *comparison* or *relational* operators in *Python*. These operators are listed in the Table 2.2.

Operator	Meaning
$<$	less than
$\leq$	less than or equal to
$>$	greater than
$\geq$	greater than or equal to
$==$	equal to
$!=$	not equal to

Table 2.2: Relational operators in *Python*

As you might have guessed these operators are used to compare objects. The result of such comparison if a (boolean) *True* or *False* value. Here are some examples:

Chapter 2, IPython session 25

```
In [7]: 5 <= 63
Out[7]: True

In [8]: 2 == 2
Out[8]: True

In [9]: a = 4

In [10]: 4 == a
Out[10]: True

In [11]: 0 == False
Out[11]: True

In [12]: 1 == False
Out[12]: False

In [13]: 1 == True
Out[13]: True

In [14]: 0 == True
Out[14]: False

In [15]: True == True
Out[15]: True

In [16]: False == True
```

```
Out[16]: False

In [17]: False == False
Out[17]: True

In [18]:
```

In this example we should note the use of the double equal sign (= =) to mean equality of both side terms in an expression. We should also notice that the integer zero (0) is considered equal to *False*, while one (1) is considered to be *True* (see input cells `In [11]:` and `In [13]:` with the corresponding output cells). It is left to you as an exercise to try some other comparison using these operators.

Expressions involving relational operators can (in an advanced usage of them) be combined with logical *and* (&) and *or* (|) operators to form new compound, complex expressions. We'll show its uses later in the book. What we have said about these operators is sufficient to continue with our discussion on the *Python while loop*.

2.7.4 The wheat problem via the *Python while loop*

Recall that to find the answer of the wheat problem we need to implement the addition $2^0 + 2^1 + 2^2 + \cdots 2^{63}$, the terms of which follows the pattern of two to the power of each one of the terms $0, 1, 2, 3, \cdots, 63$.

To make this addition, we can do it directly, by adding all the terms at once as we did on page 32. Another way is to do it step by step (like walking up an stair) by adding term by term. That is, start with 2^0 (let's call it the ground floor), then add to it 2^1 (your first step up in the stair) $2^0 + 2^1$, then add to this result 2^2 (your second step up in the stair), to obtain $2^0 + 2^1 + 2^2$, then add to this new result 2^3 (your third step up in the stair), to obtain $2^0 + 2^1 + 2^2 + 2^3$, and so forth, until reaching the step sixty-three.

You might have notice that to do the addition following the aforementioned procedure, one needs a mean to store or keep track of partial results. For instance, when we add the two terms $2^0 + 2^1$, it would be nice to hold the corresponding result of that addition so it can be added, in the next step, to 2^2 without we typing it again in the form $2^0 + 2^1 + 2^2$. Then, this new result needs to be saved and added, in the next step, to 2^3, and so on. This can be done with a *Python* variable. Let's call it *lasuma*, and it can be started holding the value of zero $lasuma = 0$ (in programming this is said to be initializing a variable to zero) because, as you have learned in the prealgebra course work, zero added to any number keep the number unchanged. Then, at any step j of our procedure this variable can be updated to hold the new value $lasuma = lasuma + 2^j$. Recalling that in *Python* programming the equal (=) sign means to execute the operations on the right of the sign and put (store) the result on the variable at the left of the sign, *Python* does this operation by computing first 2^j and then adding to it the value already contained in the variable *lasuma*, the result is then stored on the variable *lasuma*, erasing whatever was in there before.

The stated procedure could be witting in the following way (in programming this is called an *algorithm*):

1. Set (initialize) the variable: $lasuma = 0$
2. Set (initialize) the exponent $j = 0$
3. Update the variable $lasuma = lasuma + 2^j$
4. Update the exponent by one: $j = j + 1$
5. Check the exponent $j \leq 63$
6. If $j \leq 63$, Repeat steps 3--6, otherwise continue next step 7
7. Show the value on the variable *lasuma*

Accordingly, we need a way to automatically implement the stated procedure. For that we will use the *Python while loop* which has he following general structure:

```
while (condition):
  Body or Indented set of instructions to be executed
```

Not only comparisons between numbers can be used as conditions in while loops: any expression that has a boolean (True or False) value can be used. Such expressions are known as logical or boolean expressions. The keyword

In the first line of code, **`while`** is a *Python* reserved keyword, while *condition*, which is provided by the user (programmer), must be any *Python* expression that can be evaluated to either logical value *True* or *False*. The (indented) body of the loop (`Instructions to be Executed`) is executed while the *condition* takes the value *True*. The colon at the end of *condition* is a required sign, as is also required the indentation spaces for each instruction forming the body of the loop. The number of indentation spaces is at least one, and is the same for each line that belongs to the body of the loop (it is customary to use a minimum of four spaces).

Using the aforementioned elements, our devised procedure to perform the summation in our wheat exercise could be implemented in the following lines of code:

Chapter 2, IPython session 26

```
In [1]: n = 63

In [2]: lasuma = 0

In [3]: j = 0
In [4]: while (j <= n):
   ...:     lasuma = lasuma + 2**j
   ...:     j = j + 1
   ...:

In [5]: lasuma
Out[5]: 18446744073709551615

In [6]: 2**64 - 1 == lasuma
Out[6]: True

In [7]:
```

A description of these lines of code is as follows: on input cell `In [1]:` the variable n is assigned the value 63, which then is used build the boolean (*True* or *False*) expression ($j <= n$) forming the *condition* of the *while loop*. Input cells `In [2]:` and `In [3]:` initialize to zero both variables *lasma* (used to hold partial sum of the terms in the addition operation, until reaching the final value of it after the *while loop* ends). and j (used to control the termination of the *while loop*).

Following the flow of our *Python* lines of code, when the input cell `In [4]:` is reached, the variable j holds the value of zero and the *condition* of the *while loop* ($j <= n$) takes the value of *True* (can you see why?), having as consequence that the lines of code forming the body of the *while loop* are executed, by first assigning to the variable $lasuma = 1$ (can you see why?) and then updating the variable $j = 1$ (can you see why?). The execution of the *while loop* continues by checking again its *condition* ($j <= n$), which once more evaluates to *True*, updating this time $lasuma = 3$ and $j = 2$, repeating again the checking of the *condition* ($j <= n$), etc. This process continues (see exercise 2.6, on page 63) until the variable $j = 64$, after which the *condition* of the *while loop* ($j <= n$) is no longer *True* (can you recall why?) and the loop is exited.

After exiting the *while loop*, the flow of the code continues executing line of code on input line `In [5]:`, which prints on screen (see output line `Out[5]:`) the expected answer for the wheat problem. On input line `In [6]:` we makes use of the *Python* equal operator to compare the obtained result kept on the variable *lasuma* with the known result ($2^{64} - 1$). As expected, the comparison is shown to be *True* on the corresponding output cell `Out[6]:`.

We let you as an exercise to write this program to a file following the discussion starting on page 39.

In summary, *while loop* is another *Python* alternative to implement in *Python* repetitive tasks. And you can congratulate yourself as reaching this far you have gone through your second *Python* program. Again, by now you have enought knowlege of *Python* programing to start doing your own coding to try (iteractively, in *IPython* console) many of your prealgebra routine and non-routine exercises involving number crunching.

Both the *for loop* and the *while loop* are powerful *Python* instructions. We will made use of them in more complex situations, including the reading and writing of files.

2.8 The *guess two digits* game explained using *SymPy*

A popular game goes a follow (we'll follow the steps in an *IPython* session, but you are encouraged to use pencil and paper):

1. You ask a friend to think and write down (hide from you) a whole number containing three digits:

   Chapter 2, IPython session 27

   ```
   In [1]: thenumber = 371

   In [2]: thenumber
   Out[2]: 371

   In [3]:
   ```

2. Now you ask your friend to write (hiding it from you, of course) the number in reverse order:

   Chapter 2, IPython session 28

   ```
   In [3]: reversed_number = 173

   In [4]: reversed_number
   Out[4]: 173

   In [5]:
   ```

3. In this step you ask your friend to (again, you are not allowed to see it) subtract the smaller number from the larger one:

Chapter 2, IPython session 29

```
In [5]: difference = thenumber - reversed_number

In [6]: difference
Out[6]: 198

In [7]:
```

4. At this point your friend should tell you the number in the ones place. From that you will be able to tell your friend the other two digits, in the respective order. In our example she must tell you the number 8 (which is the one taking the ones place). From that you will tell her (of course without looking her writing) that the other two digits are 19.

Could you explain how will you be able to guess the two hidden (from you) digits in this game?

This game allows us to anticipate a brief introduction to *SymPy*: the module to perform symbolic or algebraic computations in *Python*. It will help us to give you necessary elements to answer the posed question.

Certainly, you are encouraged to also follow the steps we are going to execute in the computer using pencil and paper. That will help you get a deeper view of the mathematical behavior of the numbers involved in the game. Here we will be happy with showing the steps, without given a comprehensive explanation of *SymPy* (for that you might one to read the *SymPy* manual [`http://sympy.org/en/index.html`]. Many other functionalities of *SymPy* will be described in the next and the following chapters, specially when covering the topic of solving equations algebraically). Accordingly, consider this example as a motivational argument to continue your study of *Python*.

Recall, from your prealgebra course work, that a number of three digits xyz can be written in the form $xyz = x{\times}100 + y{\times}10 + z$, where x, y, and z can be any of the digits (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9). In *Python*, the above recipe can be implemented in algebraic terms via *SymPy* as follows (remember that whatever is after the numeral or hastag (#) symbol is a comment in *Python* and you do not need to type them as you follow the example).

We start by making available some *SymPy* functionality into any *IPython* session. In this example we are going to do so as we need a particular *SymPy* function (in other setups it is common to call everything we are going to use all at once, at the beginning).

Our first *SymPy* function will be *symbols*. It is the function that allows us to define variables, so *Python* will consider them as symbols not having a numerical value assigned to them, as we have been doing previously. This is done in the following lines of code:

Chapter 2, IPython session 30

```
In [7]: from sympy import symbols # 'symbols' is a sympy function used to

In [8]: x, y, z = symbols('x, y, z') # defines Python variables as symbols
                                 # (in this case x, y, and z)
In [9]:
```

Take note of the syntax *from sympy import function*. This is one of the recommended ways to make available a particular *function* from any *Python* module to the current computational session. This is used if knowing (before hand) we are not replacing any other existing function with that name in our setup or (if that happen) it is irrelevant. We will encounter other recommended ways in the chapters to come.

The *SymPy* function *symbols* is made available to the current *IPython* session on input cell `In [7]:`, and the variables x, y, and z are defined as symbols on input cell `In [8]:`.

In case you want to get some help on what the *SymPy* function *symbols* does or to see some extra examples of using it, any time after executing the line of code on input cell `In [7]:` you could execute at any input cell the command *symbols?* or *symbols??* (this method can also be applied to get help for any other function, and only works in the *IPython* console). To move around the displayed help page you could use the **up** and **down** keys of your computer keyboard. Hitting the **space bar** will move the help page screen by screen. You could quit the help page by pressing (at any time) the keyboard **q** key.

Now, after having defined the variables x, y, and z we can use them as generic names for any digit that makes any three digit number. Consequently, the next line of codes execute the steps of the game (pay attention that *Python* does not output any error for trying to use a variable that has not been assigned a value):

Chapter 2, IPython session 31

```
In [9]: thenumber = x*100 + y*10 + z # STEP 1: a three-digit whole number

In [10]: thenumber             # x, y, and z are symbols not numbers
```

```
Out[10]: 100*x + 10*y + z

In [11]: reversed_number = z*100 + y*10 + x # STEP 2: reversing the number

In [12]: reversed_number
Out[12]: x + 10*y + 100*z

In [13]: difference = thenumber - reversed_number # STEP 3: takes the
    difference
                                            # This assumes 'thenumber'
                                            # is the largest number
In [14]: difference
Out[14]: 99*x - 99*z

In [15]:
```

The just shown lines of codes does not need much explanation. You will be able to follow them with some attention. If you have any trouble, go back and follow the steps of the numerical example starting, on page 47.

To make a bit clear the result obtained at the *IPython* output cell `Out[14]:`, we would re-arrange the terms in that result by factoring it. This is done via the *SymPy* function *factor*, which allow the symbolic factorization of mathematical expressions. This is done in the following lines of code:

Chapter 2, IPython session 32

```
In [15]: from sympy import factor # 'factor' is another sympy function
                              # used to perform symbolic factorization

In [16]: difference = factor(difference) # factor the variable 'difference'
                                   # and reassign the result
                                   # to it.

In [17]: difference # checking that the result has been factorize
Out[17]: 99*(x - z)

In [18]:
```

The result shown on the *IPython* output cell `Out[17]:` tells us to take the difference between the left most digit (the one in the hundreds place) and the right most digit (the one in the ones place) of the given number and multiply the result by ninety-nine. Isn't cool the simplicity of this result after so many arithmetic operations!!

Now, Recall that this result was obtained taken for granted that the given number was larger than the reverse number. But, what happen if that is not the case, that the reverse number is the largest one between the two. Well, let's find out that in the following line of codes:

Chapter 2, IPython session 33

```
In [18]: difference2 = reversed_number - thenumber # STEP 3 in case the
                                          # the largest number is
                                          # the 'reversed_number'
In [19]: difference2
Out[19]: -99*x + 99*z

In [20]:
```

The result (at the *IPython* output cell `Out[19]:`) does not look very different than the one we already know. To make it a bit clear, let's re-arrange the terms in that result by factoring it. This time it will be done via the *SymPy* function `collect_const`, which allow the symbolic factorization of numbers in mathematical expressions (we let you as an right away exercise to see what happen is using the *factor* function as we did above). The following lines of code shows the result:

Chapter 2, IPython session 34

```
In [20]: from sympy import collect_const # 'collect_const' is a sympy function
                                   # used to collect numbers
In [21]: collect_const(difference2)
Out[21]: 99*(-x + z)

In [22]: (-x + z) == (z - x)
Out[22]: True

In [23]:
```

This time the result is shown at the *IPython* output cell `Out[21]:`. It is different from what we obtained previously in what the difference is the other way around as obtained there. In this case we need to take the difference between the right most and the left most digits of the given digits and (as before) multiply the result by ninety-nine.

In any case, carrying out the steps of the game in symbolic form, we arrive at the result that the difference between the number and its reverse form has the form $99 * (x - z)$ (if digit x is larger than digit z) or $99 * (z - x)$ (if digit z is larger than digit x). At this point you need to convince yourself that in either case the involved difference can take only any of the values (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) which needs to be multiplied by the number 99. This is done in the following *Python* lines:

Chapter 2, IPython session 35

```
In [26]: possible_values = list(range(10))

In [27]: possible_values
Out[27]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [28]: for i in possible_values:
    ...:  print('(x - z) or (z - x)={0} ==> 99x{0} = {1:03d}'.format(i,
       99*i))
    ...:

(x - z) or (z - x)=0 ==> 99x0 = 000
(x - z) or (z - x)=1 ==> 99x1 = 099
(x - z) or (z - x)=2 ==> 99x2 = 198
(x - z) or (z - x)=3 ==> 99x3 = 297
(x - z) or (z - x)=4 ==> 99x4 = 396
(x - z) or (z - x)=5 ==> 99x5 = 495
(x - z) or (z - x)=6 ==> 99x6 = 594
(x - z) or (z - x)=7 ==> 99x7 = 693
(x - z) or (z - x)=8 ==> 99x8 = 792
(x - z) or (z - x)=9 ==> 99x9 = 891

In [29]:
```

Considering that your friend should tell you the units place digit, can you see the pattern which allows you to guess correctly the other two digits? (hint: besides noticing the middle digit, take a look at the result of adding the hundreds place and the units place digits). In section 4.4, starting on page 156, this game is programmed to be played with the computer.

2.9 Some common errors due to unfollowing *Python* rules

Before digging deeper into *Python* programming, it is convenient to be aware of some problems that could arise when executing *Python* scripts either in the *IPython* console or directly from a terminal or system shell. This will help to keep away from you states of frustration and anxiety when doing *Python* programming.

The nice thing about these set of errors (a full list of them can be found at [`https://docs.python.org/3/library/exceptions.html`] is that the *Python* interpreter is able to capture them, raising (for the newcomer) cryptic messages, many of which are non intuitive ones (you might have experienced some of them already). Even though you need to pay attention to avoid this errors (you don't want that after several days your running program crash because of one of them) another set of (more dangerous) programming errors are logical ones (also called (*software bugs*) which are hard to find and the interpreter does not know anything about them. These kind of errors eventually could lead to *computational numerical disasters* (for some stories see On Software Bugs and Computational Numerical Disasters in the reference section of this chapter, on page 64).

Thus, what follows is a brief introduction of some common errors the interpreter could capture pointing out why they happen. The idea is that you can start building a library based on your experience to deal with them whenever they occur to you:

- **IndentationError**: This might happen when an input instruction start with unnecessary indentation (extra white spaces beginning the code instruction.)

Chapter 2, IPython session 36

```
In [1]: a = 2 # rightly indented

In [2]:  b = 9 # wrongly indented. This IPython does no through an error

In [3]: for i in [1,2,3]:
   ...:     a = 2 # set the indentation level
   ...:       b = 4 # # wrongly indented. An error happen
  File "<ipython-input-3-1a6820f6308e>", line 3
    b = 4 # # wrongly indented. An error happen
    ^
IndentationError: unexpected indent

In [4]: for i in [1,2,3]:
   ...:     a = 2 # set the indentation level
   ...: b = 4 # # wrongly indented. An error happen
  File "<tokenize>", line 3
    b = 4 # # wrongly indented. An error happen
    ^
IndentationError: unindent does not match any outer indentation
    level

In [5]: for i in [1,2,3]:
   ...:     a   =    2  # fix indentation level. internal spaces are ignored
   ...:     b =  4 # right indentation
   ...:     c = 5   # right indentation
   ...:

In [6]:
```

As shown on input line `In [2]:`, recent versions of *IPython* ignores the extra indentation starting a line of code. Nevertheless, the write way of writing starting line code instructions is with the right indentation level as shown on the input lines `In [1]:` and `In [5]:`. Notice the hat symbol $\wedge$ under the wrongly indented variable b. It is the way *Python* uses to tell us where we can look for what is causing the error.

- **NameError**: this error happen when *Python* encounters a symbol or variable name that it doesn't recognize.

Chapter 2, IPython session 37

```
In [7]: z
---------------------------------------------------------------------
NameError                              Traceback (most recent call last)
<ipython-input-7-a8a78d0ff555> in <module>()
----> 1 z

NameError: name 'z' is not defined

In [8]:
```

- **SyntaxError**: this error happen when an instruction is wrongly written. Here are some examples:

Chapter 2, IPython session 38

```
In [15]: print a) # open left enclosing round parenthesis is missing
  File "<ipython-input-15-12497d19cc2b>", line 1
    print a)
          ^
SyntaxError: Missing parentheses in call to 'print'

In [16]: print(a) # error goes away using enclosing set of round parenthesis
2

In [17]: a b # something is missing between a and b
  File "<ipython-input-17-7557d2f3a6ad>", line 1
    a b
      ^
SyntaxError: invalid syntax

In [18]: a*b # No error if inserting an operator
Out[18]: 8

In [19]:
```

- **TypeError**: this error happen when combining *Python* objects of different nature in a manner not allowed by *Python*:

Chapter 2, IPython session 39

```
In [20]: [1,2] + [1,2,3] # the + sing concatenate lists
Out[20]: [1, 2, 1, 2, 3]

In [21]: [1,2] + [1,2,3]
Out[21]: [1, 2, 1, 2, 3]

In [22]: 2 + [1,2,3] # the + sign is not defined between this two objects
----------------------------------------------------------------
TypeError                              Traceback (most recent call last)
<ipython-input-22-6ca8d9b5b92b> in <module>()
----> 1 2 + [1,2,3]
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'list'

In [23]:
```

- **ZeroDivisionError**: this error happen when somewhere a division by zero is being executed:

Chapter 2, IPython session 40

```
In [23]: 4/0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-23-6de94738d89d> in <module>()
----> 1 4/0

ZeroDivisionError: division by zero

In [24]:
```

- **IndexError**: this error happen when trying to get elements of objects supporting index (like *list*) beyond its length:

Chapter 2, IPython session 41

```
In [24]: milista=[10, 2, 3] # a list having three elements

In [25]: milista[0] # first (leftmost) element in the list
Out[25]: 10

In [26]: milista[2] # last (rightmost) element in the list
Out[26]: 3

In [27]: milista[3] # wrong element in the list
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-27-ca13b927419f> in <module>()
----> 1 milista[3] # wrong element in the list

IndexError: list index out of range
```

```
In [28]:
```

- **ValueError**: this error happen when trying to get elements outside the allowed range

Chapter 2, IPython session 42

```
In [28]: milista.index(2) # get the third element of the list
Out[28]: 1

In [29]: milista.index(2) # get the index of the object 2 in the list
Out[29]: 1

In [30]: milista[milista.index(2)] # get the element of index 1 in the
    list
Out[30]: 2

In [31]: milista.index(200) # no element 200 is the list. An error happen
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-31-f11de82a96b7> in <module>()
----> 1 milista.index(200) # no element 200 is the list. An error happen

ValueError: 200 is not in list

In [32]:
```

Remember that any of these errors will make your program crash. That is an unwanted behavior if it happen after hours or days of executing the program. Even though *Python* will capture them, it if better for us to find them first, before executing the program.

2.10 Chapter Summary

In this chapter you have done a lot! By now you can continue writing and executing computations with whole numbers using *Python* via the *IPython* console using the rules of *Python* you learned in this chapter which included defining variables, creating *Python list* objects, applying *Python* relational operators, and performing repetitive computations via the *Python for* and *while* loops. You also were able to work out and understand the math behind the popular game o guessing two digits which you can use to impress any audience not reading this book. At the end of the chapter, you became acquainted with some common errors hat could happen when wrongly applying the *Python* rules.

In the next chapter we will continue with you working on some extra computation with whole numbers, including setting and solving equations algebraically via the *SymPy* module. We also we will studying more *Python* objects to make your programming stronger in the following chapters,

Appendix of Chapter 2

A.1 Proof that for all $n \geq 0$, $2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$

In this appendix we will write a proof that $2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$ for all $n \geq 0$. This proof uses *mathematical induction*

The proof start by first checking that the equality holds for $n = 0$:

$$\begin{aligned} 2^0 &= 2^1 - 1 \\ 1 &= 2 - 1 \\ 1 &= 1 \end{aligned}$$

Then, we check the equality hods for $n = 1$:

$$\begin{aligned} 2^0 + 2^1 &= 2^2 - 1 \\ 1 + 2 &= 4 - 1 \\ 3 &= 3 \end{aligned}$$

And again we check it holds for $n = 2$:

$$\begin{aligned} 2^0 + 2^1 + 2^2 &= 2^3 - 1 \\ 1 + 2 + 4 &= 8 - 1 \\ 7 &= 7 \end{aligned}$$

The fact that the equality holds for these few values, some how it guides our intuition to get ways to verify it for any value $n \geq 0$. If we find a value for which the equality does not hold, then we have proved it to be wrong via a *counterexample*.

Since that is not the case, we have not found any *counterexample* disproving the equality, let's proceed by assuming it might be true for a general whole number $n = k$. If that is true, then it should happen that:

$$2^0 + 2^1 + 2^2 + \cdots + 2^k = 2^{k+1} - 1 \tag{A.1}$$

Taking for granted that this last equation is true, then (since k could be any whole number) it should also be true that the equality holds for the next whole number after k. That is, the equality should also holds for $n = k + 1$, which means that:

$$\underbrace{2^0 + 2^1 + 2^2 + \cdots + 2^k}_{\text{Left hand side of equation (A.1)}} + 2^{k+1} = 2^{k+2} - 1 \tag{A.2}$$

We can see that the enclosed terms by the under-brace bracket on the equation (A.2) is equal to the right hand side of the equation (A.1). This means that we can replace the enclosed part by the under-brace bracket on the equation (A.2) by the the term on the right hand side of the equation (A.1). In doing this replacement we obtain:

$$2^{k+1} - 1 + 2^{k+1} = 2^{k+2} - 1 \tag{A.3}$$
$$2 \times 2^{k+1} - 1 = 2^{k+2} - 1 \tag{A.4}$$
$$2^{k+2} - 1 = 2^{k+2} - 1 \tag{A.5}$$

The result of this last equation (A.5) confirms what we wanted to prove. That is, we have proved that $2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$ for all whole number $n \geq 0$.

Exercises of Chapter 2

Exercise 2.1 *Using an* IPython *console execute the following* Python *instructions:*

```
v0 = 5
g = 10
t = 6
y = v0*t - (g*t**2)/2
print(y)
```

Compare the previous way of computing with the next one:

```
5*6 - (10*6**2)/2
```

Which way of computing do you like?

Exercise 2.2 *What do you think will be the output of the following computing instructions:*

```
a = 5; b = 5; c = 5
a/b + c + a*c
a/(b + c) + a*c
a/(b + c + a)*c
```

Confirm your insight executing in an IPython *console these instructions.*

Exercise 2.3 *In an* IPython *console explore how to plot a data set executing the following (perhaps now cryptic) lines of code:*

```
import numpy as np
import matplotlib.pyplot as plt

mu, sigma = 10, 5
x = mu + sigma * np.random.randn(10000)
```

```
plt.hist(x, 50, normed=1, facecolor='g')
plt.xlabel('X')
plt.ylabel('Y')
plt.title(r'$\mu=10,\ \sigma=5$')
plt.grid(True)
plt.show()
```

Exercise 2.4 *In an* IPython *console explore some capabilities of symbolic computation via* SymPy *executing the following (perhaps now cryptic) instructions:*

```
del x, y;
x = 1
x + x + 1
from sympy import Symbol
x = Symbol('x')
x + x + 1
x.name
type(x)
s = x + x + 1
s**2
(s + 2)*(s - 3)
from sympy import expand, factor
expand( (s + 2)*(s - 3) )
factor( 4*x**2 + 2*x - 6 )
factor( x**3 + 3*x**2 + 3*x + 1 )
from sympy import pprint
pprint(s)
pprint(factor( x**3 + 3*x**2 + 3*x + 1 ))
pprint( expand( (s + 2)*(s - 3) ) )
from sympy import solve
solve( (s + 2)*(s - 3) )
solve( 4*x**2 + 2*x - 6 )
solve( s )
```

Exercise 2.5 *In an* IPython *console execute the following lines of code (a modification of the code presented on page 38) illustrating the values taken by the* `j-iterator` *and the variable* `lasuma` *on each iteration of the* for loop *(you might want to change the value of* n *to a smaller value, so the printed outcome can fit on screen):*

```
n = 64
powers = range(n)
```

```
lasuma = 0

for j in powers:
    lasuma = lasuma + 2**j
    print('At j = {0}, lasuma = {1}'.format(j, lasuma))

print(lasuma)
```

In this code we used the Python `print()` *function, which is used to print output to the screen of the computer. We will have a further discussion of this function later in the book (see section 4.2, on page 146).*

Exercise 2.6 *On an* IPython *console execute the following lines of code (a modification of the code presented on page 45) illustrating the values taken on each iteration by the control variable of the* while loop `j` *and the variable* `lasuma`*. (you might want to change the value of n to to a smaller value, so the printed outcome can fix on screen):*

```
n = 63

lasuma = 0

j = 0

while (j <= n):
    lasuma = lasuma + 2**j
    print('At j = {0}, lasuma = {1}'.format(j, lasuma))
    j = j + 1

print(lasuma)
```

In this code we used the Python `print()` *function, which is used to print output to the screen of the computer. We will have a further discussion of this function later in the book (see section 4.2, on page 146).*

References of Chapter 2

Books and/or Articles

- **Marecek, L. and Smith, M. A.** (2017). Prealgebra, Rice University, OpenStax `https://openstax.org`.
  Book available for free at: `http://cnx.org/content/col11756/1.9`
- **Rossant, C.** (2013). Learning IPython for Interactive Computing and Data Visualization, Packt Publishing.
- **Tahan, M.** (1993) The man who counted: a collection of mathematical adventures, W. W. Norton & Company.

References on the WEB

- **The man who counted** (english translation):
  `https://archive.org/details/TheManWhoCounted-English-MalbaTahan`
  The man who counted (spanish translation):
  `http://www.librosmaravillosos.com/hombrecalculaba/index.html`
- **A wikipedia entry for the wheat problem presented on page 32** is at:
  `https://en.wikipedia.org/wiki/Wheat_and_chessboard_problem`
- **A wikipedia entry for heart bit rates mentioned on page 33** is at:
  `https://en.wikipedia.org/wiki/Heart_rate`
- **On Software Bugs and Computational Numerical Disasters**:
  `https://en.wikipedia.org/wiki/Software_bug`
  `https://en.wikipedia.org/wiki/List_of_software_bugs`
  `www5.in.tum.de/~huckle/bugse.html`
  `http://www.parseerror.com/bugs/`

3

Applications involving Whole Numbers via *Python*

"I didn't fail once, I invented the lightbulb. It was just a 2000-step process."
Thomas Edison

3.1 Introductory remarks

In the previous chapter we started to write small *Python* programs taking advantage of the study of the basic operations with whole numbers covered in your Prealgebra course work.

In this chapter we will continue introducing more *Python* key concepts that will allow you to write even more complex programs. We will introduce the *Python if statement* as well as the *and* (&) and *or* (|) which are helpful to combine relational operators to generate boolean states of *True* or *False* values that can be applied to fork or bifurcate the execution of a *Python* program. Since we will make emphasis on writing programs, we will start the chapter with a brief mention of the text editors suitable for it.

In this regards, this chapter is particularly important because you will go more in deep into the **mechanisms of programming** as you study with a careful read the way we structure the programs (efficient or not) presented in this chapter.

As we mentioned earlier in the text, the task of programing requires high order complex thinking skills which during the development of the steps to write a program activates thought processes proper of an active learner acquiring independent computational thinking skills and performance. In fact, to write a program successfully demands the execution of three major steps: *designing* (the algorithm)--*implementing* (in a programing language the algorithm)--*assessing* (the correctness of the program), which are basic cognitive functions recognized as universally necessary to enable good performance in problem solving.

The aforementioned prescribed steps might be familiar to you if you recall the general steps of the *scientific method* which you have practiced in a laboratory session elsewhere (i.e. in a Physics or Chemistry class). Additional, useful discussion on applying general problem-solving strategies are found in the reference cited on page 144. The analogy with laboratory work could be of interest because as you get use to *design*, *implement*, and *test* algorithms that finds solutions to problems you will actually executing a simulated experiment, hence the buzzwords *computer simulations* will be part of your vocabulary.

Accordingly, after finishing this chapter, you'll be equipped with some extra basic *Python* tools that will help you fully explore numerically practically all of the topics, including the solving equations (not only the ones involving whole numbers), that you will be studying in your Prealgebra official course work, A practical non-routine example (the sailors, the coconuts, and the monkeys problem(will be discussed in detail (including its analytical solution).

3.2 Writing program using an appropriated text editor

As shown by the examples of the small *Python* codes you wrote and executed in the previous chapter, we are quickly reaching the point on which typing lines of code in the *IPython* console will be pretty clumsy. A better option would be to type the lines the code directly in a file and execute the file either from the system shell or terminal (using the *python* command) or from the *IPython* console via the *IPython* magic *%run* command, as we did in the previous chapter. For that we need to use an appropriated text editor such as the Windows *notepad* editor or the *Linux gedit* editor. There is a jungle of programming text editors available out there. As you get acquainted with some of them you'll be making your choice. The ones mentioned above are enough for this and more advanced programming course. Let's mention that the *Jupyter Notebook* is another option to write and execute *Python* scripts, but we are not covering it in this book.

Keep in mind that when writing *Python* scripts, you need to follow strictly the *Python* rules as we have been doing in the *IPython* console.

3.2.1 A brief introduction to the *gedit* editorindexText editor Gedit

In case you are reading this book accompanied with *Linux*, the **gedit** text editor is a good choice to start writing your programs. In the *Ubuntu Linux* distribution it can be installed via the command (executed in the system shell or terminal):

Chapter 3, System shell command 1

```
$ sudo apt-get install gedit
```

Before continuing, let's us type and execute in an *IPython* console the following lines of code (we will explain them in a while, so don't worry if you don't understand what this code does) :

Chapter 3, IPython session 1

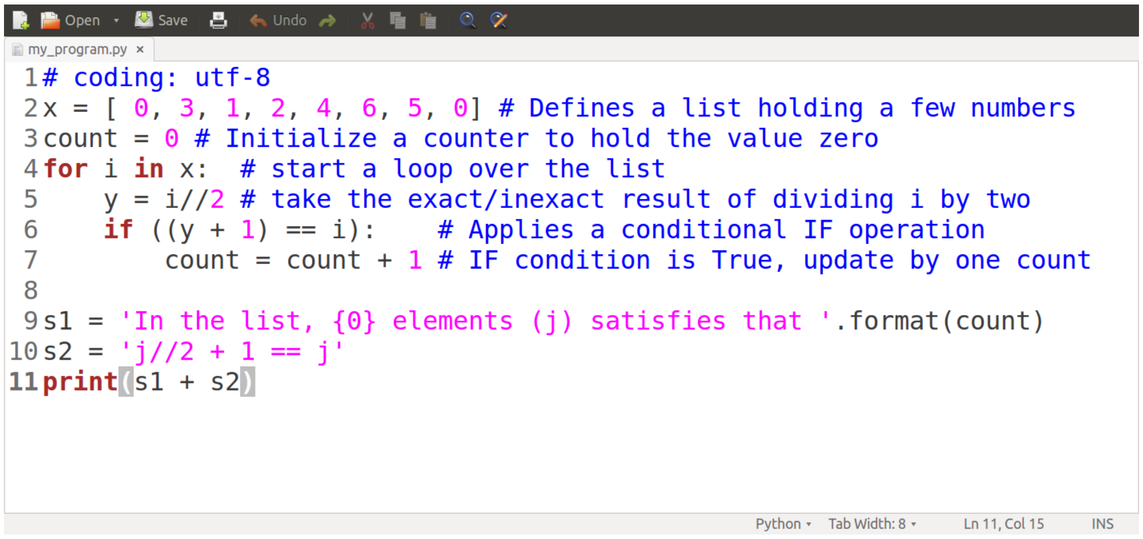
```
In [1]: x = [ 0, 3, 1, 2, 4, 6, 5, 0] # Defines a list holding a few numbers
In [2]: count = 0 # Initialize a counter to hold the value zero

In [3]: for i in x: # start a loop over the list
   ...:   y = i//2 # take the exact/inexact result of dividing i by two
   ...:   if ((y + 1) == i): # Applies a conditional IF operation
   ...:       count = count + 1 # IF condition is True, update by one count
   ...:

In [4]: s1 = 'In the list, {0} elements (j) satisfies that
    '.format(count)

In [5]: s2 = 'j//2 + 1 == j'

In [6]: print(s1 + s2)
In the list, 2 elements satisfies that element//2 + 1 == element

In [7]: %save my_program.py 1-6
```

After executing the last line of code (input cell `In [7]:`) you'll have created the file name `my_program.py` in your current working directory. You can list or executed this *Python* program from the *IPython* console by executing the commands:

Chapter 3, IPython session 2

```
In [9]: ls -l my_program.py # list the program
-rw-rw-r-- 1 srojas srojas 494 Jan 21 15:45 my_program.py

In [10]: %run my_program.py # run or executing the program
In the list, 2 elements (j) satisfies that j//2 + 1 == j

In [11]: pwd # list the current working directory
Out[11]: '/home/srojas/The_prealgebra_book/CH03'

In [12]: cp my_program.py my_program_backup.py # make a copy of your program

In [13]: ls -l my_program_backup.py # make sure the copy was meke correctly
-rw-rw-r-- 1 srojas srojas 494 Jan 21 16:01 my_program_backup.py

In [14]:
```

Before making changes to any working program it is a good idea to make a copy of it, as we



```
# coding: utf-8
x = [ 0, 3, 1, 2, 4, 6, 5, 0] # Defines a list holding a few numbers
count = 0 # Initialize a counter to hold the value zero
for i in x:  # start a loop over the list
    y = i//2 # take the exact/inexact result of dividing i by two
    if ((y + 1) == i):    # Applies a conditional IF operation
        count = count + 1 # IF condition is True, update by one count

s1 = 'In the list, {0} elements (j) satisfies that '.format(count)
s2 = 'j//2 + 1 == j'
print(s1 + s2)
```

Figure 3.1: Opening of the file `my_program.py` in **`gedit`**

did here on input cell `In [12]:`. This copy can be made directly from the *Linux* system shell or terminal the same *cp* command as used in the *IPython* console.

Now, assuming the **`gedit`** text editor is already installed in your system, you can start it by execiting in a system shell or terminal the command:

Chapter 3, System shell command 2

```
$ gedit
```

In case a program is already stored in a file (like `my_program.py`), you can open it for editing via executing (where the program resides) in the system shell or terminal the command (note that this way of starting **`gedit`** also works if the file name does not exist. It will be taken as the default filename to save anything you write via **`gedit`**) :

Chapter 3, System shell command 3

```
$ gedit my_program.py
```

After executng that command, you'll be presented with a **`gedit`** window as shown in Figure 3.1.

The first thing to notice about the **gedit** text editor is that you can move around the text using the mouse as well as the keyboard arrow keys for such task. The second thing is that by default it labels the lines written on it on the left of the windows (in our case, he file we are looking at contains 11 written lines). To activate or deactivate that feature, you can go to the `Edit` menu and choose the `Preferences` options. In the windows that opens check or uncheck the box to the left of `Display line numbers`. That `Preferences` windows also allows the changing of the gedit windows letters font size. For that, you hit

th the left mouse buttom the windows tab `Font & Colors`. In that section you should uncheck the box to the left of `Use the system fixed width font`, and choose the `Editor Font` tab to select the font size you want to use. Onece chosen, you hit the `select` tab to keep the changes or hit the `close` tab to leave things unchanged.

To save changes done to the program, you could hit the `seve` tab on top of the **gedit** window. You'll be prompted for a filename in case you are dealing with a new file. Alternatively, you could go to the `File` menu and then choose `Save` to keep things in the current name or `Save as` to store things in a new file. To exit the **gedit** text editor choose to `Quit` option from the `File` menu.

The few instructions of the **gedit** text editor we have worked with are enought to start writing and or modifying the programs we will be writing and executing in this and subsequent chapters of the book. Please read the **gedit** text editor manual to become proficient in its usage in case you decide to keep it as your working tool for writing *Python* scripts.

3.3 The *Python if statement*

An important idea of programming is the control of the execution flow of a program [`https://en.wikipedia.org/wiki/Control_flow`]. We have seen it before, when working with the *for* and *while* loops in the previous chapter. Going in and out of the loop is a fork in the execution of a program. *Python* also includes the *if statement* [`https://docs.python.org/3/reference/compound_stmts.html#if`] along with other options to interrupt the lineal execution of a program [`https://docs.python.org/3/tutorial/controlflow.html`].

3.3.1 The simple `if statement`

The *Python if statement* simple has the following construction:

```
if (condition):
    Body or set of indented instructions to be executed
```

This *if statement* has a structure similar to the *while* loop: the body of this *if statement* or set of indented *instructions to be executed* are executed whenever *condition* is *True*, otherwise they are ignored.

Here is our illustrative example. As you can recognize, it is the program you already wrote as instructed in page 66, also shown in Figure 3.1, on page 68: (after the discussion following these lines of code, for the subsequent sections to come we will assume that you have saved this code in the file `my_program.py`, as instructed on page 66):

Chapter 3, IPython session 3

```
In [5]: x = [ 0, 3, 1, 2, 4, 6, 5, 0] # Defines a list holding a few numbers

In [6]: count = 0 # Initialize a counter to hold the value zero

In [7]: for i in x: # start a loop over the list
   ...:   y = i//2 # take the exact/inexact result of dividing i by two
   ...:   if ((y + 1) == i): # Applies a conditional IF operation
   ...:       count = count + 1 # IF condition is True, update by one count
   ...:

In [8]: s1 = 'In the list, {0} elements (j) satisfies that
   '.format(count)

In [9]: s2 = 'j//2 + 1 == j'

In [10]: print(s1 + s2)
In the list, 2 elements satisfies that element//2 + 1 == element

In [11]:
```

Following the flow of these lines of *Python* code, you'll find that what it does is to take a set of numbers (stored in the *Python list x*, on input cell `In [5]:`) and counting (see the input cells `In [6]:` and the indented instruction under the simple *if statement*) the number of them satisfying the property that adding one to the (exact or inexact) result of dividing each number by two (see the first indented instruction under the *for loop*, on input cell `In [7]:`) the same number is obtained (see the *condition* of the simple *if statement*). Certainly you can follow these steps by hand computation, using pencil and paper (and we encourage you to do so) to get a better idea of how the program works. Think, now, on doing it by hand computation over millions of numbers.

We left you as an exercise to simplify a bit the number of lines in this code. Can you think on how the program can be modified if you want to print on screen the numbers (in the given *Python list* on input cell `In [5]:`) satisfying the stated property?

3.3.2 The `if--else statement`

The simple **`if statement`** just studied left us with a sense of incompleteness. In fact when doing a plan to go, let's say to the beach, we usually plan ahead to do something else if, for example, the weather does not help. And in that kind of situation is where the **`if--else statement`** comes to play. Its general structure (as you might have anticipated) is as follows:

```
if (condition):
    Instructions executed IF (condition) is True
else:
    Instructions executed IF (condition) is False
```

Let's add some extra (unnecessary) instructions to our previous program, on page 70. For that, you want to open in the **`gedit`** text editor the file `my_program.py`, as we did before from system shell o terminal by executing:

Chapter 3, System shell command 4

```
$ gedit my_program.py
```

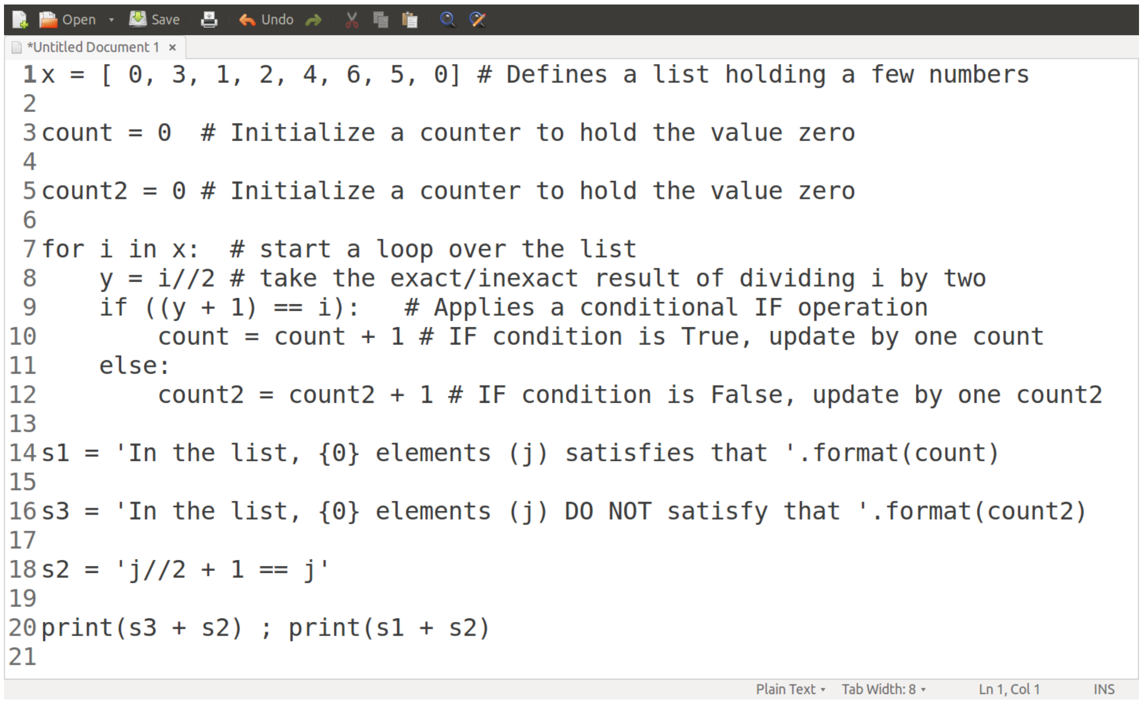
After that, modify the file to read exactly as shown in Figure 3.2 (on page 72) save it and quit from the **`gedit`** text editor (in case you have trouble editing the file, go to the directory named `chapter_03` of the programs that comes with this book that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap03_prog_02_ifelse.py` and make a copy of it via executing the command **`cp chap03_prog_02_ifelse.py my_program.py`** in the system shell or terminal. Alternatively, you can continue with what follows replacing `my_program.py` by `chap03_prog_02_ifelse.py`).

Once you are done with the changes in the file and saved it, let's see the content of the file in an *IPython* console by executing the line of code on input cell `In [3]:` below:

Chapter 3, IPython session 4

```
In [3]: more my_program.py
```

Executing that line of code will show you on the computer screen the content of the file `my_program.py`:



```
x = [ 0, 3, 1, 2, 4, 6, 5, 0] # Defines a list holding a few numbers

count = 0  # Initialize a counter to hold the value zero

count2 = 0 # Initialize a counter to hold the value zero

for i in x:  # start a loop over the list
    y = i//2 # take the exact/inexact result of dividing i by two
    if ((y + 1) == i):   # Applies a conditional IF operation
        count = count + 1 # IF condition is True, update by one count
    else:
        count2 = count2 + 1 # IF condition is False, update by one count2

s1 = 'In the list, {0} elements (j) satisfies that '.format(count)

s3 = 'In the list, {0} elements (j) DO NOT satisfy that '.format(count2)

s2 = 'j//2 + 1 == j'

print(s3 + s2) ; print(s1 + s2)

```

Figure 3.2: An example using the **if--else** construction

Chapter 3, IPython session 5

```
x = [ 0, 3, 1, 2, 4, 6, 5, 0] # Defines a list holding a few numbers

count = 0 # Initialize a counter to hold the value zero

count2 = 0 # Initialize a counter to hold the value zero

for i in x: # start a loop over the list
    y = i//2 # take the exact/inexact result of dividing i by two
    if ((y + 1) == i): # Applies a conditional IF operation
        count = count + 1 # IF condition is True, update by one count
    else:
        count2 = count2 + 1 # IF condition is False, update by one count2

s1 = 'In the list, {0} elements (j) satisfies that '.format(count)

s3 = 'In the list, {0} elements (j) DO NOT satisfy that
    '.format(count2)

s2 = 'j//2 + 1 == j'
```

```
print(s3 + s2) ; print(s1 + s2)

In [4]:
```

Take a moment to review this program, paying particular attention to the **if--else** construction inside the *for loop*. This construction can go anywhere in he program where we need it. Now run the program by executing the line of code on input cell `In [4]:` below:

Chapter 3, IPython session 6

```
In [4]: %run my_program.py
In the list, 6 elements (j) DO NOT satisfy that j//2 + 1 == j
In the list, 2 elements (j) satisfies that j//2 + 1 == j

In [5]:
```

Could you evaluate the validity of the printed output?. Could you see which added lines of codes are unnecessary and why? Taking out (deleting or commenting) unnecessary lines of codes, could you see which lines of codes should be modified and how to get the same printed output on the computer screen?

3.3.3 The `if--elif--else statement`

As the name of the *Python* **if--elif--else statement** might have suggested to you, it is a conditional selection of options based on several possible states (not just two as in the previous case). For instance, think about the three states of the traffic light when driving: if you find it in red, you do something; if you find it in yellow, you do something else; and else you continue. Thus, the **if--elif--else statement** helps to put together an action plan for several possible scenarios, checking them one by one and stopping the checking once a condition is fullfilled and the action plan for that scenario is executed. The **if--elif--else statement** has the following general construction:

```
if (condition 1):
    Instructions executed IF (condition 1) is True
elif (condition 2):
    Instructions executed IF (condition 1) is False
    and (condition 2) is True
elif (condition 3):
    Instructions executed IF (condition 2) is False
    and (condition 3) is True
  ⋮     ⋮     ⋮
elif (condition $n$):
    Instructions executed IF (condition $n-1$) is False
    and (condition $n$) is True
else:
    Instructions executed IF (conditions 1 to $n$) are False
```

Let's note that the **if--elif--else** construction is not equivalent to a set of simple **if statement** coming one after the other. In the former case, the checking for options to be executed stops once a condition happen to be *True*, no matter the state of the remaining options. In the later case, each condition is verified and anyone resulting to be *True* is executed.

To illustrate the use of the **if--elif--else statement**, let's modify our previous program, on page 71. For that, you want to open the file `my_program.py` using (in our case) the **gedit** text editor, as we did before from the system shell o terminal by executing:

Chapter 3, System shell command 5

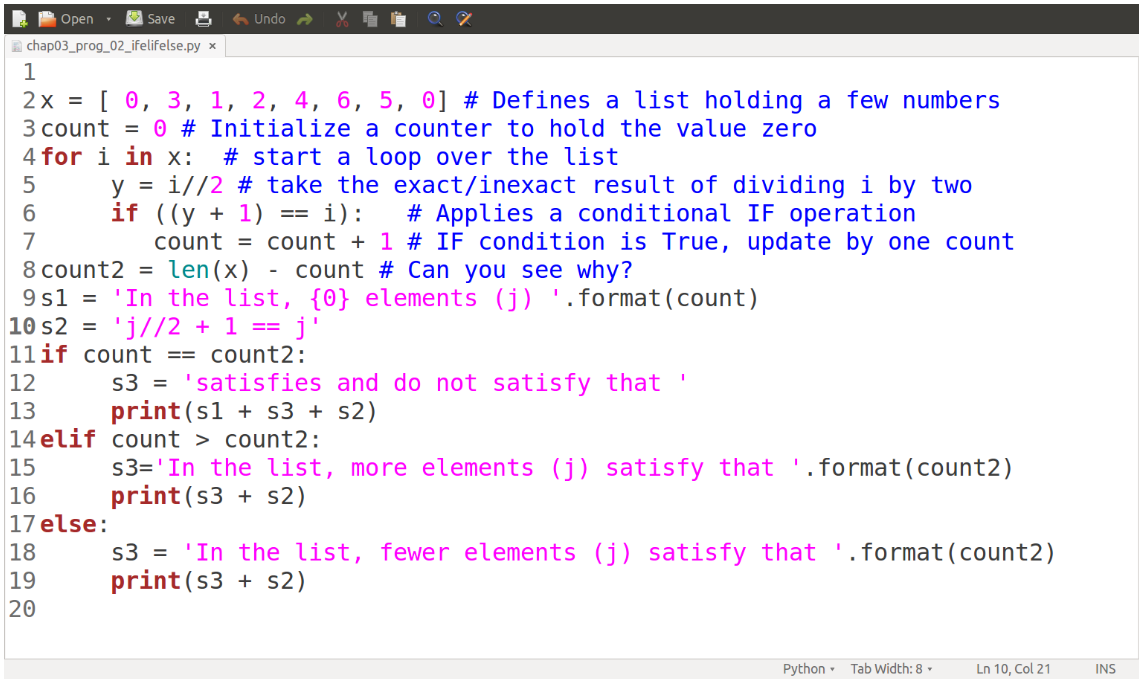
```
$ gedit my_program.py
```

Then modify the file to read exactly as shown in Figure 3.3 (on page 75) save it and quit from the **gedit** text editor (in case you have trouble editing the file, go to the directory named `chapter_03` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap03_prog_03_ifelifelse.py` and make a copy of it via executing the command **cp chap03_prog_03_ifelifelse.py my_program.py** in the system shell or terminal. Alternatively, you can continue with what follows replacing `my_program.py` by `chap03_prog_03_ifelifelse.py`).

Once you are done with the changes in the file and saved it, let's see the content of the file in an *IPython* console by executing the line of code on input cell `In [2]:` below:

Chapter 3, IPython session 7

```
In [2]: more my_program.py
```



```python

x = [ 0, 3, 1, 2, 4, 6, 5, 0] # Defines a list holding a few numbers
count = 0 # Initialize a counter to hold the value zero
for i in x:  # start a loop over the list
        y = i//2 # take the exact/inexact result of dividing i by two
        if ((y + 1) == i):   # Applies a conditional IF operation
           count = count + 1 # IF condition is True, update by one count
count2 = len(x) - count # Can you see why?
s1 = 'In the list, {0} elements (j) '.format(count)
s2 = 'j//2 + 1 == j'
if count == count2:
        s3 = 'satisfies and do not satisfy that '
        print(s1 + s3 + s2)
elif count > count2:
        s3='In the list, more elements (j) satisfy that '.format(count2)
        print(s3 + s2)
else:
        s3 = 'In the list, fewer elements (j) satisfy that '.format(count2)
        print(s3 + s2)

```

Figure 3.3: An example using the **`if--elif--else`** construction

Executing that line of code will show you on the computer screen the content of the file `my_program.py`:

Chapter 3, IPython session 8

```python
x = [ 0, 3, 1, 2, 4, 6, 5, 0] # Defines a list holding a few numbers
count = 0 # Initialize a counter to hold the value zero
for i in x: # start a loop over the list
    y = i//2 # take the exact/inexact result of dividing i by two
    if ((y + 1) == i): # Applies a conditional IF operation
       count = count + 1 # IF condition is True, update by one count
count2 = len(x) - count # Can you see why?
s1 = 'In the list, {0} elements (j) '.format(count)
s2 = 'j//2 + 1 == j'
if count == count2:
    s3 = 'satisfies and do not satisfy that '
    print(s1 + s3 + s2)
elif count > count2:
    s3='In the list, more elements (j) satisfy that '.format(count2)
    print(s3 + s2)
else:
    s3 = 'In the list, fewer elements (j) satisfy that '.format(count2)
    print(s3 + s2)

In [3]:
```

Compare this program with the previous one, on page 71. Could you answer the quetions posed at the end of the previous section, on page 73.? Pay particular attention to the **`if--elif--else`** construction. This construction can go anywhere in he program where we need it. Now run the program by executing the line of code on input cell `In [3]:` below:

Chapter 3, IPython session 9

```
In [3]: %run my_program.py
In the list, fewer elements (j) satisfy that j//2 + 1 == j

In [4]:
```

Take a moment to evaluate the validity of the printed output.

3.4 The *Python and* (&) and *or* (|) operators

As we have seen, to execute operations with the *while loop* and any of the *if statements* it is required to pass a boolean (*True* or *False*) condition to the *Python* interpreter. In our examples we have used simple (strightforward) conditions whose states of *True* or *False* can be easily verified. But from experience we know that *True* and *False* states could be made from complex combinations of the truthfulness condition of other (allow us to called) micro-states or sub-states.

Accordingly, to wider the possibilities to reproduce such conditions, *Python* includes (among others) the operators *and* (&) and *or* (|) that allows the combination of boolean expressions in order to make complex branching of the flow of a program via *True* or *False* boolean conditions (in case you are wondering about the bar symbol (|), it is the one that appears with the back slash (\) symbol in an standard English keyboard).

The general form to operate with these operators is like

((`lhs condition`) **`operator`** (`rhs condition`))

The parenthesis are not mandatory, but we recommend using them to make clear what is being composed. The external parenthesis will contain the result of combining the conditions contained in each internal parenthesis via the applied (&/|) operator.

A *true table* for these operators helps deciding how to build our branching conditions. In Table 3.1 it is shown the true table for the operator *and* (&).

and (&)	True	False
True	True	False
False	False	False

Table 3.1: The operator **and** (&) true table

One way to read the table is assigning any of the states (of *True* or *False*) shown in the left most column (under **and** (&)) to the `lhs condition`, and similarly assigning any of the states (of *True* or *False*) following horizontally the **and** (&) row to the `rhs condition`. The result of compounding them via the AND operator is the value of the intersecting cell. For instance having `lhs condition = True` and the `rhs condition = False`, the intercepting cell reads `False` (you read it: `True` AND `False` results in `False`). In the following *IPython* session which shows these operations using both representations of the operator:

Chapter 3, IPython session 10

```
In [5]: True and True
Out[5]: True

In [6]: True & True
Out[6]: True

In [7]: True and False
Out[7]: False

In [8]: True & False
Out[8]: False

In [9]: False and True
Out[9]: False

In [10]: False & True
Out[10]: False

In [11]: False and False
Out[11]: False
```

```
In [12]: False & False
Out[12]: False

In [13]:
```

The *or* (|) operator is shown in Table 3.2. We let you as an right away exercise to write its operations in an *IPython* session (recall that the bar symbol (|) appears with the back slash (\) symbol in an standard English keyboard).

or (|)	True	False
True	True	True
False	True	False

Table 3.2: The operator **or** (|) true table

In the remaining sections of this chapters and in the following one you'll find examples using these operators.

3.5 Statistical measures

Let's continue our fun experience (or workout exercises, if you like) in writing *Python* programs (scripts) by making the task of a few statistical measures automatic. Recall that whenever you see the word statistic, generally it has implicit the use of a pretty good amount of numbers (a lot of them) for the statistical properties that we are computing to make sense. Doing such computations by pencil and paper are a demanding (usually boring) task. Since computers likes to crunch numbers, not matter in which situation, many has written statistical software for computing statistical measures for us.

In this section we will have some fun writing our own pieces of *Python* code to do a few of such statistical computations. This way, while practicing the *Python* notions we have covered so far, we'll be getting used to necessary background to write any piece of statistical computation that one might need but is unavailable in our favorite statistical software. Additionally, we will introduce a few extra *Python* functionality like generating whole numbers randomly (actually will be generating *pseudo random* whole numbers, but eventually we'll use the phrase a *random* whole number, or its plural, for short).

Thinking carefully about the logic of the few programs (and about the problems that they were written for) covered in this book will lead you to improve your computational thinking skills, that for sure will help you to even discover alternatives ways to implement the codes shown here. More importantly, such carefully thinking will engage yourself on effective learning, which will lead to the enrichment of your independent thinking performance.

3.5.1 Computing *means* or *averages* using *Python*

The *mean* (also known as *arithmetic mean* or *average*) is a *statistical measure* of the central tendency of a set of values with no strange behavior. It is computed as the quotient of the result of adding the given set of values and the number of the given set of values. The set of values could represent the ages of the students taking this programming class (or, to make it a bit larger, the ages of everyone taking a Prealgebra class in the world).

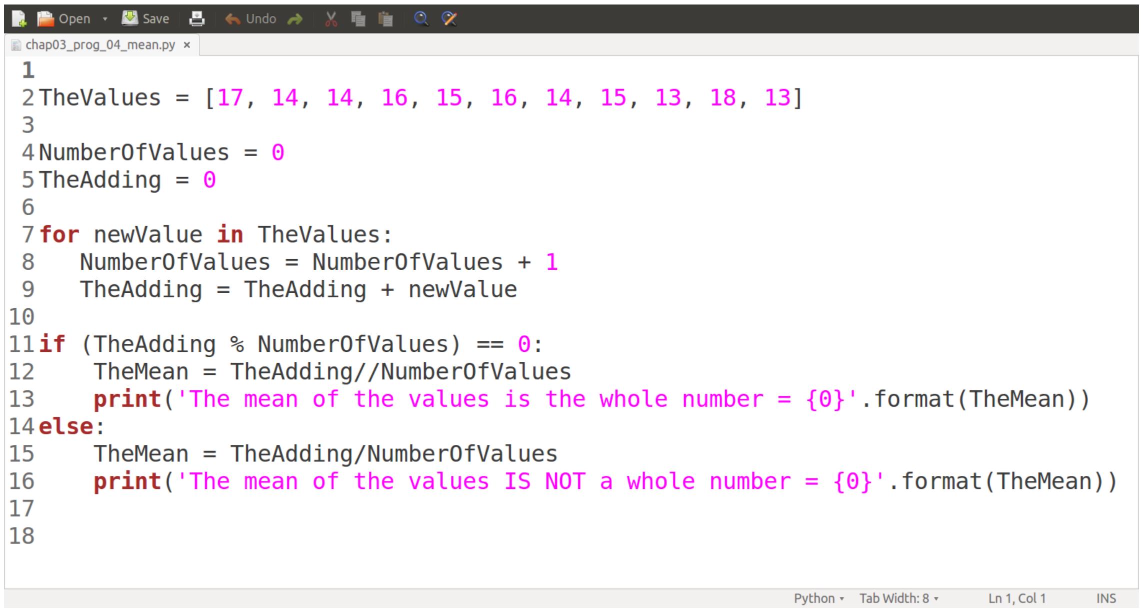
Accordingly, a detailed recipe (*algorithm*) to compute the *mean* of a set of values could involve the following steps:

1. Initialize a variable (i.e. *TheValues*) holding the set of values.
2. Initialize to zero a variable (i.e. *NumberOfValues*) that will be used to count how many elements are in the set of values.
3. Initialize to zero a variable (i.e. *TheAdding*) that will hold the adding of the set of values.
4. Read an element of the set of values.
5. Update the counter of values by one.
6. Add the read value to the variable *TheAdding* holding the addition of the values.
7. Continue by repeating steps 4--7, until reaching the end of the given set of values.
8. Compute the mean, assigning it to a variable (i.e. $TheMean = TheAdding/NumberOfValues$).
9. Print the result to the screen.

So, to write our *Python* program we need to collect the values and store them in a meaningful way in the computer memory, so we can have easy access to them. We also need to device a procedure to add them and count how many of them are. Let's point out that in this book we are going to stress the use of basic programming structures like *if statements* and *for* and *while* loops. The main reason for this is that they are common to any programming language and you need to know how to use them properly not to abuse instructions to break or exit them abruptly. As you gain experience with *Python* you will learn many alternatives to write programs in fewer lines of code that perhaps are more efficient. Since not every programming has available such alternatives you need to become acquainted on how the afore mentioned basic programming structures works.

Continuing writing our program, let's give a quick refreshment of the tools we have at hand to approach our task. We know that *Python* objects can be assigned to variables. This objects (that we know so far) could be numbers or a collection of them put together in a *list*. We also now that we can do recursive or repetitive computations via the *for loop* and the *while loop*. Additionally, we have learned in the previous sections of this chapter, that we could also make use of conditional execution of the flow of a program via a set of *if statements*, in case we need them. As anticipated by the presented algorithm, no *if statement* is required to make the computation of the average of a set of values.

Figure 3.4 (on page 80) shows a *Python* program that you can follow by hand (pencil and paper) computation to fully understand its flow. This program is available in the directory named `chapter_03` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap03_prog_04_mean.py` and make a copy of it via executing the command **`cp chap03_prog_04_mean.py my_program.py`** in the system shell or terminal. Alternatively, you can continue with what follows replacing `my_program.py` by `chap03_prog_04_mean.py`.



```
 1
 2 TheValues = [17, 14, 14, 16, 15, 16, 14, 15, 13, 18, 13]
 3
 4 NumberOfValues = 0
 5 TheAdding = 0
 6
 7 for newValue in TheValues:
 8     NumberOfValues = NumberOfValues + 1
 9     TheAdding = TheAdding + newValue
10
11 if (TheAdding % NumberOfValues) == 0:
12     TheMean = TheAdding//NumberOfValues
13     print('The mean of the values is the whole number = {0}'.format(TheMean))
14 else:
15     TheMean = TheAdding/NumberOfValues
16     print('The mean of the values IS NOT a whole number = {0}'.format(TheMean))
17
18
```

Figure 3.4: Program that computes the mean of a set of values

We will study this code with some level of details, so you can follow the flow of other codes in this chapter the same way we will be doing here.

Thus, following the flow of this code starts with the execution of line 2, initializing the variable *TheValues* to hold the set of values to which we want to compute its average (which are contained or grouped in a *Python list*). The flow of the code continues executing lines of code 2 and 3, initializing to zero two variables that we are going to use to hold partial results (after exiting the *for loop*, the variable *NumberOfValues* will hold the total number of values that are being averaged while the variable *TheAdding* will hold the sum of the set of values). Then, the flow of the codes enters the *for loop* (line of code 7), starting by assigning to the variable *newValue* the first element in the the list (that is, number 17). Entering the body of the loop, the variable *NumberOfValues* is increased by one (it will get the value $1 = 0 + 1$, while the variable *TheAdding* takes the value $17 = 0 + 17$). Then, the flow of the code goes back to the line of code 7, and the *for loop* checks if there are more values to process, finding that there are and assign the second value in the list (that is, number 14) to the variable *newValue*. Entering the body of the loop, the variable *NumberOfValues* is increased by one (it will get the value $2 = 1 + 1$, while the variable *TheAdding* takes the value $31 = 17 + 14$). Then, the flow of the code goes back to the line of code 7, and the *for loop* checks if there are more values to process finding that there are and assign the third value in the list (that is, number 14) to the variable *newValue*. And the process continues, until it reaches the last value in the list and the *for loop* exit (continuing the execution of the code by hand, you'll get the values that each one of the variables of the code should have after exiting the *for loop*. Could you think about how you'll check those values with the code?).

After exiting the *for loop*, the flow of the code encounters the *if--else statement* defined on lines of code 11--16 (is it necessary?). The *if* part of the statement verifies whether the division of the value hold by the variable *TheAdding* by the value hold by the variable *NumberOfValues* has remainder of zero (meaning that the result of the division is a whole number, the ones we have been studying so far). In case that happen, the mean is computed via the *Python* double slash division operator and printed to the screen, otherwise the *else* part of the statement is executed printing to the screen a (real) number assigned to the variable *TheMean*, which is correct but that we have not studied yet.

Now, let's execute this code on the computer, via the *IPython* console. Before, we can see the content of the file in an *IPython* console by executing the line of code on input cell `In [2]:` below:

Chapter 3, IPython session 11

```
In [2]: more my_program.py
```

Executing that line of code will show you on the computer screen the content of the file `my_program.py`:

Chapter 3, IPython session 12

```
TheValues = [17, 14, 14, 16, 15, 16, 14, 15, 13, 18, 13]

NumberOfValues = 0
TheAdding = 0

for newValue in TheValues:
   NumberOfValues = NumberOfValues + 1
   TheAdding = TheAdding + newValue

if (TheAdding % NumberOfValues) == 0:
    TheMean = TheAdding//NumberOfValues
    print('The mean of the values is the whole number =
        {0}'.format(TheMean))
else:
    TheMean = TheAdding/NumberOfValues
    print('The mean of the values IS NOT a whole number =
        {0}'.format(TheMean))
```

Once again, for the few values given in this exercises, you can follow these steps by hand computation, as we show above, using pencil and paper (and we encourage you to do so) to get a better idea of how the program works. Think, now, on doing it by hand computation over larger set of values (like computing the average age of all the students at your grade in your institution).

Notice the **`if--else`** construction we are using to display on the computer screen the obtained result. This program is valid for any set of numbers. Since we are studying whole numbers, we are assuming that the given set of values are all greater or equal than zero. Later in the book we will be looking at ways to verify that input objects to a program takes the right value.

Now run the program by executing the line of code on input cell `In [3]:` below:

Chapter 3, IPython session 13

```
In [3]: %run my_program.py
The mean of the values is the whole number = 15

In [4]:
```

We left you as an exercise to simplify a bit the number of lines in this code. How could you modify this program to print the mean of the given set of values regardless it is a whole number or not.?

Take a moment to evaluate the validity of the printed output. How do we know the program is given the right answer?

One way for doing so is to try several set of of input values for which we already know the answer that the program must reproduce. That give us a high level of confidence that the logic of the program is correct. Another issue to consider is the efficiency of the code, but that is beyond the scope of this book.

Once we have a working code, eventually we can think on making the code simpler. Sometimes, fewer lines of code helps to find possible bugs. It turns out that this code can be simplified using the *Python built in* function (or method) *sum* as follows:

Chapter 3, IPython session 14

```
In [4]: TheMean = sum(TheValues)/len(TheValues)

In [5]: print('The mean of the values = {0}'.format(TheMean))
The mean of the values = 15.0

In [6]:
```

As you could see, this simplification gives you a non whole number answer, but from the use of your calculator we know you are able to understand the meaning of the extra '.0'.

An alternative way of doing it is as follows:

Chapter 3, IPython session 15

```
In [6]: print('The mean of the values = {0}'.format(
   ...: sum(TheValues)/len(TheValues)))
The mean of the values = 15.0

In [7]:
```

Let's mention that the Anaconda *Python* distribution includes the *NumPy* and *SciPy* modules for performing this and many other statistical computations efficiently, for small and big data sets. We will mention a few extra words about them, later in the chapter, in the advance (optional) section, starting on page 96. It is advisable to keep in mind that the mean of any set of values can always be computed, but not always it will have a meaningful meaning.

3.5.2 Computing the *median* of a set of values using *Python*

The *median* is another (more robust) *statistical measure* of the central tendency of a set of values (which can even have strange behavior). It is computed by ordering (from least to greatest) the set of values and taking (if the set of values is odd) the middle number or the average of the two middle numbers (if the the set of values is even). You can see that the median separates the set of ordered values into two equal parts from both ends of the set.

As you can anticipate, this is a bit complicated than computing averages. A roughly recipe (*algorithm*) to compute the *median* of a set of values could involve the following steps:

1. Initialize a variable (i.e. *TheValues*) holding the set of values.

2. Sort the set of values kept in the variable (i.e. *TheValues*) and assign it to a variable (that could be *TheValues* to save computer memory).
3. Count how many values are in the list and assign it to a variable (i.e. *NumberOfValues*).
4. If the number of values is odd, take the middle element of the set of ordered values as the median of the set values. Else, if the number of values is even, take the average of the two middle elements in the set of ordered values as the median of the set values.
5. Print the result to the screen.

Comparing this recipe with the one (given of page 79) we implemented in the previous section to compute averages, the new (and hard) part is sorting efficiently the set of values (step 2). You'll find a huge amount of discussions in this subject out there, on the Internet. Accordingly, we are not going to our own sorting program in this book (but see Problem 3.2, on page 139). Instead, we will make use the fact that we need to overcome that requirement to introduce (or use) a function readily available to sort a set of values in a *Python list* object.

Let's start by typing in an *IPython* console the list of values of our previous program (you can take it directly from the file `chap03_prog_04_mean.py`, referred to in the previous section).

Chapter 3, IPython session 16

```
In [7]: TheValues = [17, 14, 14, 16, 15, 16, 14, 15, 13, 18, 13]

In [8]:
```

Make a copy of it via the built in *copy* method of a *Python list*, and ensure yourself it is really a copy:

Chapter 3, IPython session 17

```
In [9]: TheValuesBackUp
Out[9]: [17, 14, 14, 16, 15, 16, 14, 15, 13, 18, 13]

In [10]: TheValuesBackUp == TheValues
Out[10]: True

In [11]:
```

Now let's sort the values using the *sort* built in method of a *Python list*:

Chapter 3, IPython session 18

```
In [11]: TheValues.sort()

In [12]: TheValues
Out[12]: [13, 13, 14, 14, 14, 15, 15, 16, 16, 17, 18]

In [13]: TheValuesBackUp
Out[13]: [17, 14, 14, 16, 15, 16, 14, 15, 13, 18, 13]

In [14]: TheValuesBackUp == TheValues
Out[14]: False

In [15]:
```

As you can see, the built in *sort* method, order the values in the *list* from smaller to larger values, and assign or place the result in the same variable (this is way we made a copy of the initial set of values, in case they are required for extra computations in that initial state).

Another aspect of our recipe to compute the median is to count (step 3) how many values are in the given set. From the previous section, you already know how to do that via a *for loop* (can you?). But (from that section) we also know a better way which involves the using of the built in *Python len* function:

Chapter 3, IPython session 19

```
In [15]: NumberOfValues = len(TheValues)

In [16]: NumberOfValues
Out[16]: 11

In [17]:
```

Now comes the part (step 4) of deciding how to compute the median regarding the number of values is odd or even. For our exercise, we need to use the recipe for an odd total number of values. That is, we need to take the middle element in the sorted list of values. Thinking a bit about it, we can locate the middle element in any set of values (not only sorted ones) by taking the result of dividing the total number of elements in the list by two and adding to it the remainder (that will be one). In our example, the total number of elements is 11. Dividing it by two gives 5, with a remainder of one, locating then the middle value at the position $6 = 5+1$

(in our sorted list of values it is one of the numbers 15 contained in the list, specifically the one occupying the sixth place counted from either end of the list). Recalling that in a *Python list* the elements are counted from left to right starting from zero, this means that the position sixth in the list corresponds to the fifth element (counting from left to right we have the elements 0, 1, 2, 3, 4, 5, and the others). Thus we see that it corresponds to the result of our division without adding the remainder. Let's see this in the *IPython* session (remember that to take the element n in a list we use the construction $nameOflist[n]$):

Chapter 3, IPython session 20

```
In [17]: temp = NumberOfValues//2

In [18]: temp
Out[18]: 5

In [19]: TheMedian = TheValues[ temp ]

In [20]: TheMedian
Out[20]: 15

In [21]:
```

Let's see now the situation on which the number of values is even. For that we will delete one element from our toy example list. For that we will use the *pop* built in method for any *Python list* object that takes as argument the position of the element we want to extract. In the example we'll delete the element at position 9th):

Chapter 3, IPython session 21

```
In [21]: TheValues
Out[21]: [13, 13, 14, 14, 14, 15, 15, 16, 16, 17, 18]

In [22]: TheValues.pop(9) # delete he 9th element in the list
Out[22]: 17

In [23]: TheValues
Out[23]: [13, 13, 14, 14, 14, 15, 15, 16, 16, 18]

In [24]: NumberOfValues = len(TheValues)

In [25]: NumberOfValues
```

```
Out[25]: 10

In [26]:
```

Notice that the *pop* built in method returns the deleted value (see output cell `Out[22]:`) which can be assigned to a variable if necessary for additional computational tasks. Output cell `Out[23]:` shows that the value 17 is no longer in the *TheValues* list, and it has now a *NumberOfValues* of 10 (which, as you know, is an even number). In this case we need to take the average of the two middle numbers in the list. Output `Out[23]:` shows that they are the last 14 value and the first 15 value (counted from left to right), which corresponds to the $4th$ and $5th$ positions in the list. Thinking a bit how to get them, we see that as we did in the previous computation of the mean, the $5th$ position is obtained by taking the result of dividing the total number in the list by two, and the other middle value is obtained by moving backward one place from this position. That is, given that we are assigning the furthest position of the middle element in the variable *temp*, the position of the first middle element is obtained by subtracting one from the furthest position (i.e. $temp - 1$). This is implemented in the following *IPython* cells:

Chapter 3, IPython session 22

```
In [26]: TheValues[ temp ]
Out[26]: 15

In [27]: TheValues[ temp - 1 ]
Out[27]: 14

In [28]: TheMedian = (TheValues[ temp - 1 ] + TheValues[ temp ])/2

In [29]: TheMedian
Out[29]: 14.5

In [30]:
```

Well, the result for the median (on output cell `Out[29]:`) is not a whole number (generally the median, as well as the mean, will not be a whole number), but we are sure you know how to handle it.

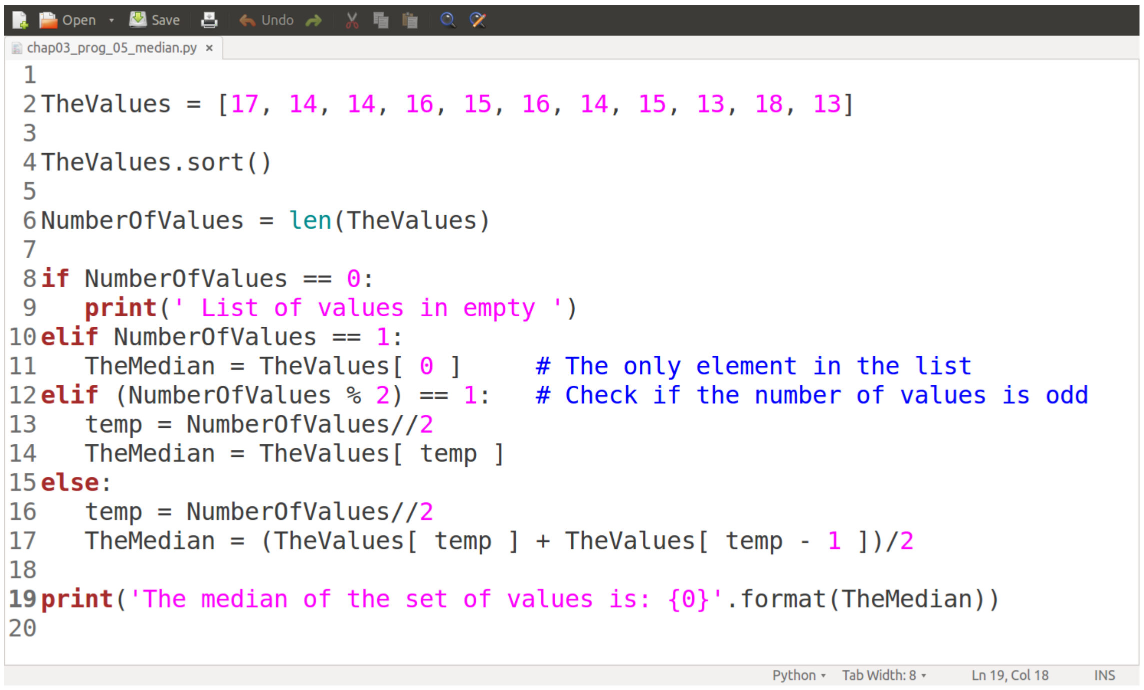
Hokey dokey!, you did a great job getting here. The next step is to make automatic the shown steps, so we do not depend on visual inspection to decide how to proceed as we did here.

Perhaps you have already anticipated that to implement these operations an *if--else* statement will do, using as the conditional control variable the *Python mod* or remainder (%) operator. In fact, this is the case, but instead we have implemented an *if--elif--else* statement to handle a couple of particular situations we have not discussed here (could you explain them?).

The full listing of the program is shown in Figure 3.5 (on page 88). Additionally, this program is available in the directory named `chapter_03` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap03_prog_05_median.py` which you can execute in the *IPython* console using the magic %run command as we have done before:

Chapter 3, IPython session 23

```
In [30]: %run chap03_prog_05_median.py
The median of the set of values is: 15

In [31]:
```



```

TheValues = [17, 14, 14, 16, 15, 16, 14, 15, 13, 18, 13]

TheValues.sort()

NumberOfValues = len(TheValues)

if NumberOfValues == 0:
    print(' List of values in empty ')
elif NumberOfValues == 1:
    TheMedian = TheValues[ 0 ]      # The only element in the list
elif (NumberOfValues % 2) == 1:    # Check if the number of values is odd
    temp = NumberOfValues//2
    TheMedian = TheValues[ temp ]
else:
    temp = NumberOfValues//2
    TheMedian = (TheValues[ temp ] + TheValues[ temp - 1 ])/2

print('The median of the set of values is: {0}'.format(TheMedian))

```

Figure 3.5: Program that computes the median of a set of values

3.5.3 Computing the *mode* of a set of values using *Python*

The *mode* is a *statistical measure* that counts the value that occurs most often in a set of values. That is, to find the mode of a set of values, we need to count the number of repetitions of each value. The value with the most repetitions is the mode. In case every value appear only once (i. e. no value is repeated), we say that the set of values has no mode. If there are two or more values having the most repetitions, the set of values is said to be multi *mode* or that its *mode* is not unique.

Accordingly, a roughly recipe (*algorithm*) to compute the *mode* of a set of values could be stated in the following terms:

1. Initialize a variable (i.e. *TheValues*) holding the set of values.
2. Pass through each value and count how many times each value is repeated in the set of values.
3. Take the value (or the values) with the most repetitions in the set, and assign it to the variable holding the mode.
4. Print the result to the screen.

This recipe does not look too complicated to implement, but don't be fooled by its looking simplicity. Actually the new (and hard) part is finding an efficient way to go through the values and counts the repetition of each one (step 2). We will happy to write a program that computes rightly the mode of a set of values. It will be a homework for you to check its efficiency, later on, when you get more experience with *Python*. We will make a detailed discussion of our approach as it gives a lot of practice applying what we have learned so far, in particular *looping*, which is at the heart of repetitive computing and not always you can avoid them.

Let's start by typing in an *IPython* console the list of values of our previous program (you can take it directly from the file `chap03_prog_05_median.py`, referred to in the previous section).

Chapter 3, IPython session 24

```
In [1]: TheValues = [17, 14, 14, 16, 15, 16, 14, 15, 13, 18, 13]

In [2]:
```

A quick inspection of the set of values shows that the set has the value 14 as its mode (because it is the value repeated the most in the set).

We now need to go through the elements in the list and count how many times each one is repeated.

Our intuitive approach to do this (at a first thought) is to go through each value in the list an write in a different *list* (i.e. *holdvals*) how many times that value appear in the set of values. Two things to notice in this approach: first, that the counting will be written in the new *list* having the same position that each one has in the *list* holding the set of values. Second, the procedure is repeated with each element in the *list*, regardless if it has been already counted.

Chapter 3, IPython session 25

```
In [2]: holdvals = []

In [3]: for j in TheValues:
   ...:    temp = 0
   ...:    for k in TheValues: # count how many times value in j is repeated
   ...:       if j == k:
   ...:          temp = temp + 1
   ...:    holdvals.append(temp)
   ...:

In [4]: print(holdvals)
[1, 3, 3, 2, 2, 2, 3, 2, 2, 1, 2]

In [5]:
```

As you must convince yourself, the first entry (1) in the *list holdvals* corresponds to the count of the first element (number 17) in the *list TheValues*. The second entry (3) in the *list holdvals* corresponds to the count of the second element (number 14) in the *list TheValues*, and so forth.

Let's explain a bit more this piece of code by following its flow. On input cell `In [2]:` an empty *list* is created. We will use it to keep the repetition of each value. Then, on input cell `In [3]:` a nested *for loop* (two of them) is used to go through each value in the list (the external loop) and compare it with each one (including itself) of the values in the list (this is done via the internal *for loop* in combination with the *if statement*. We keep track of how many times each value is repeated with the *temp* variable, which is first initialized to zero after one value is read and assigned to the variable *j* in the external *for loop*, and then it is increased by 1 in the body of the *if statement*. After exiting the internal *for loop*, the assigned value of this variable *temp* is added to the *list* defined on input cell `In [2]:`. You can see that the added score will be at the position of the current value in the variable *i*.

Our next step is to find the most repeated value in the set. For that we need to find the largest (maximum) score keep in the *list* (i. e. *holdvals*) we just filled out holding how many times each value is repeated in the data set. We take advantage of this fact to introduce the built-in *Python max* function (but check Problem 3.1, on page 139). As usual, you can look at some documentation of this function (*max*) directly from an *IPython* cell by executing the command *max?* or *max??*. Accordingly, the next line of code in our program read as:

Chapter 3, IPython session 26

```
In [5]: themax = max( holdvals )

In [6]: themax
Out[6]: 3

In [7]:
```

Inspecting the values in the *list* obtained as output of the input cell `In [4]:`, you can convince yourself that the output cell `Out[6]:` is, indeed, the maximum value assigned to the variable *themax* via the *max* function as defined in the input cell `In [6]:`.

Now, continuing with our program, we need to find out whether our set of values has only one mode or it is multi-mode (the mode is not unique). One way to find out that is to go through the values kept in the *list holdvals* and check (using the the given set of values) if the maximum count correspond to one or more than one value in the set.

In order to do this, we first extract the repeated values corresponding to the maximum score (we do it regardless it is repeated or not) and set them in a new *list* (i. e. *repeated*). The following lines of code shows a way of doing this:

Chapter 3, IPython session 27

```
In [7]: MostRepeatedVals = []

In [8]: for k in range( len(holdvals) ):
   ...:   if holdvals[k] == themax:
   ...:      MostRepeatedVals.append( TheValues[k] )
   ...:

In [9]: print('The most repeated values are: ', MostRepeatedVals)
The most repeated values are: [14, 14, 14]

In [10]:
```

As you can see, in this toy exercise the variable *MostRepeatedVals* holds the value(s) (only one type in this example) most repeated in the data set. To consider the possibility that the mode is not unique, we need to verify that the values in the list *MostRepeatedVals* are or not unique. This is done in the following lines of code (see Problem 3.3, on page 140):

Chapter 3, IPython session 28

```
In [10]: for k in MostRepeatedVals:
    ...:   MultiMode = 0
    ...:   for j in MostRepeatedVals:
    ...:      if j != k:
    ...:         MultiMode = MultiMode + 1
    ...:

In [11]:
```

As mentioned, these lines of code are written to check if our data set has or not a unique mode. In case the mode is not unique, the variable *MultiMode* will be assigned a value greater that one. A special case is when the data set does not have a mode (the variable *MultiMode* takes assigned the value of the length of the data set). This variable then help us to write the final part of our code that prints on the screen the mode of the data set according to the value taken by this variable *MultiMode*:

```
if MultiMode == 0:
   print('The mode of the data set is unique:')
   print('  Value {0} is repeated {1} times.'
                    .format(MostRepeatedVals[0], len(MostRepeatedVals) ))
elif MultiMode == (len(MostRepeatedVals)-1):
   print('The data set has no mode')
else:
    print('The mode of the data set is not unique:')
    different = MostRepeatedVals.copy()
    pairs = []
    while different != []:
         i=0
         l = different[0]
         temp = []
         for k in different:
            if k != l:
               temp = temp + [k]
            else:
               i = i + 1
         pairs = pairs + [[l, i]]
```

```
        different = temp
    for i in pairs:
        print(' Value {0} is repeated {1} time(s).'.format(i[0], i[1]))
```

The structure of this *if--elif--else statement* should be clear to you from what we have previously commented. The hard part to understand if the body of the *else* part. What this part of the code does is to take apart each mode of the data set and prints it to the screen. Notice the test condition for the *while loop*. This gives you an idea that such test condition could be very complex constructions having, of course, as output a *True*/*False* boolean condition. We encourage you to spend some time following the flow of this piece of code using pencil and paper to fully understand what it does and, perhaps, devise ways to improve it.

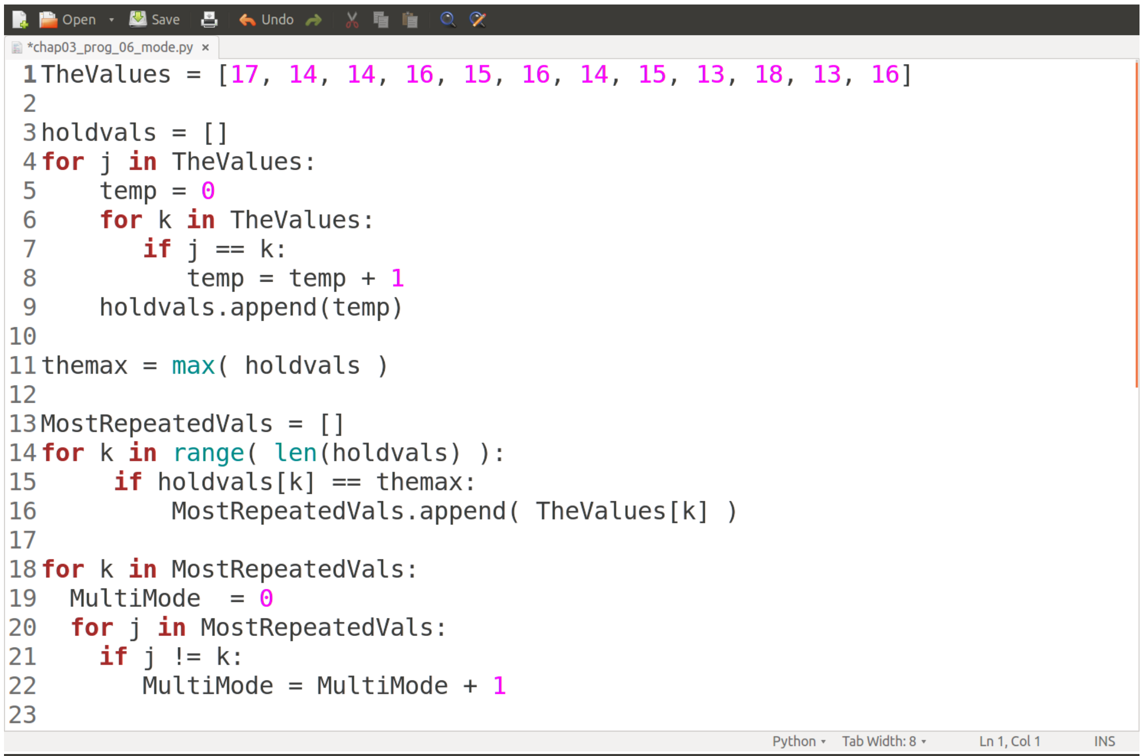
The full listing of the program is shown in Figure 3.6 (on page 95). Additionally, this program is available in the directory named `chapter_03` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap03_prog_06_mode.py` which you can execute in the *IPython* console using the magic %run command as we have done before:

Chapter 3, IPython session 29

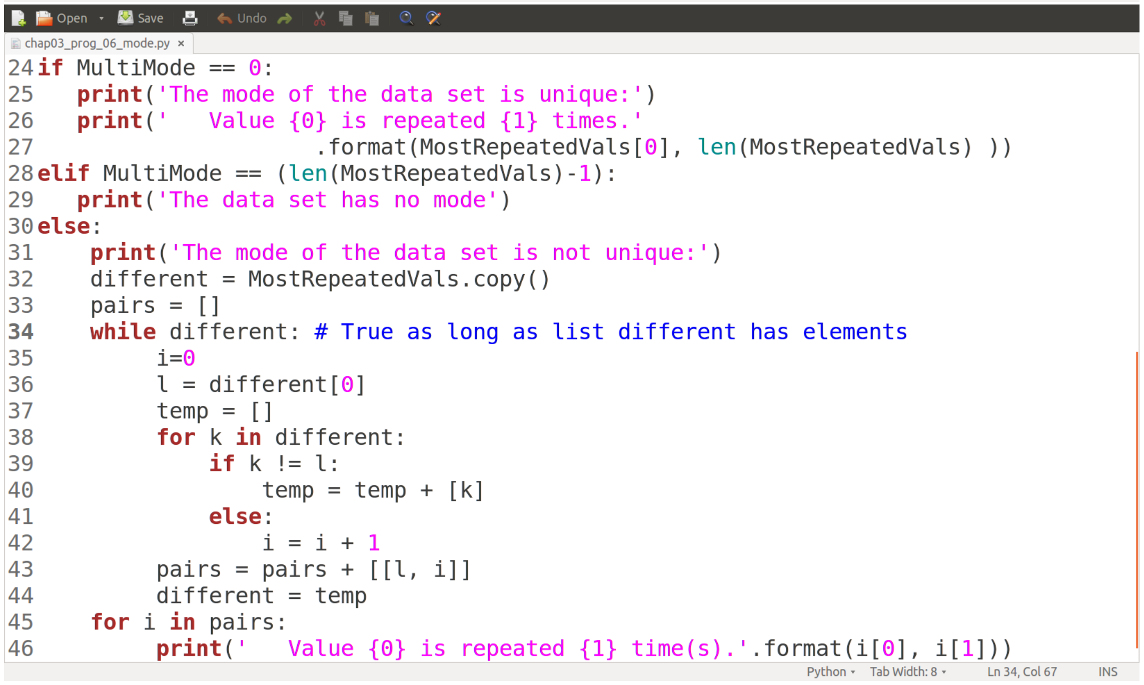
```
In [1]: %run chap03_prog_06_mode.py
The mode of the data set is not unique:
  Value 14 is repeated 3 time(s).
  Value 16 is repeated 3 time(s).

In [2]:
```

This program involves a great level of complexity in the logic and the flow of the code instructions to accomplish the task of finding the *mode* of a set of values. It also offer an excellent option to practice the testing of programs. For that you need to try many input data set for which you know the answer spamming several cases: the data set should include set of values having a mode, no mode, and non unique mode. A good source of testing data could be the workout exercises in your Prealgebra course work, including the ones in your textbook. The examples we have use employs input data consisting of whole numbers. To test the general validity of the program, you should consider running the program on different set of numbers, including negative integers and also you can try data set containing real numbers (if you are already familiar with them). Trying a mixture set of values will help to advice the potential user of the program about conditions on which it will fail and also it helps to see what improvement requires the program to be used in general.

To fully evaluate that your program is given the right answer, *Python* offer ways to perform the computation of the *mode* of a data set via a *mode* function that comes with its statistical package, as we will show in section 3.5.4, starting on page 96. In the next section we will cover how to generate some testing data pseudo-randomly (meaning no truly random).



```python
TheValues = [17, 14, 14, 16, 15, 16, 14, 15, 13, 18, 13, 16]

holdvals = []
for j in TheValues:
    temp = 0
    for k in TheValues:
       if j == k:
          temp = temp + 1
    holdvals.append(temp)

themax = max( holdvals )

MostRepeatedVals = []
for k in range( len(holdvals) ):
     if holdvals[k] == themax:
         MostRepeatedVals.append( TheValues[k] )

for k in MostRepeatedVals:
  MultiMode  = 0
  for j in MostRepeatedVals:
    if j != k:
       MultiMode = MultiMode + 1

```



```python
if MultiMode == 0:
   print('The mode of the data set is unique:')
   print('   Value {0} is repeated {1} times.'
                     .format(MostRepeatedVals[0], len(MostRepeatedVals) ))
elif MultiMode == (len(MostRepeatedVals)-1):
   print('The data set has no mode')
else:
    print('The mode of the data set is not unique:')
    different = MostRepeatedVals.copy()
    pairs = []
    while different: # True as long as list different has elements
         i=0
         l = different[0]
         temp = []
         for k in different:
             if k != l:
                 temp = temp + [k]
             else:
                 i = i + 1
         pairs = pairs + [[l, i]]
         different = temp
    for i in pairs:
         print('   Value {0} is repeated {1} time(s).'.format(i[0], i[1]))
```

Figure 3.6: Program that computes the mode of a set of values

3.5.4 Computing statistical measures via *Python* statistical modules

We have written programs to compute the *mean*, *median*, and *mode* of a set of values. Writing these functions was a good exercise to apply the basic *Python* programming instructions and to give it a first try to our computational thinking skills.

As you might have guessed already, *Python* comes with some modules that already contains functions or methods to compute the mentioned quantities. You can use these functions to check if the ones written by us is given correct answer.

Let's see how it works via the following *IPython* session:

Chapter 3, IPython session 30

```
In [1]: import statistics as stat

In [2]: TheValues = [17, 14, 14, 16, 15, 16, 14, 15, 13, 18, 13]

In [3]: stat.mean( TheValues )
Out[3]: 15

In [4]: stat.median( TheValues )
Out[4]: 15

In [5]: stat.mode( TheValues )
Out[5]: 14

In [6]:
```

On input cell `In [1]:` we make available to the current *IPython* session the *Python statistics* module with the pseudonym given by the name *stat* (you could use any other valid name if you wish to). If you want to see the available methods in the statistical module, in an *IPython* input cell type *stat.* and hit the computer **TAB**-key. A small window will appear on the *IPython* session which you can browse using the arrow keys on the keyboard. Alternatively, you could execute the command *dir(stat)*. The full set of methods will be displayed on the computer screen. In the shown list you will find *mean*, *median*, and *mode* methods available in the module. To read some help about using, for instance, the *mean* method, you could execute in an *IPython* input cell *stat.mean?* or *stat.mean??*. You can do similarly with any other method available in the module.

Thus, on input cells `In [3]:`--`In [6]:` it is done the computation of the the *mean*, *median*, and *mode* of the set of values given on input cell `In [2]:`. You could check that the answer on each output cell corresponds to the ones obtained by our programs.

Now, see what happen if you give to the *mode* method a set of values having a non-unique *mode*:

Chapter 3, IPython session 31

```
In [6]: TheValues = [17, 14, 14, 16, 15, 16, 14, 15, 13, 18, 13, 16]

In [7]: stat.mode( TheValues )
---------------------------------------------------------------------------
StatisticsError                           Traceback (most recent call last)
<ipython-input-7-3ba4cbb69130> in <module>()
----> 1 stat.mode( TheValues )

~/myProg/Anaconda35001/lib/python3.6/statistics.py in mode(data)
    505     elif table:
    506         raise StatisticsError(
--> 507             'no unique mode; found %d equally common values' %
    len(table)
    508             )
    509     else:

StatisticsError: no unique mode; found 2 equally common values

In [8]:
```

You only get an *StatisticsError* message indicating that the data set has *no unique mode* and that *2 equally common values* were found. Here is then an example for writing our own function.

An alternative way to compute these statistical quantities is as follows:

Chapter 3, IPython session 32

```
In [9]: from scipy import stats

In [10]: stats.mode(TheValues)
Out[10]: ModeResult(mode=array([14]), count=array([3]))

In [11]:
```

This time, the *mod* method in this *SciPy* module returns the smallest most common value of the set of values. There might be situations on which that will be fine. Nevertheless, it is better to show the full set of most common values as our program does. The message of these two output is the same: let's keep on programming!!.

3.6 Generating pseudo-random data sets for program validation and verification

We have been stressing the important fact of program validation and verification, under the premise that *a program is correct if it produces correct results*. And it is always true that we can not verify the validity of a program for all possible input data set. Thus, to be confident that our program is working correctly, it is necessary that we run it on a variety input data set. The cases we have been studying in this section (*mean*, *median*, and *mode*) has the advantage that you can find many exercises with the answer given in your Prealgebra course work and in the respective textbook. In case you find a conflicting result a few of the answers that you have, it is possible that such answer might be incorrect, but it can also be a case on which your program fails.

Accordingly, it is advisable to always verify the validity of your program in as many as possible (perhaps controlled) input data sets. This testing will help strengthening your programming thinking skills.

Python offers a well deal of options to generate controlled data set via pseudo random number generation. They are numbers that looks random but, since (via the setting the seed to generate them) they can be reproduced on another execution of the program, such numbers are not truly random.

One of the possibilities to generate sequences of pseudo random number is via the *Python* module *random*. It can be used to generate integers and non-integers pseudo random number (to see the available methods via the *random* module type *rnd.* and hit the computer **TAB**-key).

We will be using the *Python* module *random* here to produce sequences of (pseudo) random whole numbers (but you are encourage to explore the other possibilities). On an *IPython* console try the following:

Chapter 3, IPython session 33

```
In [1]: import random as rnd

In [2]: rnd.randint(0,10)
Out[2]: 1

In [3]: rnd.randint(0,10)
Out[3]: 7
```

```
In [4]:
```

On input cell `In [1]:` the functionality of the *Python* module *random* is loaded into the *IPython* session and assigned to it the name *rnd* (you could use any other name). The method that will generate whole numbers is called *randint.*. One way of using this method is shown on input cell `In [2]:` and `In [3]:` (you will get different numbers every time you run these lines of code on your *IPython* session). In general, *randint* will generate integers in any interval provided by the user. An alternative way of using this method is as follows:

Chapter 3, IPython session 34

```
In [4]: inival = 0

In [5]: endval = 10

In [6]: HowManyNumbers = 10

In [7]: myrandomlist = []

In [8]: for i in range(HowManyNumbers):
   ...:   myrandomlist.append( rnd.randint(inival, endval) )
   ...:

In [9]: myrandomlist
Out[9]: [3, 10, 4, 8, 3, 0, 8, 7, 10, 4]

In [10]:
```

In here we set the interval on which we want the pseudo random number via the variables *inival* and *endval*. We then set the numbers of random numbers to be generated in the variable *HowManyNumbers*, and then the numbers are generated and assigned to the *Python list myrandomlist*. Again, executing this code will generate a different sequence of pseudo random numbers in your computer. And every time you execute it, a different set of numbers will be generated (but always in the specified range).

As you could have anticipated, now you can generate any sequence of pseudo random numbers to feed any of the programs we have written so far. The problem with doing it this way is reproducibility, meaning that to test any of the codes you want to feed the same set of values to any other program you have to test your own one and see that both programs gives the same answer.

One way of using the same sequence of pseudo random numbers to feed any program is by using the method *seed* available in the *random* module. It is used in the following way:

Chapter 3, IPython session 35

```
In [10]: myrandomlist = []

In [11]: rnd.seed( 324567 )

In [12]: for i in range(HowManyNumbers):
    ...:   myrandomlist.append( rnd.randint(inival, endval) )
    ...:

In [13]: myrandomlist
Out[13]: [4, 10, 3, 5, 4, 0, 1, 3, 9, 7]

In [14]:
```

This means that to get the same sequence of pseudo random numbers as the ones shown on output cell `Out[13]:`, you need to set the *seed* method having as argument the number 324567 (as shown on input cell `In [11]:`) before starting the generations of the numbers. Using this *seed* you can ensure the same input to your programs. For any other reproducible sequence just use another argument in the *seed* method.

Let's finish his section by pointing out another method available in the random module. In case you want to check if the answer obtained from your program is independent of the order in which the input data is given to the program, you can use the *shuffle* method to rearrange the input data set to check for such possibility. This is done in the following way:

Chapter 3, IPython session 36

```
In [14]: rnd.shuffle( myrandomlist )

In [15]: myrandomlist
Out[15]: [3, 4, 3, 0, 10, 4, 9, 5, 7, 1]

In [16]:
```

Notice that the reordered data is set in the same *list*. Now the variable *myrandomlist* (as shown via output cell `Out[15]:`) has the same set of values arranged differently.

Now you have the necessary tool to verify the validity of your programs by using generated pseudo random input data set. Keep in mind that every time that you write a test for your program, it is also a test for whichever method you are using to check the result. More important, doing some systematic testing will develop your computational intuition to determine the validity of your program when no alternative way for checking the results given by the program is easily available.

3.7 Writing your own *Python Functions*

So far we have mentioned some built-in *Python* functions (like *max*, *len*, or *range*) and others available from other *Python* modules (like the *mode* function from the *statistics* or *SciPy* modules). In fact, *Python* (via the many modules that make its ecosystem) provides you with a great deal of ready-made functions for performing mathematical operations, text processing, web development, and so forth. There is practically no-field in which experts have worked on implementing their best recipes in pieces of *Python* software that we should reuse.

Surely, you are thinking about how you can start gaining experience in writing your own functions to make your contributions too, right?

Well, for your surprise, the small programs we have written so far are almost functions!! We only need to make a few changes to write them as formal *Python* functions, whose general format is as follows:

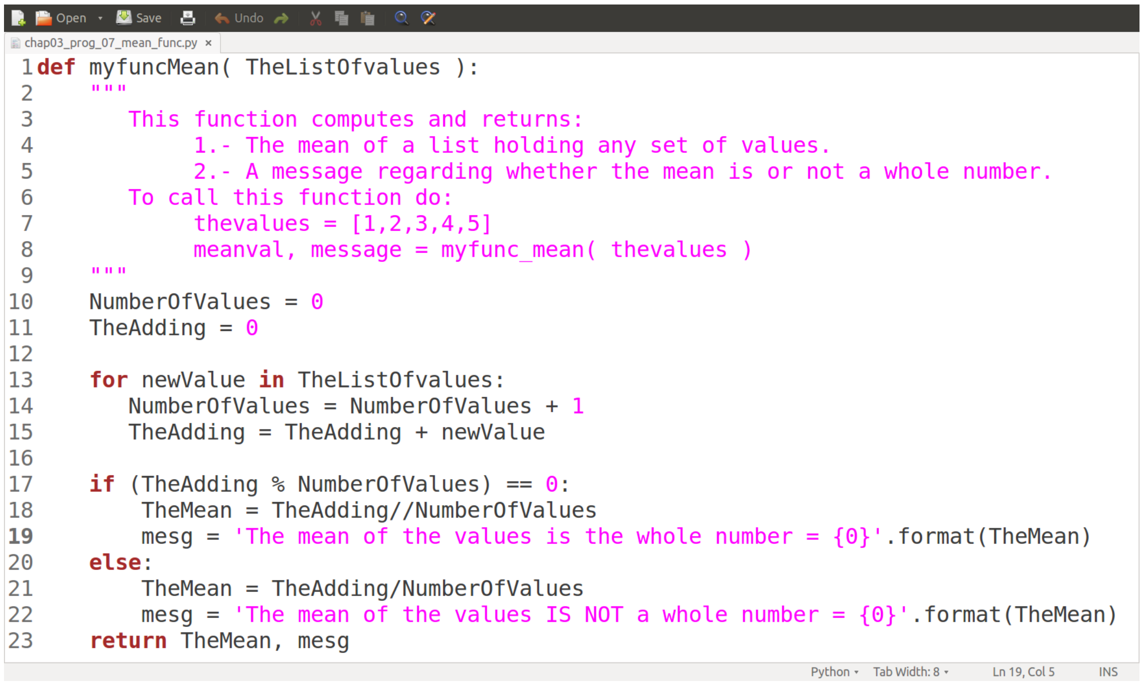
```
def FunctionName(optional set of comma-separated arguments ):
    ...
    Optional block of properly indented Python instructions
    ...
    return Optional set of Python objects
```

In this general format, the *arguments* took by *functions* are objects they can receive from the calling program to make operations on them and, then, *return*ing back to the calling program (usually with a result or new object). For example, the *len* function can receive as argument a *Python list* and returns to the calling program the number of elements in the list. Thus, calling functions is a way of forking the flow of a program (similar to using a *while*/*for loop* or an *if statement*).

From this formality, it is clear that functions are essentially a collections of *Python* instructions (statements) that, once it is made available to the calling program, using, for example, the *import* instruction, as we have done in some of the programs we have written so far, can be execute at any time in the program. In other words, *functions* are *Python* objects that encapsulates in one place a *Python* (sub) program.

When defining a function, while the arguments are optional, the instruction *def* followed by the function name (which includes the enclosing round parenthesis) is obligatory, as well as is the colon. The body of the function could be empty, and (if not sending back anything to the calling program) the *return* instruction can be omitted (the function ends after the last indented instruction). It is recommended to always use the *return* instruction to end a function.

In Figure 3.7, on page 102, it is shown changed as a function the *mean* program of Figure 3.4 (page 80). The former, is available in the directory named `chapter_03` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap03_prog_07_mean_func.py`.



```
 1def myfuncMean( TheListOfvalues ):
 2    """
 3       This function computes and returns:
 4            1.- The mean of a list holding any set of values.
 5            2.- A message regarding whether the mean is or not a whole number.
 6       To call this function do:
 7            thevalues = [1,2,3,4,5]
 8            meanval, message = myfunc_mean( thevalues )
 9    """
10    NumberOfValues = 0
11    TheAdding = 0
12
13    for newValue in TheListOfvalues:
14       NumberOfValues = NumberOfValues + 1
15       TheAdding = TheAdding + newValue
16
17    if (TheAdding % NumberOfValues) == 0:
18        TheMean = TheAdding//NumberOfValues
19        mesg = 'The mean of the values is the whole number = {0}'.format(TheMean)
20    else:
21        TheMean = TheAdding/NumberOfValues
22        mesg = 'The mean of the values IS NOT a whole number = {0}'.format(TheMean)
23    return TheMean, mesg
```

Figure 3.7: Function that returns the mean of a set of values

First, notice the general format of defining a function starting, on line 1, with the *def* instruction, followed by the name of the function *myfuncMean*; the argument *TheListOfvalues* inside parenthesis; and the colon. Then, it follows the body of the function, lines 2--22 properly indented, ending on (the also indented) line 23 with the *return* instruction, returning (in this example) two objects, the average of the values *TheMean* and a message *mesg*, to the calling program.

Second, notice the lines of code 2--9. They are comments enclosed within a triple pair of double quotes (the opening one on line 2 and the closing one on line 9) documenting the function. This is a good programming practice. You need to write comments that will help you, later on, in a few months, or even years, to understand your code (being it a function or not). This documentation is called a *doc string*, containing a short description of the purpose of the function or program and gives a brief explanation of what the different arguments and the return values are. It also should illustrates how the function is used. This *doc string* is printed to the screen when executing *FunctionName?* (in this case *myfuncMean?*) in an *IPython* session after loading the function into memory via the *import* instruction as we will be showing shortly.

Third, Notice that we have omitted the *print* instruction from the *if statement* of the function (lines of code 17--22). Instead we define a variable holding the message that the *print* instruction was sending to the computer screen. This variable can then be printed in the calling program. Are those line of code necessary?

Now, to load the function in an *IPython* session, we use the *import* instruction as we have been doing previously, when loading functions from some modules:

Chapter 3, IPython session 37

```
In [1]: from chap03_prog_07_mean_func import myfuncMean

In [2]: myfuncMean?
Signature: myfuncMean(TheListOfvalues)
Docstring:
This function computes and returns:
    1.- The mean of a list holding any set of values.
    2.- A message regarding whether the mean is or not a whole number.
To call this function do:
    thevalues = [1,2,3,4,5]
    meanval, message = myfuncMean( thevalues )
File:    ~/The_prealgebra_book/CH03/chap03_prog_07_mean_func.py
Type:    function

In [3]:
```

On input cell `In [1]:` we load the function, while on input cell `In [2]:` we display the *doc string* in case there is any available.

We can now call the function with the provided example and verify it works as advertised:

Chapter 3, IPython session 38

```
In [3]: thevalues = [1,2,3,4,5]

In [4]: meanval, message = myfuncMean( thevalues )

In [5]: meanval
Out[5]: 3

In [6]: message
Out[6]: 'The mean of the values is the whole number = 3'

In [7]: myfuncMean( thevalues )
Out[7]: (3, 'The mean of the values is the whole number = 3')

In [8]: Out[7]
Out[8]: (3, 'The mean of the values is the whole number = 3')

In [9]: type(Out[7])
Out[9]: tuple

In [10]: Out[7][0] = 23
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-ba7001730f56> in <module>()
----> 1 Out[7][0] = 23

TypeError: 'tuple' object does not support item assignment

In [11]:
```

On input cell `In [3]:` the set of values is assigned to the variable *thevalues*, which is then passed as argument to the function. We could use any other valid name for the list of values. Then, on input cell `In [4]:` the function if called, assigning its returned objects to the variables *meanval* and *message*. We know we need to call the function that way from its documentation in the *doc string*. In case we don't need to use the objects returned by the functions, we can call it without assigning its returned values to a new defined variable, as is done on input cell `In [7]:` (remember that the output is assigned to the corresponding output cell, in this case `Out[7]`, as can be seen on input cell `In [8]:`). Notice that the output is delivered enclosed in parenthesis, which are provided by the *Python* interpreter as we did not write them in our function. This *Python* object is called a *tuple* (see output cell `Out[9]:`), they behave similarly like a *Python list* with the important difference that we can not change any of its elements (see input cell `In [10]`, according to which a *tuple* is called an *immutable* object.

To use our function on the input values used in our program on page 80, we do the following:

Chapter 3, IPython session 39

```
In [11]: TheValues = [17, 14, 14, 16, 15, 16, 14, 15, 13, 18, 13]

In [12]: TheMean, message = myfuncMean( TheValues )

In [13]: TheMean
Out[13]: 15

In [14]: message
Out[14]: 'The mean of the values is the whole number = 15'

In [15]:
```

A few words of caution about the variables inside a function are in order. First, do not let undefined variables inside any function. Always initialize them to a value. The reason is that the *Python* interpreter will try to assign to non initialized variables inside any function (called *local* variables) a value from the calling program (called *global* variables), before issuing an error for trying to use a variable that does not have a value assigned to it. One way to avoid this is to consistently use a convention to name your variables inside a function and another convention to name your variables in the main program. Second, name your functions in a way that do not match any existing built-in *Python* function or from any other module you use in your program. Following this little advice can save you from unnecessary headaches debugging your code.

There is much more to say about writing your own *Python* functions, but for now what we have done is sufficient to help you develop computational thinking independence by engaging yourself in effective learning via the writing as functions the programs to compute the *median* (on page 88) and the *mode* (on page 95) of a set of values, taking care of cleaning your functions from unnecessary *print* instructions. Check also the exercises to improve such programs, and verify that each function works as expected.

Further discussions on functions will be given as the need to use them arise along the book development. We encourage you to satisfy your appetite on knowing more about functions consulting the references given at the end of this chapter, starting on page 144.

3.8 Miscellaneous application programs

This section includes a few programs that will help you to further explore some topics of your Prealgebra course work. Accordingly, you can read aboout many of the topics presented in your Prealgebra textbook and also on Wikipedia.

The several recipes presented in this (and other) sections of the book gives you the opportunity to challenge your understanding of any of the topics presented. Taking for granted an active learning approach, as you read them and work thorough its provided *Python* implementation, you might divise ways to improve such recipes. Moreover, the *Python* infinite precision with whole numbers make it possible to further analyze some results that via hand computation are almost impossible or very tedious to do. As you learn more about *Python*, you will find that practically each one of the functions of this book has already been efficiently implemented in some *Python* modules. For now, the idea is that you do your own correct implementation via the elementary *Python* instructions we have learned so far regardless of efficiency.

By the way, be ware that many of the programs in this section applies only to whole numbers (excluding zero, in some cases). Consequently, this set of programs are missing a check to ensure (or enforce) that the input data is of the right type for them to work properly. Later on (in the book) we will learn a bit about how to use *assertion statements* to make such verification at the beginng of any program.

3.8.1 Factors of a whole number

From your Prealgebra course work, it is said that given whole number a, b and c such that $a \times b = c$, then a and b are factors or divisors of c. Consequently, the factors of a whole number are the numbers dividing it evenly (the remainder is zero).

Thus, an straightforward recipe to find the factors of a whole number is a follows:

1. Set the *number*
2. Set a variable to hold the list of factors (i.e. $ListOfFactors = []$)
3. set $n = 1$
4. Find the remainder of dividing the *number* by n
5. If the remainder if equal to zero, includes n to the $ListOfFactors$
6. Increase n by one
7. Repeat steps 4--7 if $n \leq number$.
8. Print to screen the list of factors of the *number*.

This recipe is implemented in the function shown in Figure 3.8, on page 108. You can find this function in the directory named `chapter_03` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap03_prog_08_factor_func.py`.

We encourage you to follow the flow of the code in the function. What follows is an example of using this function:

Chapter 3, IPython session 40

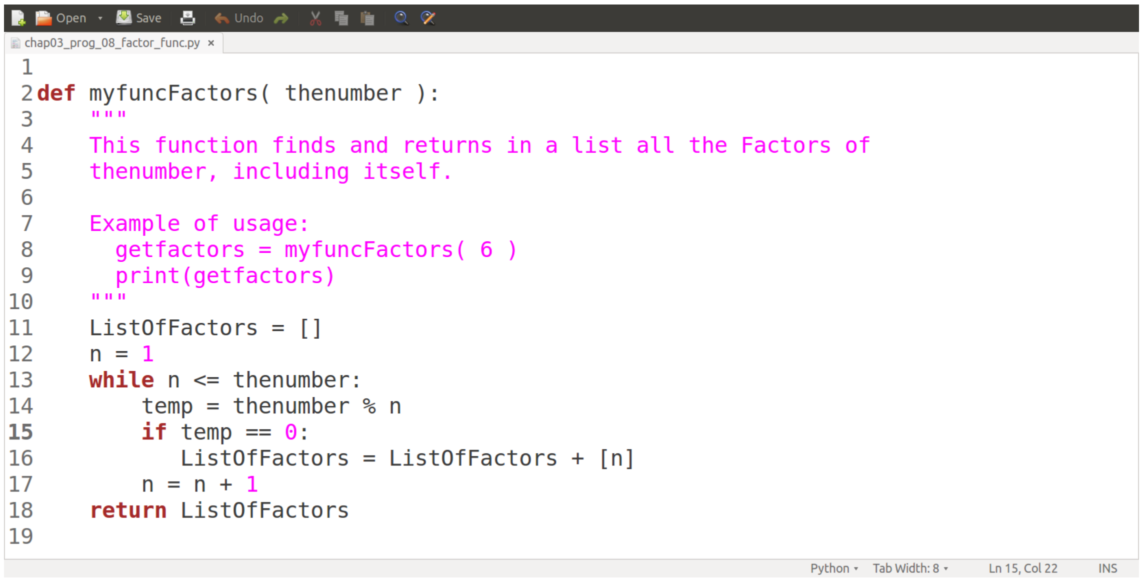
```
In [1]: from chap03_prog_08_factor_func import myfuncFactors

In [2]: myfuncFactors?
Signature: myfuncFactors(thenumber)
Docstring:
This function finds and returns in a list all the Factors of
thenumber, including itself.

Example of usage:
  getfactors = myfuncFactors( 6 )
  print(getfactors)
File:    ~/The_prealgebra_book/CH03/chap03_prog_08_factor_func.py
Type:    function

In [3]: getfactors = myfuncFactors( 6 )

In [4]: print(getfactors)
[1, 2, 3, 6]

In [5]:
```

Alternatively, one can make available the function in an *IPython* session as follows (it is recommended that you start a new *IPython* console):

Chapter 3, IPython session 41

```
In [1]: %run chap03_prog_08_factor_func.py

In [2]: getfactors = myfuncFactors( 6 )

In [3]: print(getfactors)
[1, 2, 3, 6]

In [4]:
```

In some of the following sections we will write programs reusing this function.



```python

def myfuncFactors( thenumber ):
    """
    This function finds and returns in a list all the Factors of
    thenumber, including itself.

    Example of usage:
      getfactors = myfuncFactors( 6 )
      print(getfactors)
    """
    ListOfFactors = []
    n = 1
    while n <= thenumber:
        temp = thenumber % n
        if temp == 0:
           ListOfFactors = ListOfFactors + [n]
        n = n + 1
    return ListOfFactors

```

Figure 3.8: Function that returns the factors of a whole number

3.8.2 Is this a prime number?

A *prime number* is any number having as only factors one and the number itself. The literature on prime numbers is immensely huge, and you will find out there a great jungle on computer codes to find them.

From its definition, you can see immediately that we can use our function that find the factors of a number to decide whether a given whole number is or not prime. Certainly, this function is not optimal for this task and you are encourage to find better ways on doing it. After working out this section you could continue by trying exercise 3.5, on page 141.

Thus, an straightforward recipe to find if a given number is prime or not is as follows:

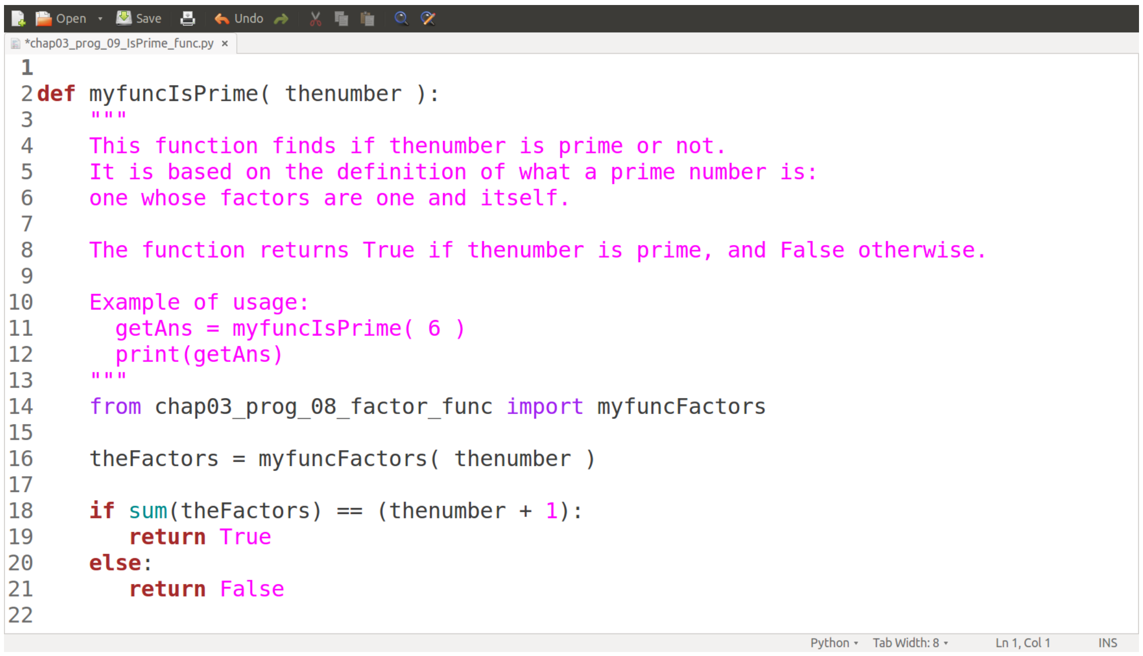
1. Set the *number*.
2. Find the factors of the given *number*.
3. Add the factors.
4. If the addition of the factors equals the addition of one and the *number* itself, returns True (for yes, the *number* is prime). Otherwise returns False (for no, the *number* isn't prime).
5. Print to screen the result.

This recipe is implemented in the function shown in Figure 3.9, on page 109. You can find this function in the directory named `chapter_03` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap03_prog_09_IsPrime_func.py`.

We encourage you to follow the flow of the code in the function. What follows is an example of using this function:

Chapter 3, IPython session 42

```
In [1]: %run chap03_prog_09_IsPrime_func.py

In [2]: getAns = myfuncIsPrime( 11 )

In [3]: getAns
Out[3]: True

In [4]:
```


*chap03_prog_09_IsPrime_func.py

```
def myfuncIsPrime( thenumber ):
    """
    This function finds if thenumber is prime or not.
    It is based on the definition of what a prime number is:
    one whose factors are one and itself.

    The function returns True if thenumber is prime, and False otherwise.

    Example of usage:
      getAns = myfuncIsPrime( 6 )
      print(getAns)
    """
    from chap03_prog_08_factor_func import myfuncFactors

    theFactors = myfuncFactors( thenumber )

    if sum(theFactors) == (thenumber + 1):
       return True
    else:
       return False
```

Figure 3.9: Function returning if a whole number is prime or not.

3.8.2.1 Generating prime numbers: the Sieve of Eratosthenes algorithm

The generation of prime numbers is As already mentioned, the literature on prime numbers is immensely huge indicative that this subject has kept captivated mathematicians for long time, continuing nowadays. A major topic of interest is how to generate these numbers efficiently.

A long standing simple clever method to generate prime numbers is the famous *sieve algorithm*, invented in the ancient city of Alexandria by Eratosthenes of Cyrene (about 276–296 B.C.). It is the most efficient way to list all the primes in the range 2 through up to a few millions n, involving the following steps, suitable for hand, pencil and paper, calculation (recall that a *prime number* is any number having as only factors one and the number itself):

1. Enumerate the numbers from 2 up to n.
2. Circle 2 as the first prime number, and cross out from the list every second number (these numbers are evenly divided by 2 and then, according with its definition, are not prime numbers).
3. Circle the first not crossed out number (3) as the second prime number, and cross out from the list all higher multiple of 3 (6, 9, 12, 15, and son on).
4. Circle the first not crossed out number (5) as the next prime number, and cross out from the list all higher multiple of 5 (10, 15, 20, 25, and so on).
5. Circle the first not crossed out number (7) as the next prime number, and cross out from the list all higher multiple of 7 (14, 21, 28, 35, and so on).
6. Continue repeating this procedure until the first not crossed out number in the list is the one whose square is greater than n.
7. All the numbers that has been circle out are the primes from 2 through n.

A moment of thought about this method, an straightforward computer implementation of it goes as follows:

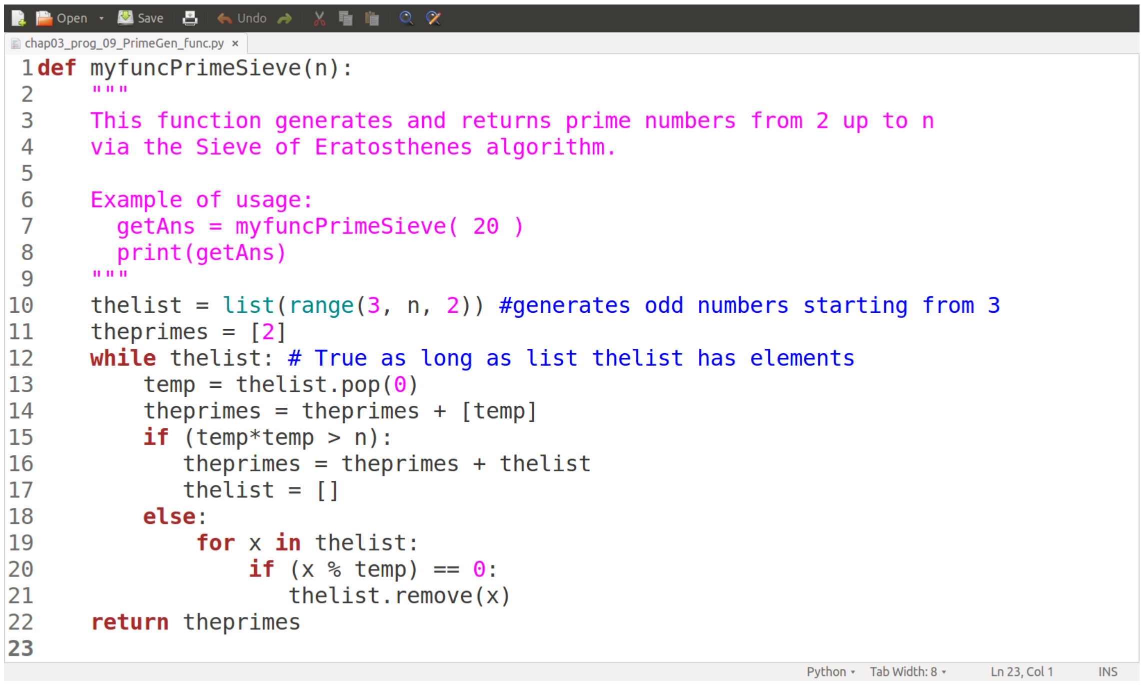
1. Initialize a list (i.e. *theprimes*) holding 2 as the first prime.
2. Initialize a list (i.e. *thelist*) holding all the odd numbers from 3 through n.
3. Remove from *thelist* its first number and append it to the *theprimes* list.
4. If the square of the just appended number to the *theprimes* list (in step 3) is greater than n, merge both lists: the *theprimes* and the *thelist*. Set *thelist* to be empty.
5. If the square of the just appended number to the *theprimes* list (in step 3) is not greater than n, go through *thelist* and remove from it all the elements evenly divided (remain of zero) by the just appended element in *theprimes* list.
6. Repeat steps 3--6 until *thelist* is empty.
7. Print to screen *theprimes*.

This recipe is implemented in the function shown in Figure 3.10, on page 111. You can find this function in the directory named `chapter_03` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap03_prog_09_PrimeGen_func.py`.

We encourage you to follow the flow of the code in the function. What follows is an example of using this function:

Chapter 3, IPython session 43

```
In [1]: from chap03_prog_09_PrimeGen_func import myfuncPrimeSieve

In [2]: primes = myfuncPrimeSieve(50)

In [3]: primes
Out[3]: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

In [4]:
```



```python
def myfuncPrimeSieve(n):
    """
    This function generates and returns prime numbers from 2 up to n
    via the Sieve of Eratosthenes algorithm.

    Example of usage:
      getAns = myfuncPrimeSieve( 20 )
      print(getAns)
    """
    thelist = list(range(3, n, 2)) #generates odd numbers starting from 3
    theprimes = [2]
    while thelist: # True as long as list thelist has elements
        temp = thelist.pop(0)
        theprimes = theprimes + [temp]
        if (temp*temp > n):
           theprimes = theprimes + thelist
           thelist = []
        else:
            for x in thelist:
                if (x % temp) == 0:
                   thelist.remove(x)
    return theprimes

```

Figure 3.10: Function returning prime numbers from 2 up to n.

3.8.3 Is this an abundant number?

We know how to find the factors of a whole number, which includes the number itself. By withdrawing the number itself from its factors, we are left with the *proper divisors* of the number.

An *abundant number* is any whole number. whose proper divisors add up to more than the number itself.

From its definition, you can see immediately that we can use our function that find the factors of a number to decide whether a given whole number is or not abundant via the following recipe:

1. Set the *number*.
2. Find the factors of the given *number*.
3. Find the addition of the proper divisors by adding the factors and subtracting the *number*.
4. If the addition of the proper divisors is greater than the *number* itself, returns True (for yes, the *number* is abundant). Otherwise returns False (for no, the *number* isn't abundant).
5. Print to screen the result.

This recipe is implemented in the function shown in Figure 3.11, on page 113. You can find this function in the directory named `chapter_03` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap03_prog_10_IsAbundant_func.py`.

We encourage you to follow the flow of the code in the function. What follows is an example of using this function (see exercise 3.8, on page 142, for an extra practice problem):

Chapter 3, IPython session 44

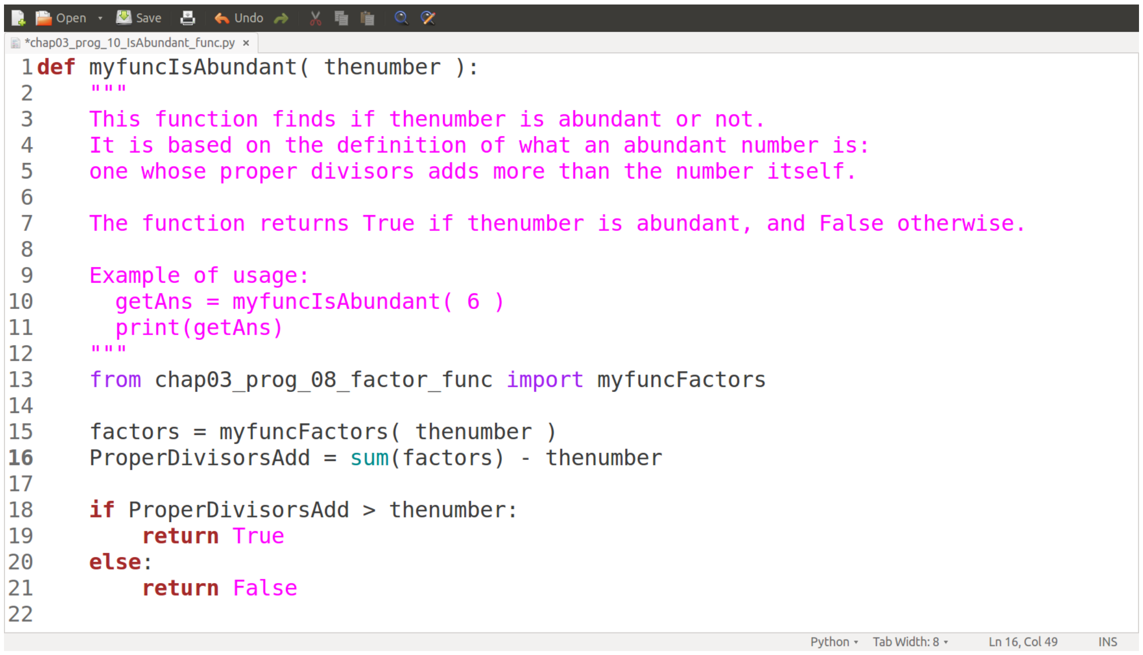
```
In [1]: from chap03_prog_10_IsAbundant_func import myfuncIsAbundant

In [2]: getAns = myfuncIsAbundant( 11 )

In [3]: getAns
Out[3]: False

In [4]:
```

3.8.4 Is this a perfect number?

We know how to find the factors of a whole number, which includes the number itself. By withdrawing the number itself from its factors, we are left with the *proper divisors* of the number.

A *perfect number* is any whole number. whose proper divisors add up to exactly the number itself.

From its definition, you can see immediately that we can use our function that find the factors of a number to decide whether a given whole number is or not perfect via the following recipe:



```python
def myfuncIsAbundant( thenumber ):
    """
    This function finds if thenumber is abundant or not.
    It is based on the definition of what an abundant number is:
    one whose proper divisors adds more than the number itself.

    The function returns True if thenumber is abundant, and False otherwise.

    Example of usage:
      getAns = myfuncIsAbundant( 6 )
      print(getAns)
    """
    from chap03_prog_08_factor_func import myfuncFactors

    factors = myfuncFactors( thenumber )
    ProperDivisorsAdd = sum(factors) - thenumber

    if ProperDivisorsAdd > thenumber:
        return True
    else:
        return False
```

Figure 3.11: Function returning if a whole number is abundant or not.

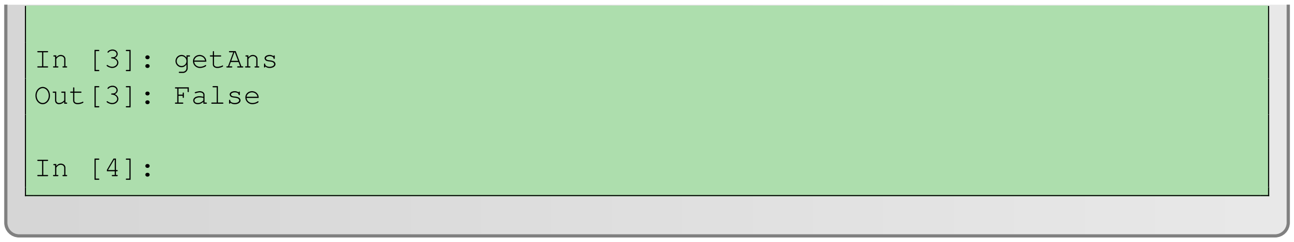
1. Set the *number*.
2. Find the factors of the given *number*.
3. Find the addition of the proper divisors by adding the factors and subtracting from it the *number*.
4. If the addition of the proper divisors is equal to the *number* itself, returns True (for yes, the *number* is perfect). Otherwise returns False (for no, the *number* isn't perfect).
5. Print to screen the result.

This recipe is implemented in the function shown in Figure 3.12, on page 114. You can find this function in the directory named `chapter_03` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap03_prog_11_IsPerfect_func.py`.

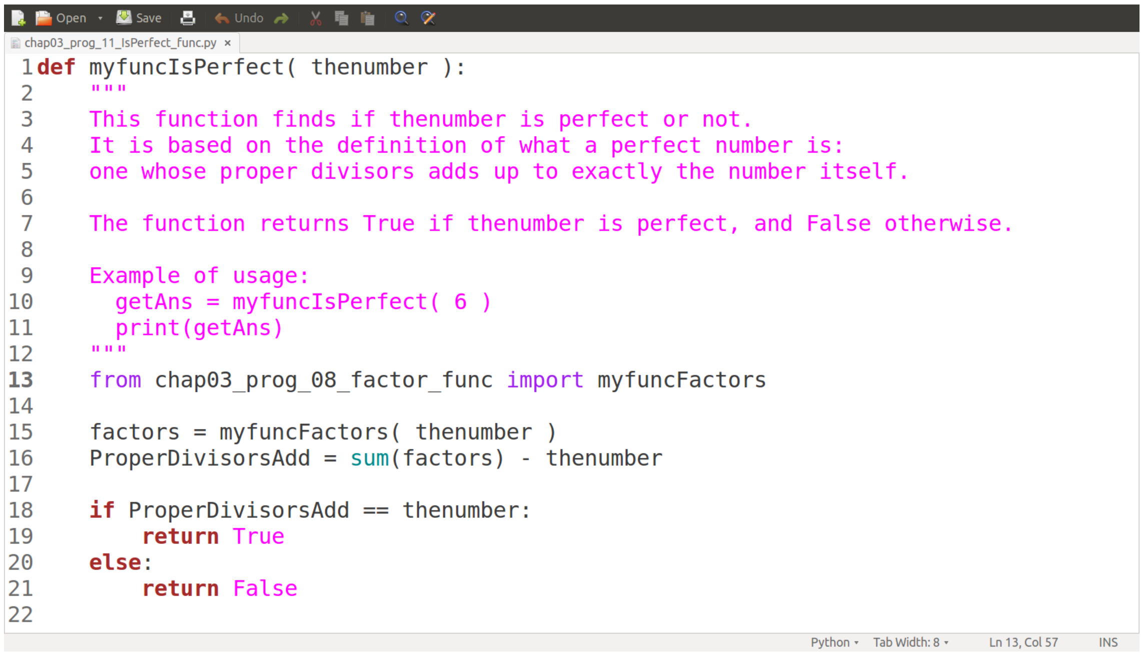
We encourage you to follow the flow of the code in the function. What follows is an example of using this function (see exercise 3.9, on page 142, for an extra practice problem):

Chapter 3, IPython session 45

```
In [1]: from chap03_prog_11_IsPerfect_func import myfuncIsPerfect

In [2]: getAns = myfuncIsPerfect( 11 )
```



```
In [3]: getAns
Out[3]: False

In [4]:
```



```python
def myfuncIsPerfect( thenumber ):
    """
    This function finds if thenumber is perfect or not.
    It is based on the definition of what a perfect number is:
    one whose proper divisors adds up to exactly the number itself.

    The function returns True if thenumber is perfect, and False otherwise.

    Example of usage:
      getAns = myfuncIsPerfect( 6 )
      print(getAns)
    """
    from chap03_prog_08_factor_func import myfuncFactors

    factors = myfuncFactors( thenumber )
    ProperDivisorsAdd = sum(factors) - thenumber

    if ProperDivisorsAdd == thenumber:
        return True
    else:
        return False

```

Figure 3.12: Function returning if a whole number is perfect or not.

3.8.5 Greatest common divisor (GCD) or greatest common factor (GCF) of natural numbers

The *greatest common divisor* (GCD) (also known as *greatest common factor*) of a set of whole numbers is the greatest number that divides all of the given numbers exactly (with no remainder).

An straightforward method to find the GCD is by perusing the listing of the factors of the given numbers. One way of executing this idea to find the GCD value is by comparing the factors of the smaller number with the factors of the rest, and taking apart the common values (the ones that appear in all of them). Then, take the maximun value of this common factor list. A detailed recipe might go as follows:

1. Set the *numbers*.

2. Sort the *numbers* from smaller to larger.
3. Find the factors of the smaller number and set it to a particular variable (like *smallestNumFactors*).
4. Collect the factors of the other numbers in a different variable (like *othersNumFactors*).
5. Compare the values in *smallestNumFactors* with each one of the list for each other number in *othersNumFactors*. Collect the common factors among the *numbers*.
6. Get the maximum value among the common factors of the *numbers*.
7. Print to screen the result.

This recipe is implemented in the function shown in Figure 3.13, on page 116. You can find this function in the directory named `chapter_03` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap03_prog_12_GCD_func.py`.

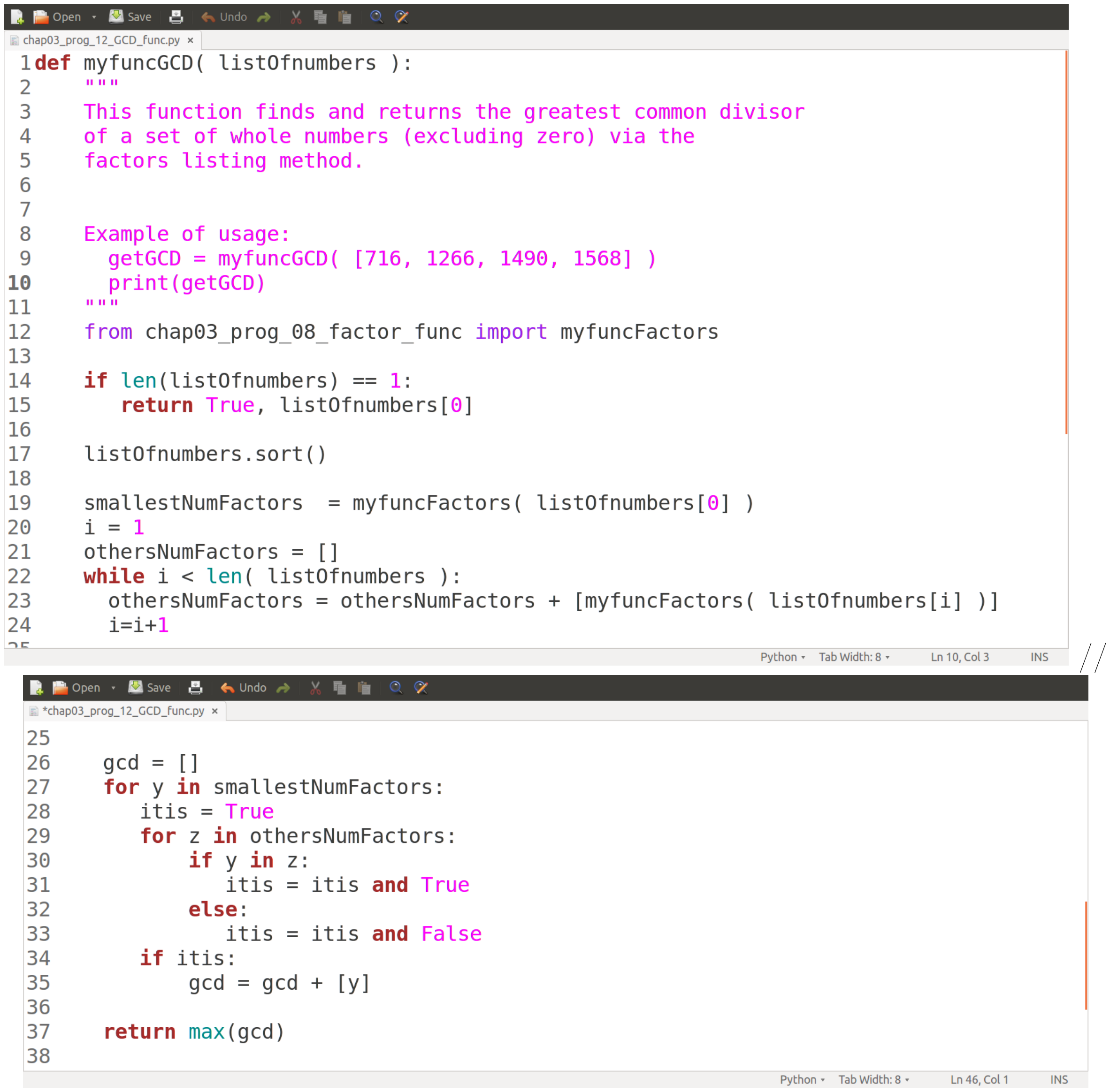
We encourage you to follow the flow of the code in the function. Let's point out a general description of it. (the referred line numbers correspond to the numeration of the lines of Figure 3.13, on page 116).

In line of code 1 introduced the definition of the function. Then, in lines of codes 2--11, a brief description of the function is given. Following line of code 12, the function to find the factors on a whole number is made available to be used in this function. Lines of code 14--15 just check if only one number is contained in the variable *listOfNumbers*, holding the set of numbers to which the GCD value is being seek. In that case the function ends returning that value to the calling program of this function. Then follows line of code 17 on which the given set of numbers are sorted in ascending order (from small to larger). In line of code 19 the factors of the smallest number are assigned to the variable *smallestNumFactors*.

In lines of code 20--24 the factors of the others numbers are collected in a list of lists and assigned to the variable *othersNumFactors*. This means that the elements in this list are *list* objects (and not naive numbers as we have been using so far). This is an example, as mentioned before, that a *Python list* can hold any type of *Python* objects, given an idea of the flexibility and powerfulness of the language. Let's illustrate a bit more this idea in an *IPython* session:

Chapter 3, IPython session 46

```
In [1]: list1 = [1,2,3,4]

In [2]: list2 = [6,5,4,3,2,1]

In [3]: list3 = [10,12]

In [4]: together = [ list1, list2, list3 ]
```



```python
def myfuncGCD( listOfnumbers ):
    """
    This function finds and returns the greatest common divisor
    of a set of whole numbers (excluding zero) via the
    factors listing method.


    Example of usage:
      getGCD = myfuncGCD( [716, 1266, 1490, 1568] )
      print(getGCD)
    """
    from chap03_prog_08_factor_func import myfuncFactors

    if len(listOfnumbers) == 1:
       return True, listOfnumbers[0]

    listOfnumbers.sort()

    smallestNumFactors  = myfuncFactors( listOfnumbers[0] )
    i = 1
    othersNumFactors = []
    while i < len( listOfnumbers ):
      othersNumFactors = othersNumFactors + [myfuncFactors( listOfnumbers[i] )]
      i=i+1

    gcd = []
    for y in smallestNumFactors:
       itis = True
       for z in othersNumFactors:
           if y in z:
              itis = itis and True
           else:
              itis = itis and False
       if itis:
           gcd = gcd + [y]

    return max(gcd)

```

Figure 3.13: Function returning the Greatest Common Divisor (GCD) of a set of whole numbers.

```
In [5]: together
Out[5]: [[1, 2, 3, 4], [6, 5, 4, 3, 2, 1], [10, 12]]

In [6]: together[0]
Out[6]: [1, 2, 3, 4]

In [7]: together[1]
Out[7]: [6, 5, 4, 3, 2, 1]
```

```
In [8]: together[2]
Out[8]: [10, 12]

In [9]: together[2][1]
Out[9]: 12

In [10]:
```

On input cells `In [1]:`--`In [4]:` the usual way of assigning values to a *list* is used to assign some *list* objects to the respective variables on the left hand side. Notice the content of the *list* assigned to the variable *together* on input cell `In [5]:`, and how we can take the values on it via the usual indexing of its first level elements (input cells `In [6]:`--`In [8]:`) which are *list* objects. Since these elements are *list* objects them self, their objects can be obtained using the usual indexing, as shown on input cell `In [9]:`. We encourage you to continue such exploration to get used to this way maintaining different objects in a one variable of type *list*.

Continuing with the description of our function, now comes lines of code 26--35. To compare the factors of the smaller number with those of the others and take apart the common ones, one external *for loop* is used to go through the factors of the smaller value, while an internal *for loop* is used in order to go through each set of the factors of the other numbers as a whole (not one by one as the external *for loop* does). To find out the common factors, notice the new usage of the *in* statement on the line of code 30 (*if y in z*:). This line of code tells whether the value *y* from the *smallestNumFactors* is contained in the current *list* held by the internal *for loop* variable *z*. Notice also the use of the boolean variable *itis* that is used to confirm or not the check. If, after exiting the internal *for loop*, this boolean variable holds the value of *True* on line of code 34--35, the value *y* is appended to the *gcd list*, holding the common factors that we are searching for. Finally, the function ends returning the maximun common factor in the *gcd list* to the calling program of this function.

What follows is an example of using this function (for further practice see exercises 3.10--3.11, on page 142):

Chapter 3, IPython session 47

```
In [1]: from chap03_prog_12_GCD_func import myfuncGCD

In [2]: listOfnumbers = [716, 1266, 1490, 1568]

In [3]: getAns = myfuncGCD(listOfnumbers)

In [4]: getAns
Out[4]: 2
```

```
In [5]:
```

You can verify by hand (pencil and paper) computation that this answer is correct. See exercise 3.11, on page 142 for an alternative way of verifying this answer or the next section.

3.8.6 Prime Factorization of whole numbers

The prime factorization of a number is the expression of the number as a product of its prime factors. The standard way to find the prime factors of a whole number start with the smallest prime number as a trial divisor and continue with prime numbers as trial divisors until the final quotient is prime.

At first sight sight, implementing this algorithm by hand (pencil and paper) computation should require to have handy a table of ordered prime numbers. Nevertheless, a moment of though will lead to implementing this method of prime factorization by hand (pencil and paper) computation, by an increase of one by one of the dividend, as follows: we can start dividing the given number by the first prime number 2 and, if the remainder is zero, take the quotient and repeat dividing by 2 until the quotient is not longer exactly divided by it. This process will also exhaust any possibility of further dividing exactly the number by any multiple of 2 (like $4 = 2 \times 2$, $6 = 2 \times 3$, $8 = 2 \times 2 \times 2$, $10 = 2 \times 5$, and so forth). Reaching this point, goes dividing the quotient by the next number which is 3 (that happen to be the next prime number after 2), and again exhaust the process of exactly dividing each obtained quotient by 3, until the quotient is not longer exactly divided by 3. This process will also exhaust any possibility of further dividing exactly the number by any multiple of 3 (like $6 = 2\times3$, $9 = 3\times3$, $12 = 2\times2\times3$, $15 = 3 \times 5$, and so on). The next number to try the division of the quotient is dividing it by 4. But this will not yield an exact division because it was already exhausted when dividing by 2. Then comes dividing the quotient by the next prime number 5. We will keep doing that division by 5 until the remainder is not longer 0. This procedure, as with the previous cases, will also exhaust any possibility of having a number that could be divided exactly by any of the multiples of 5 (like $10 = 2 \times 5$, $15 = 3 \times 5$, $20 = 2 \times 2 \times 5$, $25 = 5 \times 5$, etcetera). Continuing in this way, this procedure will get you to be dividing only by prime numbers, as required by the algorithm. Remember that the procedure will stop whenever a quotient of 1 is reached, which eventually will happen.

An straightforward implementation of this recipe is as follows:

1. Initialize a variable (i.e. n) holding the number we want its prime factorization.
2. Set a variable holding the first prime number (i.e. i).
3. Set a *list* to hold the prime factors of the number (i.e. *factors*).
4. Get the remainder of the dividing n by i.
5. If the remainder is zero, append i to the *list* of *factors* and set n to be the quotient of dividing its current value by i. Otherwise, increase i by one.

6. Repeat steps 4--6 until n gets the value one.
7. Print to screen *theprimes*.

This recipe is implemented in the function shown in Figure 3.14, on page 120. You can find this function in the directory named `chapter_03` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap03_prog_13_PrimeFactorization_func.py`.

We encourage you to follow the flow of the code in the function. What follows is an example of using this function:

Chapter 3, IPython session 48

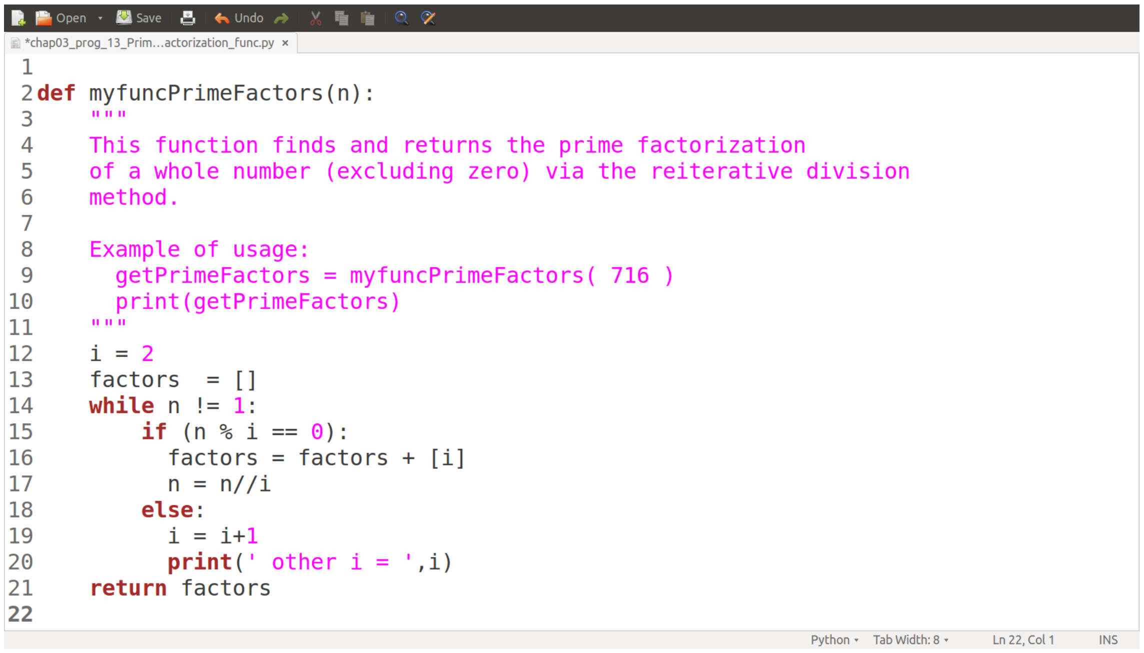
```
In [1]: from chap03_prog_13_PrimeFactorization_func import
   myfuncPrimeFactors

In [2]: thefactors = myfuncPrimeFactors(50)

In [3]: thefactors
Out[3]: [2, 5, 5]

In [4]:
```

How can you verify that this function is returning the correct prime factorization result? (see exercise 3.13, on page 142.

3.9 Solving equations involving whole numbers via *SymPy*

In this section we will introduce the basic elements of solving equations in *Python* via *SymPy* algebraically. Though the context will be limited to whole numbers, everything learned here works also for any set of numbers, including generic symbols. Certainly, the solutions are limited to the set of values the symbols could take. Consequently, after reading this section, you might want to read also the corresponding sections about equations with integers (page 176), fractions (page 195), and decimals (page 230). You could also practice your skills in solving equations via the one variable equation solver described on page 236.

Let's first remain you that this book is intended for you to enhance your learning experience of your Prealgebra course content via *Python* programming (this is not a book for you to become a programmer). Consequently, as mentioned earlier, one of the goals of the book is that you be able to apply what you learn here to other courses of your educational track and in



```python

def myfuncPrimeFactors(n):
    """
    This function finds and returns the prime factorization
    of a whole number (excluding zero) via the reiterative division
    method.

    Example of usage:
      getPrimeFactors = myfuncPrimeFactors( 716 )
      print(getPrimeFactors)
    """
    i = 2
    factors  = []
    while n != 1:
        if (n % i == 0):
          factors = factors + [i]
          n = n//i
        else:
          i = i+1
          print(' other i = ',i)
    return factors

```

Figure 3.14: Function returning the prime factorization of a whole number.

your professional work development. Consequently, before dealing with equations via *Python* programming, make sure you understand the procedures to find their solutions as presented in your Prealgebra course work. More important, whenever you are confronted with an equation the first questions you might ask is where I have seen this equation before? If you remember so, you can then further ask, can I make an algorithm from the steps I have used to solve it before? Also, keep in mind that equations are not merely a mean to find a numerical value. Equations can be used to guide our thinking and reasoning, so we can find connections between the different parts of the problem they comes from (which is usually a word problem from the Engineerings and/or the Sciences). Later in your educational track you will be amaze that very simple equations could lead to a very rich dynamics (for an example, check the references on the *logistic map* at the end of this chapter).

Knowing that the social environment has a (usually neglected) strong influence in the process of teaching and learning, let's further point out that we need to be cautious when processing opinions such as that the equations your are dealing with has nothing to do with real world applications or that what is important is *conceptual understanding* over knowing how to solve equations. From our perspective, such opinions needs to be taking as if we were handling nails. On the first opinion underly the idea of starting a fast race without warming up. Someone will certainly be hurt following trying to do so. The second comment is even worse because it is a subtle (sound) one. It is like a drop of water silently (passing unnoticed) eroding the basis of a building. It ignores that our conceptual understanding actually comes from the study of the phenomena after this was described by equations that goes refined over the years as we get better and wider data about the phenomena. In spite of how important and passionate this discussion might be, to continue with our topic of finding the solution of equations via *SymPy*, let's refer you to the references at the end of this chapter in case you have interest in a deeper view of the meaning of equations and how they are responsible of our conceptual understanding of nature.

From your Prealgebra course work, an equation expresses the equality of two mathematical expressions in the generic form: **`left hand side (LHS) = right hand side (RHS)`**. Some examples of simple, one variable (x), equations are:

$$x + 6 = 11 \tag{3.1}$$
$$3x - 2 = 16 \tag{3.2}$$
$$5x - 1 = 3x + 15 \tag{3.3}$$
$$\frac{x-2}{4} + 5 = \frac{2(1-x)}{5} + 8 \tag{3.4}$$

In these equations the numeric values are called coefficients, receiving the particular name on independent coefficients the numerical values not appearing multiplying the variable x.

When dealing with any equation, one neds to think about the meaning of the elements in the equation. Take for instance the equal sign: How do you understand it? In most cases, you might think of the equal sign in terms of representing an arithmetic identity, in the sense that the equations you are studying in your Prealgebra course work were build for you to become acquainted with methods to find the unknownw (variable) in the equation, so both sides of the equation balance (becomes an arithmetic identity). But in other cases the equal sign helps you to use parts of the equation in any other computation by substitution. Take for example a rectangular box of side a, b, and c. The volumen of this rectangular box is $V = abc$. But you could also wirte this volume using the surface of any of the face. Take, for example, the fae of sides a and b having as surphace $S_{ab} = ab$. Using this expression, because of the equal sign, one could write $V = S_{ab}c$ as an equivalent expression for the volume of the box.

Consisten with the previous idea, as the solution to any of the just listed (an any other) equations we refers to the value that can be assigned to the variable x so that a true statement (arithmetic identity) results. For instance, the value 5 satisfies the equation $x+6 = 11$, because, when x takes the value of 5 ($x = 5$), a true statement (arithmetic identity) results ($5+6 = 11$). Any other value taken by x will result in a false statement. In these equations we have used x as the variable, but we could have used any other symbol (that could be used as a variable name in *Python*). Consequently, to solve an equation means to find the solution of the equation (in this section we will be dealing with equations that can have one and only one solution).

In general symbolic terms, the example of one variable (x) equations presented above can be casted in more general terms using non-numeric or symbolic coefficients a, b, c, or d can be written as:

$$x + a = b \tag{3.5}$$
$$ax + b = c \tag{3.6}$$
$$ax + b = cx + d \tag{3.7}$$

We encourage you to pay extreme attention in your Prealgebra course work to the methods used to solve equations. *Python* and any other software will hide the solution methods from you, unless you carry them step by step, as if you were dealing with solving the equation by hand (using pencil and paper). We are not going to do that here. Instead, we will learn how to setup the equations so *Python*, via the *SymPy* module, can give you the answer, which then you can check to be sure you are getting the correct answer. But you will not know at first sight how *SymPy* found the answer.

Accordingly, we insist, you need to understand the process of solving equations as explained in your Prealgebra course work. For one reason, that it will enrich you problem-solving though processes. For another, you might need to apply them in situations were you do not have available any computer or you might be working in places with not enough budget to buy them. And yet another one, for fun, to remind yourself that you are a human being. Doing so will means you have gained understanding that goes beyond the performing of rote manipulation of the symbols in an equation. In this way you will be able to apply this knowledge of solving equations in other subjects of your educational track (like Physics, Chemistry, Biology, Psychology, and the others)

In concordance with the preceding discussion, what follows is a brief introductions on how you could use *SymPy* to find the solution of an equation. You could use it as a way to explore solutions of more general equations.

Let's get started with the basic functionality to solve equations via *SymPy*. First, load into a current *IPython* session the *SymPy* functionality:

Chapter 3, IPython session 49

```
In [1]: from sympy import symbols, Eq, solveset

In [2]: x, a, b, c = symbols('x, a, b, c')

In [3]: type(a)
Out[3]: sympy.core.symbol.Symbol

In [4]:
```

Here, on input cell `In [1]:`, we make available into the current *IPython* session the functions from *SymPy* (the order of them in the *Python* instruction on input cell `In [1]:` is irrelevant):

1. *symbols*: as we have used it before, it can be used to define symbolic variables, as is done for the variables x, a, b, and c on input cell `In [2]:`, and confirmed on output cell `Out[3]:` for one of them (the symbol a).

You can read more about *symbols* by executing on an *IPython* cell the instruction **`symbols?`** or **`symbols??`**.

2. *Eq*: which is used to setup the equation to which its solution we are seeking. It is used in the form **`Eq(LHS, RHS)`**.
   You can read more about *Eq* by executing on an *IPython* cell the instruction **`Eq?`** or **`Eq??`**.
3. *solveset*; which is the recommended way to find the solution of an equation in *SymPy*. Other alternative is to use the function *solve*, which will let you to explore on your own. It is used in the form **`solveset(equation, variable)`**.
   You can read more about *solveset* by executing on an *IPython* cell the instruction **`solveset?`** or **`solveset??`**.

To find the solution of the equation $x - 17 = 20$, after identifying that $LHS = x - 17$ and $RHS = 20$, we solve it as follows:

Chapter 3, IPython session 50

```
In [4]: LHS = x - 17

In [5]: RHS = 20

In [6]: thesol = solveset( Eq(LHS, RHS), x)

In [7]: thesol
Out[7]: {37}

In [8]: type(thesol)
Out[8]: sympy.sets.sets.FiniteSet

In [9]: thesol = list( thesol )

In [10]: thesol
Out[10]: [37]

In [11]:
```

Here, on output cell `Out[7]:`, we have the solution of our equation assigned in a *Python* data type container called *set* which does not allow indexing of its elements. For that reason, on input cell `In [9]:` the variable *thesol* is converted to a *Python list* object. This way one can test that the number (37) assigned to the variable *thesol* is in fact the solution of the equation: This is done as follows:

Chapter 3, IPython session 51

```
In [11]: thesol[0] - 17 == 20
Out[11]: True

In [12]:
```

Finding the solution of a full symbolic equation $ax+b = cx+d$ is a follows:

Chapter 3, IPython session 52

```
In [1]: from sympy import symbols, Eq, solveset

In [2]: x, a, b, c, d = symbols('x, a, b, c, d')

In [3]: LHS = a*x + b

In [4]: LHS
Out[4]: a*x + b

In [5]: RHS = c*x + d

In [6]: RHS
Out[6]: c*x + d

In [7]: thesol = list( solveset( Eq(LHS, RHS), x) )

In [8]: thesol
Out[8]: [-(b - d)/(a - c)]

In [9]: a*thesol[0] + b
Out[9]: -a*(b - d)/(a - c) + b

In [10]: c*thesol[0] + d
Out[10]: -c*(b - d)/(a - c) + d

In [11]: a*thesol[0] + b == c*thesol[0] + d
Out[11]: False

In [12]:
```

As shown on output cell `Out[11]:`, the verification of the solution is not straightforward as is with numbers. We still need to use one more *SymPy* function. The *simplify* function (or method) which comes to our rescue to finally show that we have gotten the right answer:

Chapter 3, IPython session 53

```
In [12]: from sympy import simplify

In [13]: simplify(a*thesol[0] + b)
Out[13]: (a*d - b*c)/(a - c)

In [14]: simplify(a*thesol[0] + b) == simplify(c*thesol[0] + d)
Out[14]: True

In [15]:
```

An alternative way to do a proper verification of the solution, we will use a transformed way of the equation. Instead of using $LHS = RHS$ we will work with the transformed (equivalent) equation $LHS - RHS = 0$,

Chapter 3, IPython session 54

```
In [15]: newLHS = LHS - RHS

In [16]: newLHS
Out[16]: a*x + b - c*x - d

In [17]:
```

Now, we need to replace the variable x for the solution we have just found. This is done using the *SymPy* function (or method) *subs* as follows:

Chapter 3, IPython session 55

```
In [17]: newLHS = newLHS.subs(x, thesol[0])

In [18]: newLHS
```

```
Out[18]: -a*(b - d)/(a - c) + b + c*(b - d)/(a - c) - d

In [19]: newLHS == 0
Out[19]: False

In [20]:
```

We still need to use one more *SymPy* function. The *simplify* function (or method) comes to our rescue to finally show that we have gotten the right answer:

Chapter 3, IPython session 56

```
In [20]: newLHS.simplify() == 0
Out[20]: True

In [21]: simplify(newLHS) == 0
Out[21]: True

In [22]:
```

How we can use this general result assigned to the variable *thesol.*? One way is finding the solution of any numerical equation, like the one we worked on page 123, consisting in finding the solution of the equation $x - 17 = 20$, resulting in $x = 37$. Comparing this equation with the algebraic equation $ax + b = c * x + d$, we can see that the former equation is obtained from the later one via setting or replacing $a = 1$, $b = -17$, $c = 0$, and $d = 20$. Accordingly, the setting of this values in the algebraic solution, we should get the solution (37) of our earlier equation. In fact it does:

Chapter 3, IPython session 57

```
In [18]: thesol[0].subs([(a, 1), (b, -17), (c, 0), (d, 20)])
Out[18]: 37

In [19]:
```

You can find this program in the directory named `chapter_03` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the

Preface. In there, find the file named `chap03_prog_14_Sympy_SolvingSymbolicEquation.py`, which you can execute either from a system shell (terminal) or from an *IPython* session.

We encourage you to follow the flow of the code in the function.

3.9.1 The sailors, the coconuts, and the monkeys problem

A popular problem involving whole number reads as follows (a couple of general references about it are cited at the end of the chapter):

> Five sailors survive a shipwreck and swim to an island having (among other things) coconut trees and monkeys. As soon as reaching land, the sailors gathered and piled a good amount of coconuts agreeing to equally divide them the next day. Finished the task of collecting coconuts, exhausted, they all went to sleep.
> When they were all asleep, one man woke up, and decided to take his share. He divided the coconuts into five piles an one coconut was left over that was thrown to the monkeys. After hidden his pile, he put the rest back together and went back to sleep. After a while, a second sailor wakes up and decides to take his share. He divided the coconuts into five piles an one coconut was left over that was thrown to the monkeys. After hidden his pile, he put the rest back together and went back to sleep. The third, fourth and fifth sailors each also wake up at different moments and carry out the same actions as the first two did.
> In the morning, all the sailors wake up, and ignoring what they did during the night, they divided the pile of coconuts left into five piles, but this time the division was perfect (no coconut was left over). The exercise is to figure out how many coconuts were there, in the initial pile that the sailors collected when reaching the island.

We can start working on this problem by trying to set its wording formulation in relevant equations, recognizing first that any result (including the ones obtained at intermediated steps) must be in the domain of whole numbers. Suppose there were a total of (the dividend) N coconuts in the initial pile. The first sailor to divide it by (the divisor) five found a remainder of one coconut. Calling by q_1 the (quotient) number of coconuts taken by the first sailor, and recalling that for a division $dividend = divisor \times quotient + remainder$, we have that:

$$N = 5q_1 + 1 \tag{3.8}$$

$$q_1 = \frac{N-1}{5} \tag{3.9}$$

When this first sailor takes and hides what he think is its part (of q_1 coconuts), four piles containing the same amount of coconuts ($4q_1$) remains to be divided again between five. Consequently, when the second sailor makes the division he will take and hide:

$$4q_1 = 5q_2 + 1 \tag{3.10}$$

$$q_2 = \frac{4q_1 - 1}{5} \tag{3.11}$$

When this second sailor takes and hides what he think is its part (of q_2 coconuts), four piles containing the same amount of coconuts ($4q_2$) remains to be divided again between five. Consequently, when the third sailor makes the division he will take and hide:

$$4q_2 = 5q_3 + 1 \tag{3.12}$$
$$q_3 = \frac{4q_2 - 1}{5} \tag{3.13}$$

When this third sailor takes and hides what he think is its part (of q_3 coconuts), four piles containing the same amount of coconuts ($4q_3$) remains to be divided again between five. Consequently, when the fourth sailor makes the division he will take and hide:

$$4q_3 = 5q_4 + 1 \tag{3.14}$$
$$q_4 = \frac{4q_3 - 1}{5} \tag{3.15}$$

When this fourth sailor takes and hides what he think is its part (of q_4 coconuts), four piles containing the same amount of coconuts ($4q_4$) remains to be divided again between five. Consequently, when the fifth sailor makes the division he will take and hide:

$$4q_4 = 5q_5 + 1 \tag{3.16}$$
$$q_5 = \frac{4q_4 - 1}{5} \tag{3.17}$$

When this fifth sailor takes and hides what he think is its part (of q_5 coconuts), four piles containing the same amount of coconuts ($4q_5$) remains to be divided again between five. Consequently, when all the five sailors are ready to make the division the final division, it happen that (since no remainder is found):

$$4q_5 = 5q_6 + 0 \tag{3.18}$$
$$q_6 = \frac{4q_5}{5} \tag{3.19}$$

In summary, we are left with the following set of equations:

$$q_1 = \frac{N - 1}{5} \tag{3.20}$$
$$q_2 = \frac{4q_1 - 1}{5} \tag{3.21}$$
$$q_3 = \frac{4q_2 - 1}{5} \tag{3.22}$$
$$q_4 = \frac{4q_3 - 1}{5} \tag{3.23}$$
$$q_5 = \frac{4q_4 - 1}{5} \tag{3.24}$$
$$q_6 = \frac{4q_5}{5} \tag{3.25}$$

Examining this set of equations, we see immediately that **if we know** N (the initial amount of coconuts), we can compute q_1, then q_2, following with q_3, continuing with q_4, then q_5, and finally q_6. The only property here is that in each division we must get a whole number.

Since computers are good at performing repetitive tasks, we can think on trying many trial possibilities for N until we get a value for N that lead to subsequent whole number values for the subsequent q_i quotients. A recipe would be as follows:

1. Set $N = 1$.
2. Computes q_1.
3. If q_1 is a whole number, continue computing q_2. Otherwise increase N by one and go to step 2.
4. If q_2 is a whole number, continue computing q_3. Otherwise increase N by one and go to step 2.
5. If q_3 is a whole number, continue computing q_4. Otherwise increase N by one and go to step 2.
6. If q_4 is a whole number, continue computing q_5. Otherwise increase N by one and go to step 2.
7. If q_5 is a whole number, continue computing q_6. Otherwise increase N by one and go to step 2.
8. If q_6 is a whole number, continue printing to screen N, q_1, $\cdots$, q_6. Otherwise increase N by one and go to step 2.
9. Increase N by one and go to step 2.

An implementation of this recipe is shown in Figure 3.15, on page 132. You can find this function in the directory named `chapter_03` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap03_prog_15_SailorsCoconuts.py`.

Executing this program changing the variable *howmanysols* you can see that this exercise does not have a unique solution:

Chapter 3, IPython session 58

```
In [1]: %run chap03_prog_15_SailorsCoconuts.py
Found required 3 values for N = [3121, 18746, 34371]

In [2]:
```

To evaluate the rightness of the obtained results, it occur that this problem happen to have an analytical answer which you could use to asses the evaluation of the numerical computation (see the reference by Gardner, page 110, listed in the reference section, and the Appendix , on page 133 for a derivation of it):

$$N = (1 + 5k)5^5 - 4, \forall \text{ integer } k \geq 0 \tag{3.26}$$

Notice that the obtained results corresponds, respectively, to the values $k = 0$, $k = 1$, and $k = 2$. As an exercise, to further evaluate the rightness of any of the just obtained numerical results, choose one and compute the total amount of coconuts taken by each sailor (after the final step). Do you get N adding them? Should it be? Are any of the intermediated results whole numbers? To carry out this exercise you might one to have the quantities q_1, q_2, $\cdots$, q_6 expressed in terms of the initial amount of coconuts N. An straightforward representation can be found using *SymPy* as follows:

Chapter 3, IPython session 59

```
In [1]: from sympy import symbols

In [2]: N, q1, q2, q3, q4, q5, q6 = symbols('N, q1, q2, q3, q4, q5,
   q6')

In [3]: q1 = (N - 1)/5

In [4]: q1
Out[4]: N/5 - 1/5

In [5]: q2 = (4*q1 - 1)/5

In [6]: q2
Out[6]: 4*N/25 - 9/25

In [7]: q3 = (4*q2 - 1)/5

In [8]: q3
Out[8]: 16*N/125 - 61/125

In [9]: q4 = (4*q3 - 1)/5

In [10]: q4
Out[10]: 64*N/625 - 369/625

In [11]: q5 = (4*q4 - 1)/5

In [12]: q5
Out[12]: 256*N/3125 - 2101/3125
```

```
In [13]: q6 = (4*q5)/5

In [14]: q6
Out[14]: 1024*N/15625 - 8404/15625

In [15]:
```

The preceding results are not very illuminating (see a further discussion on Appendix apend:chap03, on page 133). However, we encourage you to study and compare each one of the given representations. For example, taking the last result (the output cell `Out[14]:`) q_6 is found to have been assigned the value of $\frac{1024}{15625}N - \frac{8404}{15625}$, which you need to compare with the one given by the relation A.16, on page 135.

From this result, we can obtain a representation for N in case q_6 is given to have a unique solution of the problem:

$$q_6 = \frac{1024}{15625}N - \frac{8404}{15625} \tag{3.27}$$

$$q_6 = \frac{1}{15625}(1024N - 8404) \tag{3.28}$$

$$15625q_6 = (1024N - 8404) \tag{3.29}$$

$$15625q_6 + 8404 = 1024N \tag{3.30}$$

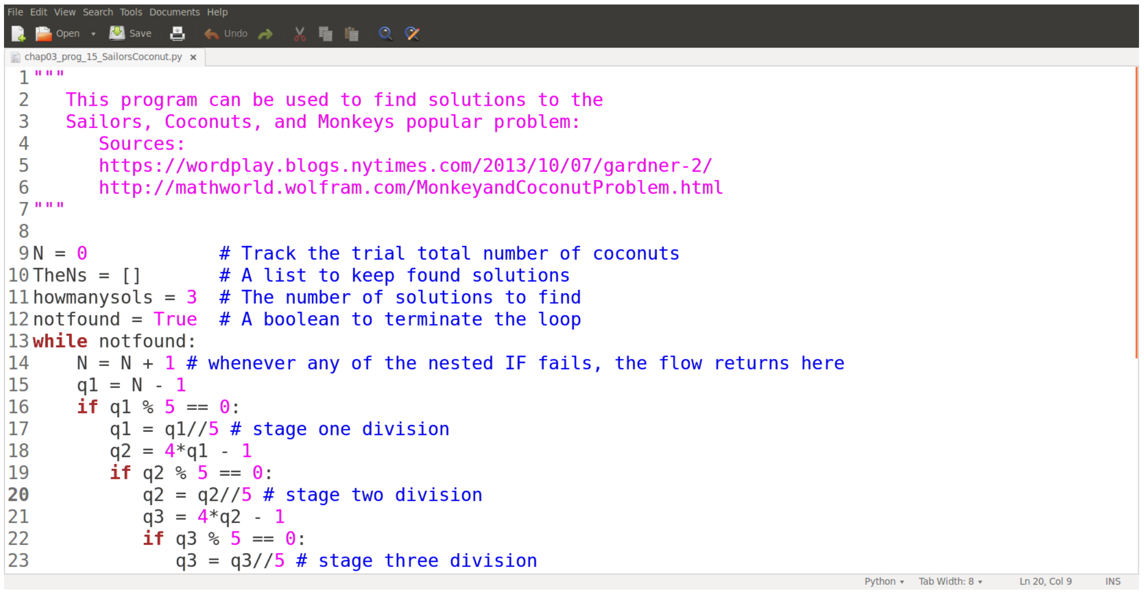
$$1024N = 15625q_6 + 8404 \tag{3.31}$$

$$N = \frac{15625q_6 + 8404}{1024} \tag{3.32}$$

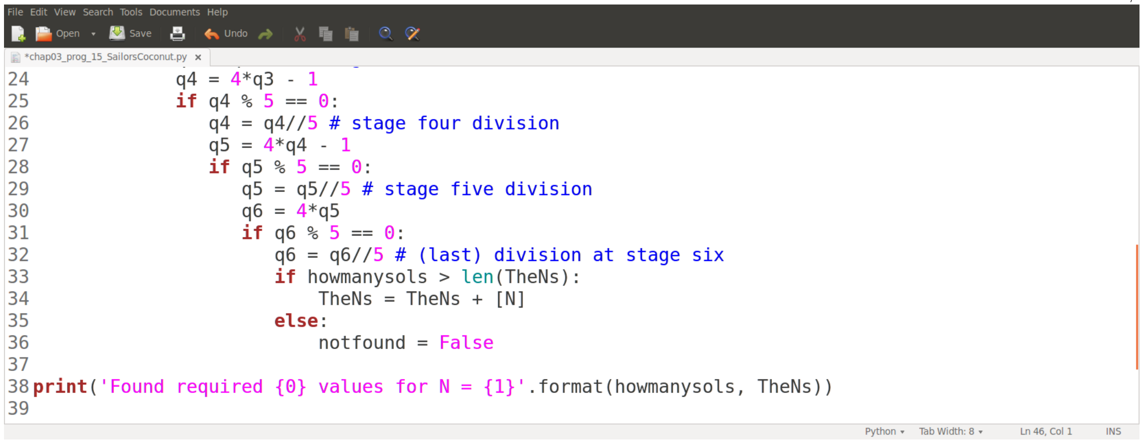
3.10 Chapter Summary

In this chapter you have done great! In order to enhance your understanding of the topics in your Prealgebra course work, you might want to continue writing and executing computations with whole numbers using your own writing *Python* functions, applying the basic *Python* elements you have learned so far: variables, *Python list* objects, *Python* relational operators, and performing repetitive computations via the *Python for* and *while* loops, in addition to some built-in *Python* functions and others coming from some other modules. Later on you can go to a more advance *Python* book to make important improvements on the efficiency of your functions. The most important thing at this moment is that your program give correct answers and you need to start thinking about developing strategies for testing the rightness of the results of your programs (and also of the results obtained by other programs).

At the end of this chapter we worked out the solution of equations using the *SymPy* module and also how to verify that the solution found satisfy the given equation. Keep in mind that the general procedure is also applied beyond whole numbers.



```
"""
   This program can be used to find solutions to the
   Sailors, Coconuts, and Monkeys popular problem:
      Sources:
      https://wordplay.blogs.nytimes.com/2013/10/07/gardner-2/
      http://mathworld.wolfram.com/MonkeyandCoconutProblem.html
"""

N = 0            # Track the trial total number of coconuts
TheNs = []       # A list to keep found solutions
howmanysols = 3  # The number of solutions to find
notfound = True  # A boolean to terminate the loop
while notfound:
    N = N + 1 # whenever any of the nested IF fails, the flow returns here
    q1 = N - 1
    if q1 % 5 == 0:
       q1 = q1//5 # stage one division
       q2 = 4*q1 - 1
       if q2 % 5 == 0:
          q2 = q2//5 # stage two division
          q3 = 4*q2 - 1
          if q3 % 5 == 0:
             q3 = q3//5 # stage three division
```



```
             q4 = 4*q3 - 1
             if q4 % 5 == 0:
                q4 = q4//5 # stage four division
                q5 = 4*q4 - 1
                if q5 % 5 == 0:
                   q5 = q5//5 # stage five division
                   q6 = 4*q5
                   if q6 % 5 == 0:
                      q6 = q6//5 # (last) division at stage six
                      if howmanysols > len(TheNs):
                          TheNs = TheNs + [N]
                      else:
                          notfound = False

print('Found required {0} values for N = {1}'.format(howmanysols, TheNs))
```

Figure 3.15: Program to find solutions to the sailors, coconuts, and monkeys problem.

In the next chapter we will learn about some of the formalities of the *Python print* function that we have been using without much explanation; we will also study some basic elements of reading from and writing to the screen and to files, including the writing of messages indicating the user of your programs that something is not right in the given input using *Python* exception handling methodology.

Appendix of Chapter 3

A.1 Algebraic solution to the sailors, the coconuts, and the monkeys problem

In approaching a way to find a solution of this problem (posed on page 127), the first thing that comes to mind, after a careful reading of its statement, is that any solution must be given in terms of the prescribed initial amount of coconuts (N) and that it must be in the domain of whole numbers (including any intermediated result). As a second step, perhaps, after recognizing the previous condition, your thought processes were directed momentarily to a quick tour recalling the properties of the whole numbers that you have learned so far in your Prealgebra course work. Then, as a third step, you might have decided to start rephrasing the wording of the problem in term of the language of equations 3.20--3.25 (on page 128) according to the given instructions in the description of the problem. Each one of the preceding steps are part of *designing* or deciding an approach to attempt a solution of the problem, that in turns are the guiding principles to propose the numerical algorithm of page 129 to find a numerical answer to the problem, that (after *implementing* in *Python* the algorithm) let us to find that it has a not unique solution, as shown in page 129. The last step is to evaluate the correctness of the found answers, for which you were let the exercise 3.15, on page 143.

In passing regarding the no uniqueness of the solution to this problem, you will find in the literature wording alternatives that imposes extra conditions to the problem so that the respective solution is unique.

We will not be explicitly concerned with those alternatives problem here. Nevertheless, to find solutions for those situations, it is better to start thinking on the possibility of finding a better analytical representation of the problem in terms of equations, improving the obtained straightforward (direct) representation that we have studied in section 3.9.1 (page 127) and summarized in the set of equations 3.20--3.25 (on page 128) and further complemented with the *SymPy* analysis of page 130.

Thinking about the problem, you might find that this preceding analysis is unsatisfactory because we don't see any pattern in the equations that captures the repetitive involved operations (until the final step) taking into account the amount of coconuts left at any intermediated step and the initial amount of coconuts N (we only have developed the straightforward setup of the problem in term of the q_i amount of coconuts taken by each sailor at different times during the night). Why is this important, you might be wondering. The answer is that the given quantity in the problem is the initial amount of coconut N, not the amount of coconuts taken by the sailors. That they must be whole numbers are conditions that needs to be fulfilled in order to considering any found solution satisfactory.

After this reflection, your thoughts processes might have turn now in finding such suitable representation. A way of doing it starts computing the amount of coconuts N_1 left after the first sailor divides the initial pile of coconuts N by 5, which results in $N_1 = 4q_1 = 4(\frac{N-1}{5})$. Furthermore, after a while of involvement based on our experience from the previous analysis of section 3.9.1, on page chap03:MonkeyCoconuts, we will find that this quantity might better be rewritten in the form:

$$N_1 = \frac{4}{5}(N+4) - 4 \tag{A.1}$$

$$N_1 + 4 = \frac{4}{5}(N+4) \tag{A.2}$$

$$q_1 = \frac{N_1}{4} = \frac{1}{4}\left(\frac{4}{5}\right)(N+4) - 1 \tag{A.3}$$

As a matter of fact, our intuition is rewarded when in the next step the amount of coconuts N_2 left in the pile after the second division is given by $N_2 = 4q_2 = \frac{4}{5}(4q_1 - 1) = \frac{4}{5}(N_1 - 1) = \frac{4}{5}(N_1 + 4) - 4$, from which one can get:

$$N_2 = \frac{4}{5}(N_1+4) - 4 = \frac{4}{5}\left(\frac{4}{5}(N+4)\right) - 4 = \left(\frac{4}{5}\right)^2 (N+4) - 4 \tag{A.4}$$

$$N_2 + 4 = \left(\frac{4}{5}\right)^2 (N+4) \tag{A.5}$$

$$q_2 = \frac{N_2}{4} = \frac{1}{4}\left(\frac{4}{5}\right)^2 (N+4) - 1 \tag{A.6}$$

Noticing the pattern that is being formed, we continue computing the next N_3 amount of coconuts left after the third division $N_3 = 4q_3 = \frac{4}{5}(4q_2 - 1) = \frac{4}{5}(N_2 - 1) = \frac{4}{5}(N_2 + 4) - 4$, from which one can get:

$$N_3 = \frac{4}{5}(N_2+4) - 4 = \frac{4}{5}\left(\left(\frac{4}{5}\right)^2 (N+4)\right) - 4 = \left(\frac{4}{5}\right)^3 (N+4) - 4 \tag{A.7}$$

$$N_3 + 4 = \left(\frac{4}{5}\right)^3 (N+4) \tag{A.8}$$

$$q_3 = \frac{N_3}{4} = \frac{1}{4}\left(\frac{4}{5}\right)^3 (N+4) - 1 \tag{A.9}$$

Follows the next N_4 amount of coconuts left after the fourth division $N_4 = 4q_4 = \frac{4}{5}(4q_3 - 1) = \frac{4}{5}(N_3 - 1) = \frac{4}{5}(N_3 + 4) - 4$, from which one can get:

$$N_4 = \frac{4}{5}(N_3 + 4) - 4 = \frac{4}{5}\left(\left(\frac{4}{5}\right)^3 (N+4)\right) - 4 = \left(\frac{4}{5}\right)^4 (N+4) - 4 \tag{A.10}$$

$$N_4 + 4 = \left(\frac{4}{5}\right)^4 (N+4) \tag{A.11}$$

$$q_4 = \frac{N_4}{4} = \frac{1}{4}\left(\frac{4}{5}\right)^4 (N+4) - 1 \tag{A.12}$$

Now the next N_5 amount of coconuts left after the fifth division $N_5 = 4q_5 = \frac{4}{5}(4q_4 - 1) = \frac{4}{5}(N_4 - 1) = \frac{4}{5}(N_4 + 4) - 4$, from which one can get:

$$N_5 = \frac{4}{5}(N_4 + 4) - 4 = \frac{4}{5}\left(\left(\frac{4}{5}\right)^4 (N+4)\right) - 4 = \left(\frac{4}{5}\right)^5 (N+4) - 4 \tag{A.13}$$

$$N_5 + 4 = \left(\frac{4}{5}\right)^5 (N+4) \tag{A.14}$$

$$q_5 = \frac{N_4}{4} = \frac{1}{4}\left(\frac{4}{5}\right)^5 (N+4) - 1 \tag{A.15}$$

Finally, dividing the amount of coconuts left after the fifth division, each sailor gets the extra amount of coconuts $N_6 = q_6 = \frac{1}{5}(4q_5) = \frac{4}{5}q_5$, from which one can get:

$$N_6 = q_6 = \frac{4}{5}q_5 = \frac{4}{5}\left(\frac{1}{4}\left(\frac{4}{5}\right)^5 (N+4) - 1\right) = \frac{1}{4}\left(\frac{4}{5}\right)^6 (N+4) - \frac{4}{5} \tag{A.16}$$

The condition here breaks the last recursive relationship we were obtaining for the number of coconuts at each stage!.

From this result, we can obtain a representation for N in case N_6 is known:

$$N_6 + \frac{4}{5} = \frac{1}{4}\left(\frac{4}{5}\right)^6 (N+4) \tag{A.17}$$

$$4\left(N_6 + \frac{4}{5}\right)\left(\frac{5}{4}\right)^6 = N + 4 \tag{A.18}$$

$$N = 4\left(N_6 + \frac{4}{5}\right)\left(\frac{5}{4}\right)^6 - 4 \tag{A.19}$$

$$N = \frac{4}{5}(5N_6 + 4)\left(\frac{5}{4}\right)^6 - 4 = (5N_6 + 4)\left(\frac{5}{4}\right)^5 - 4 \tag{A.20}$$

Now, from this equation, based on the condition that the result must be a whole number, a moment of thought tells us that to find the minimal amount of initial coconuts we need to impose that $5N_6 + 4 = 4^5$. This leads to the result $N = 5^5 - 4 = 3121$, which is the first value found with our program of page 132, and shown on page 129. To find more solutions, some extra work is required.

Furthermore, pursuing forward an analytical solution, based on our preceding result, we can start imposing the general condition that $5N_6 + 4 = \alpha 4^5$, with α any natural number (recall that the natural numbers excludes the zero from the whole numbers). This guarantees that $N = \alpha 5^5 - 4$ is also a natural number, as required by the setup of the problem. Notice that we have excluded zero as a possible solution to the problem. It might be considered a trivial (non interesting) solution of it.

To proceed with the task of finding an analytical solution satisfying all the conditions required by the problem, your thought process might be guided o ask whether assuming the hypothetical condition also lead to natural numbers as required for the intermediate steps? To find out that, we start finding the value for N_6:

$$5N_6 + 4 = \alpha 4^5 \tag{A.21}$$
$$5N_6 = \alpha 4^5 - 4 \tag{A.22}$$
$$N_6 = \frac{\alpha 4^5 - 4}{5} \tag{A.23}$$

A moment of thought tells us that for Equation A.23 be a natural number, we can choose $\alpha = 1 + 5k$ (with k any whole number) so that we can rewrite that equation in the form:

$$N_6 = \frac{(1 + 5k)4^5 - 4}{5} = \frac{5k4^5 + 4^5 - 4}{5} \tag{A.24}$$
$$N_6 = 4^5 k + \frac{4^5 - 4}{5} \tag{A.25}$$

Since $\frac{4^5-4}{5} = \frac{1020}{5} = 204$ is a natural number, it follows that N_6 will be a natural number for any (whole number) value taken by k. Via similar steps, substituting (or using) $N = (1 + 5k)5^5 - 4$ on each one of the respective equations for N_1, N_2, $\cdots$, N_5 we see that natural numbers are obtained for those quantities, which proves that Equation 3.26, on page 130 is the exact (general) solution for our problem. In carrying out such computations you might find useful the summary of intermediated results:

$$N = (1+5k)5^5 - 4 \tag{A.26}$$
$$\frac{N+4}{5^5} = 1+5k \tag{A.27}$$
$$\frac{N+4}{5^4} = (1+5k)5 \tag{A.28}$$
$$\frac{N+4}{5^3} = (1+5k)5^2 \tag{A.29}$$
$$\frac{N+4}{5^2} = (1+5k)5^3 \tag{A.30}$$
$$\frac{N+4}{5} = (1+5k)5^4 \tag{A.31}$$
$$N+4 = (1+5k)5^5 \tag{A.32}$$

Reaching this point, it is a good idea to revise your work, but this time trying to make explicit in your course of reasoning the stages of *designing* (the solution of the problem), *implementing* (the designed course of action to find a solution), and *testing* (the found solution of the problem). These stages can actually be converted into more specific directions (but not too specific to be confusing or distractive) providing sufficient guidance in the development process of problem-solving. Our research in this regards let us to propose (and used here) what we have argued elsewhere (see the reference section) to be a *dynamic problem solving strategy*, not to be confused with modes of reasoning, constructed from the following steps:

1. Understand and describe the problem.
2. Provide a qualitative description of the problem.
3. Plan a solution.
4. Carry out the plan.
5. Verify the internal consistency and coherence of the equations used and the applied procedures.
6. Check and evaluate the obtained solution.

In using this strategy, don't think of a rigidly sequential application the steps to the problem at hand. Instead you have to consider each one of them at any level of the advancing process to find a solution of the problem. Keep in mind that any inflexible problem solving strategy can only be of limited value. Moreover, a key important aspect of applying this *dynamic problem solving strategy* is your ability to ask questions.

By asking questions while solving a problem one becomes engaged in a process of self-explaining components of the underlying procedure being applied to solve the problem. Asking questions also helps in the detection of "comprehension failures" about the problem and the procedure being applied to solve it, and to take action to overcome them such comprehension failures.

Examples of questions to be asked constantly, during the process of solving a problem, include: how this knew knowledge is related to what I already know.? In which context I have seen this problem before.? In which context could I use this piece of knowledge.? How these seemingly disparate discrete pieces of knowledge be functionally and causally related.? Does the principles to be applied can be used in this situation.? Is this approach the right one.? How can I be certain of it.? Are we sure one can do this or that.?

More importantly, by know you will be aware that quantitative reasoning and problem solving skills are a desirable outcome from the process of teaching and learning of the sciences. In this regards, a well structured problem solving strategy, understood as a dynamical process, offers a feasible way to learn and analyze subjects quantitatively and conceptually. It also helps the practitioner to reach the state of an "adaptive expert", highly skillful on innovation and efficiency, a desired outcome from the perspective of a "Preparation for Future Learning" approach of the process of teaching and learning effectively leading to the formation of individuals who are highly efficient in applying (transferring) what they know to tackle new situations and are also extremely capable of innovation in the sense of being able to inhibit inadequate blocking "off the top of the head processes" or "to break free of well-learned routines" so that they can move to new learning episodes by finding, perhaps ingenious, ways to approach first time met situations.

We have no doubt that as you get into consciously practicing the basic steps of writing programs (namely *designing*, *implementing*, and *testing* algorithms) you will be approaching the stage of becoming an skillful on innovation and efficiency "adaptive expert."

Exercises of Chapter 3

Exercise 3.1 *In an* IPython *console execute the following lines of code:*

```
def myfuncIntsRandom(howmany, minVal=0, maxVal=100):
    import random
    temp = []
    for i in range(howmany):
      temp = temp + [random.randrange(minVal,maxVal+1)]
    return temp

theVals = myfuncIntsRandom(20)
print(theVals)
theMaxVal = theVals[0]

k = 0
while k < len(theVals)-1:
    k = k + 1
    if (theMaxVal < theVals[k]):
       theMaxVal=theVals[k];
print('The largest value in the set is = {0}'.format(theMaxVal))

print('max(theVals) - theMaxVal = {0}'.format(max(theVals) - theMaxVal))
```

Exercise 3.2 *In an* IPython *console execute the following lines of code:*

```
TheValues = [2, 5, 1, 0, 7, 5, 3, 8, 50, 9, 11, 4]

print("\n Data at the beginning: ", TheValues)

#----
thesize = len(TheValues)
done = True
i = 0
while done:
     done = False
```

```
    tempsize = range( thesize - i - 1 )
    for j in tempsize:
       if ( TheValues[j] > TheValues[j+1]):
          temp = TheValues[j];
          TheValues[j] = TheValues[j+1];
          TheValues[j+1] = temp;
          done = True
    i= i + 1
#----
print("\n Data at the end: ", TheValues)
```

After executing the code, what do you think this program is doing.? Could you follow the flow of the code usin paper and pencil using a small sample of values, like [1, 5, 2].? How could you improve the ***testing*** *of the code?*

Exercise 3.3 *The lines of code on page 92, could be replaced by the following ones:*

```
k = MostRepeatedVals[0]
MultiMode = 0
m = 0
mm = len(MostRepeatedVals)
while ( m < mm ):
   j = MostRepeatedVals[m]
   if j != k:
     MultiMode = MultiMode + 1 # hay otro valor
   m = m + 1
```

Could you see what is the difference between them.? Which one would you choose.? Could you think of a better option.? Could you change the code to use a for loop *instead of the* while loop.*?*

Exercise 3.4 *Execute the* mode *program on page 95, using the following data sets:*

```

TheValues = [17, 14, 14, 17, 16, 15, 16, 17, 14, 15, 13, 18, 13, 17,
   17, 16]

TheValues = [17, 14, 16, 15, 13, 18]

TheValues = [17, 14, 14, 16, 15, 16, 14, 15, 13, 18, 13]

TheValues = [17, -14, -14, 16, 15, 16, -14, 15, 13, 18, 13, 16]
TheValues = [17, -14.55, -14.55, 16.1, 15, 16.1, -14.55, 15, 13, 18,
   13, 16.1]
```

Exercise 3.5 *The following function can be used to test if a given number n is or not prime (follow its flow and compare it with the sieve method of section 3.8.2.1, starting on page 109):*

```
def myfuncISprime(n):
   import numpy as np
   if n < 2:
      return False
   if n == 2:
      return True
   i = int( np.sqrt(n) )
   while (i != 1):
       if (n % i == 0):
           return False
       i = i - 1
   return True
```

Write a program to test this function with some whole numbers (after trying a few number, you might one to review section 3.6, starting on page 98). This function can also be used as a simple prime number generator, from one up to the given whole number n. Can you write a code for doing it.?

Exercise 3.6 SymPy *offer an efficient implementation to test primality It works as follows:*

```
In [6]: from sympy.ntheory import isprime

In [7]: isprime( 350500000000555500 )
Out[7]: False

In [8]:
```

How could you verify the rightness of the obtained result.?

By the way, read about this function executing in an IPython *cell the instruction* ***isprime?*** *or* ***isprime??****.*

Exercise 3.7 *Write a program to verify that the primality of the numbers generated via the function of section 3.8.2.1, on page 111 (showing our implementation of the* sieve method *for* *prime number generation). For this you might want to review section 3.8.2, starting on page 108 and section 3.6, starting on page 98.*

Exercise 3.8 *Using the function that finds if a given whole number is abundant or not (shown in page 113), write a program that shows that for* $1 \leq n \leq 1000$ *there are* 246 *abundant whole numbers. Which one is the smallest abundant whole number.?*

Exercise 3.9 *Using the function that finds if a given whole number is perfect or not (shown in page 114), can you write a program that shows that for* $1 \leq n \leq 1000$ *there are only three whole numbers.? Which one are they.?*

Exercise 3.10 *The* Euclidean algorithm *for finding the* greatest common divisor *(GCD) of two whole numbers* x *and* y *goes as follows: if* $y = 0$*, the GCD is* x*, and vice versa. Otherwise, get the remainder* (r) *of dividing* x *by* y *(here we assume* $x \geq y$*). If* $r \neq 0$*, assign* y *to* x *and* r *to* y*, and repeat th procedure until a remainder of zero is obtained. The last non-zero remainder is the GCD between* x *and* y*. Your task is to write this algorithm as a function and test its functionality.*

Exercise 3.11 *From your Prealgebra course work, you might know that a way for verifying that a given GCD of a set of values is the right one consist in finding the quotient of dividing each one of the given values by the obtained GCD and then verifying that the only common factor among the results is the number one. In other words, the GCD of the quotients must be one. Could you write a program to verify the output of the GCD function shown on page 116.?*

Exercise 3.12 SymPy *offer an efficient implementation to find out the greatest common factor of whole numbers. It works as follows:*

```
In [1]: from sympy.core.numbers import igcd

In [2]: thegcd = igcd(5, 10, 15, 125)

In [3]: thegcd
Out[3]: 5

In [4]:
```

How can you verify that the obtained answer is the right one.? Remember to read about this function executing in an IPython *cell the instruction* **`igcd?`** *or* **`igcd??`**. *Are you surprised.?*

Exercise 3.13 *Can you think of a way to verify the rightness of the answer returned by the function computing the prime factorization of a number. shown on Figure 3.14, page 120.? Write down a program implementing your idea.*

Exercise 3.14 SymPy *offer an efficient implementation to find out the prime factorization of whole numbers. It works as follows:*

```
In [1]: from sympy.ntheory import factorint

In [2]: thefactorization = factorint(350500000000555500).items()

In [3]: thefactorization
Out[3]: dict_items([(2, 2), (3, 1), (5, 3), (3181, 1), (73456984177,
   1)])

In [4]: thefactorization = sorted( thefactorization )

In [5]: thefactorization
Out[5]: [(2, 2), (3, 1), (5, 3), (3181, 1), (73456984177, 1)]

In [6]:
```

Could you apply your program of the previous problem to check the prime factorization reported by this SymPy *function.? Remember to read about this function executing in an* IPython *cell the instruction* ***`factorint?`*** *or* ***`factorint??`****.*

By the way, in case you apply our function on page 120 in the large number of input cell `In [2]:` *you might need to hit* ***`CTRL-C`*** *to stop it as it will take a long time to print out the result.*

Exercise 3.15 *Verify the obtained numerical results for the sailors, coconuts, and monkeys problems shown in page 129, corresponds, respectively, to the values $k = 0$, $k = 1$, and $k = 2$ of the Equation 3.26, on page 130. To further evaluate the rightness of the obtained numerical results, for all of them compute the total amount of coconuts taken by each sailor (after the final step). Do you get N adding them? Should it be? Are any of the intermediated results whole numbers? Should they be? Why?*

References of Chapter 3

Books and/or Articles

- **Marecek, L. and Smith, M. A.** (2017). Prealgebra, Rice University, OpenStax `https://openstax.org`.
  Book available for free at: `http://cnx.org/content/col11756/1.9`

- **Gardner, M.** (1987). The 2nd Scientific American book of mathematical puzzles & diversions. The University of Chicago Press.

- **Problem-solving strategies**:
  Polya, G. (1988) How to solve it. A new aspect of mathematical method. Expanded edition, Princeton University Press. **Heller, J. I. and Reif, F.** (1984) Prescribing Effective Human Problem-Solving Processes: Problem Description in Physics. *Cognition and Instruction*, **1** (2), 177--216. **Rojas, S** (2012) Enhancing the process of teaching and learning physics via *dynamic problem solving strategies*: a proposal. *Revista Mexicana de Física E*, **58** (1), 7--17 (Freely available at `http://rmf.fciencias.unam.mx/pdf/rmf-e/56/1/56_1_022.pdf`).
  Rojas, S (2010) On the teaching and learning of physics problem solving. *Revista Mexicana de Física*, **56** (1), 22--28 (Freely available at `http://rmf.fciencias.unam.mx/pdf/rmf-e/56/1/56_1_022.pdf`).

- **About equations**:
  Bernstein, M. A. and Friedman, W. A. (2009). Thinking about equations: A practical guide for developing mathematical intuition in the Physical Sciences and Engineering. John Wiley & Sons, Inc.
  Cochrane, R. (2016). The Secret Life of Equations: The 50 Greatest Equations and How They Work. Octopus Publishing Group, Ltd.
  Farmelo, G. (2002) It must be beautiful. Great Equations of Modern Science. Granta Books. **Hewitt, P. G.** (2011) Equations as guides to thinking and problem solving. *The physics teacher*, **49** (), 264. **Rojas, S** (2008) On the need to enhance physical insight via mathematical reasoning. *Revista Mexicana de Física E*, **54** (1), 75--80 (Freely available at `https://rmf.smf.mx/pdf/rmf-e/54/1/54_1_75.pdf`).

- **Logistic map**:
  May, R. M. (1976) Simple mathematical models with very complicated dynamics, *Na-*

ture, **261** (5560) 459--467.
Devaney, R. L. (1992) A first course in chaotic dynamical systems. Theory and experiments. Addison-Wesley Publishing Company, INC.

References on the WEB

- **Prime numbers**:
  http://primes.utm.edu/
- **The Monkey and the coconuts problem**:
  http://mathworld.wolfram.com/MonkeyandCoconutProblem.html
- **MIT OCW Python Course**:
  http://web.mit.edu/600/www
  1. Eric Grimson, and John Guttag. 6.0001 Introduction to Computer Science and Programming in Python. Fall 2008. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA
     http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00-introduction-to-computer-science-and-programming-fall-2008/
  2. Ana Bell, Eric Grimson, and John Guttag. 6.0001 Introduction to Computer Science and Programming in Python. Fall 2008. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA
     https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/index.htm
- **Logistic map**:
  https://en.wikipedia.org/wiki/Logistic_map

4

Reading and writing (input/output) in *Python*

"The best way to predict the future is to write it."
Folk Wisdom

4.1 Introductory remarks

In the preceding chapters we started to write small *Python* programs using the *gedit* text editor which can then be executed using the *IPython* console or the system (terminal) shell. We also made use of the built-in *Python print* function without explaining its operational structure (you were suppose to exercise an act of faith and type it as given).

In this chapter we will introduce basic operational functionalities of *Python* for reading data entered via the keyboard or from text files. We will use the writing of data to files in text format (we let you to explore in your own other formats). Complementing these actions, we will study the controlling of input data via assertion statements capturing exceptions and unexpected type of data. With this knowledge you can start writing short *Python* games to be played in a terminal of system shell. We will do that with the *guess a two digit* game analyzed in section 1, on page 47.

Accordingly, after finishing this chapter, you'll be equipped with a basic set of *Python* tools that will allow you to write fully functional programs that will allow you o take advantage of the computational power of your computer to fully explore the topics of your prealgebra course work, limited for now to whole (and integers) numbers until studying in the next chapter the representation of real numbers in *Python*. Keep in mind that a major goal of your investment reading this book is (as an independent learner) to enhance your skills in applying what you have learned to tackle new first (unfamiliar) met situations. The successful acquisition of such desired outcome requires your engagement in an effective learning involvement of constantly applying the programming stages of *designing* (actions), *implementing* (the actions in some order), and *assessing* (the performance of such actions) to any other real world situations.

4.2 *Python print* function

So far we have been using the *print* function in the simple form, using simple or double quotes:

Chapter 4, IPython session 1

```
In [1]: printout0 = 198

In [2]: printout1 = 'is a string'

In [3]: print('1st output = {0} ; 2nd output {1}'.format(printout0,
   printout1))
1st output = 198 ; 2nd output is a string

In [4]: print("1st output = {0}; 2nd output {1}".format(printout0,
   printout1))
1st output = 198; 2nd output is a string

In [5]: print('2nd output {1} ; 1st output = {0}'.format(printout0,
   printout1))
2nd output = is a string ; 1st output = 198

In [6]: print('This {1}, then a number {0}. This {1}
   ...'.format(printout0, printout1))
This is a string, then a number 198. This is a string ...

In [7]:
```

In these printing instructions we have let the *Python* interpreter decide how to print to screen each required output (i.e., we are not passing instructions about how many digits to print and how to print them). Just notice the correspondence between the numbers enclosed in curly braces $\{\cdots\}$ and the parameters inside the parenthesis of the *format* instruction. The first parameter *printout0* has been assigned what is going to be printed in place of $\{0\}$, while the second parameter *printout1* has been assigned what is going to be printed in place of $\{1\}$. And that is the sequence for for printing to screen the remainder parameters, if there were more. In case there are no parameters to be passed to the *format* instruction, the later can be omitted from the string to be printed by the *print* function.

As shown in the preceding *IPython* session (input cells `In [5]:` and `In [6]:`), we should point out that while the sequence of parameters inside the parenthesis of the *format* instruction are numbered in the sequence they appear (from left to right starting from zero), they could appear in any order (and as many times as required) enclosed by the curly braces inside the body (*string*) statement to be printed on the computer screen, surrounded by single or double quotes to the left of the *format* instruction.

The set of characters enclosed by single or double quotes are called *Python string* objects. Among other properties, similar to a *Python list*, *string* objects support *indexing* and concatenation via the plus (+) sign. There is also available the *Python* built-in function *str* that allows the conversion of other *Python* objects to a *string* type object. The following *IPython* session illustrates a few aspects of working with *string* objects:

Chapter 4, IPython session 2

```
In [10]: a = 'xyz + / \ @ # 123 1 a d " % *'

In [11]: a
Out[11]: 'xyz + / \\ @ # 123 1 a d " % *'

In [12]: print(a)
xyz + / \ @ # 123 1 a d " % *

In [13]: a[0]
Out[13]: 'x'

In [14]: a[4]
Out[14]: '+'

In [15]: a[-1]
Out[15]: '*'

In [16]: type(a)
Out[16]: str

In [17]: type(a[6])
Out[17]: str

In [18]: a[6] + a[8]
Out[18]: '/\\'

In [19]: a[9]
Out[19]: ' '

In [20]: a[8]
Out[20]: '\\'

In [21]: a[8] = 'this'
---------------------------------------------------------------------------
TypeError                               Traceback (most recent call last)
<ipython-input-21-9faac9dbb34f> in <module>()
```

```
----> 1 a[8] = 'this'

TypeError: 'str' object does not support item assignment

In [22]: a[0] + 1
---------------------------------------------------------------------------
TypeError                         Traceback (most recent call last)
<ipython-input-22-8830f3e33e89> in <module>()
----> 1 a[0] + 1

TypeError: must be str, not int

In [23]: a[0] + str(1)
Out[23]: 'x1'

In [24]: a[14]
Out[24]: '1'

In [25]: a[14] + str(1)
Out[25]: '11'

In [26]: '\n'
Out[26]: '\n'

In [27]: print('\n')



In [28]: print('This is a \t tab')
This is a  tab

In [29]: print("It's the printing of single quote")
It's the printing of single quote

In [30]: print('It\'s the printing of single quote')
It's the printing of single quote

In [31]: print('Printing double quotes ("... ")')
Printing double quotes ("... ")

In [32]: print("Printing single ' and double quotes (\"... \")")
Printing single ' and double quotes ("... ")

In [33]:
```

The backslash (`'\'`) string symbol has a special meaning inside the *print* function (recall that by itself, not enclosed in quotes, the backslash (\) symbol indicates line continuation). For instance, on input cell `In [12]:` it is interpreted as a way of escaping (printing) itself, as a double backslash string (`'\\'`) prints only one backslash. Two special string characters are commonly used in strings: the newline character (`'\n'`), which is used to generate a new line, and the tab character (`'\t'`), which generates a tab. Other characters are available. We will introduce the necessary ones along the text, but we let you as a homework to find them.

By default, the *print* function applies the new line character after finishing its printing to screen (which is also the default option). There are situations on which we want to continue printing in the same line using a later *print* instruction. We can overwrite the default behavior of the *print* by adding, comma separated, `end=''` after (if there is one) the format instruction as follows:

Chapter 4, IPython session 3

```
In [22]: print('The number {0} followed by \\'.format(8));print('\\')
The number 8 followed by \
\

In [23]: print('The number {0} followed by
   \\'.format(8),end='');print('\\')
The number 8 followed by \\

In [24]: print('This is a printout ', end='');print('continued by
   another')
This is a printout continued by another

In [25]:
```

Another alternative to write the printing output is to assign repetitive strings to variables and then use them as necessary:

Chapter 4, IPython session 4

```
In [26]: thestring = 'This is a string repeated in many printing
   instructions'

In [27]: print(thestring)
```

```
This is a string repeated in many printing instructions

In [28]:
```

This basic usage of the print instruction allows you to write necessary instructions to the users of your code. We will use them to print instructions when reading data that needs to be entered from the keyboard or when writing data to files. To further understand the functionality of the *print* function you are welcome to read the documentation [`https://docs.python.org/3/library/functions.html#print`].

4.3 *Python input* function and *try--except* statement

Python provides the built-in function *input* to capture data typed by the user. The syntax of this function is as follows:

```
input(string_variable)
```

It will wait until the user enter or type whatever is being requested and hits the **RETURN** key. It can be used to pause the execution of a program. The following *IPython* session illustrate how the function works:

Chapter 4, IPython session 5

```
In [1]: x = input('Enter something and hit return: ')
Enter something and hit return: wert

In [2]: print(x)
wert

In [3]: type(x)
Out[3]: str

In [4]: x = input('Enter something and hit return: ')
Enter something and hit return: 234

In [5]: print(x)
234

In [6]: type(x)
Out[6]: str
```

```
In [7]:
```

Notice that the default output from the *input* function is of type *str* (see the output cells `Out[3]:` and `Out[6]:`), representing an *string* object discussed in the previous section. *Python* has the built-in functions *float* and *int* to convert numerical values represented as strings in actual numbers.

This fact is further illustrated by the function shown in Figure 4.1, on page 153, which was written for reading a whole number entered interactively by a user using the computer keyboard executing that function. Other examples will be shown in subsequent sections, but its usage in this book will not go beyond what is shown in the aforementioned Figure.

The functionality of using the *input* function can be enriched by the programmer (exercising its good programming skills) via the *try--except* instruction to ensure that the received data is of the type required by the *Python* program being executed. This instruction also helps to capture and handle exceptions that otherwise could make the program crash without the user knowing the reason, with eventually some precious time lost trying to find the bug.

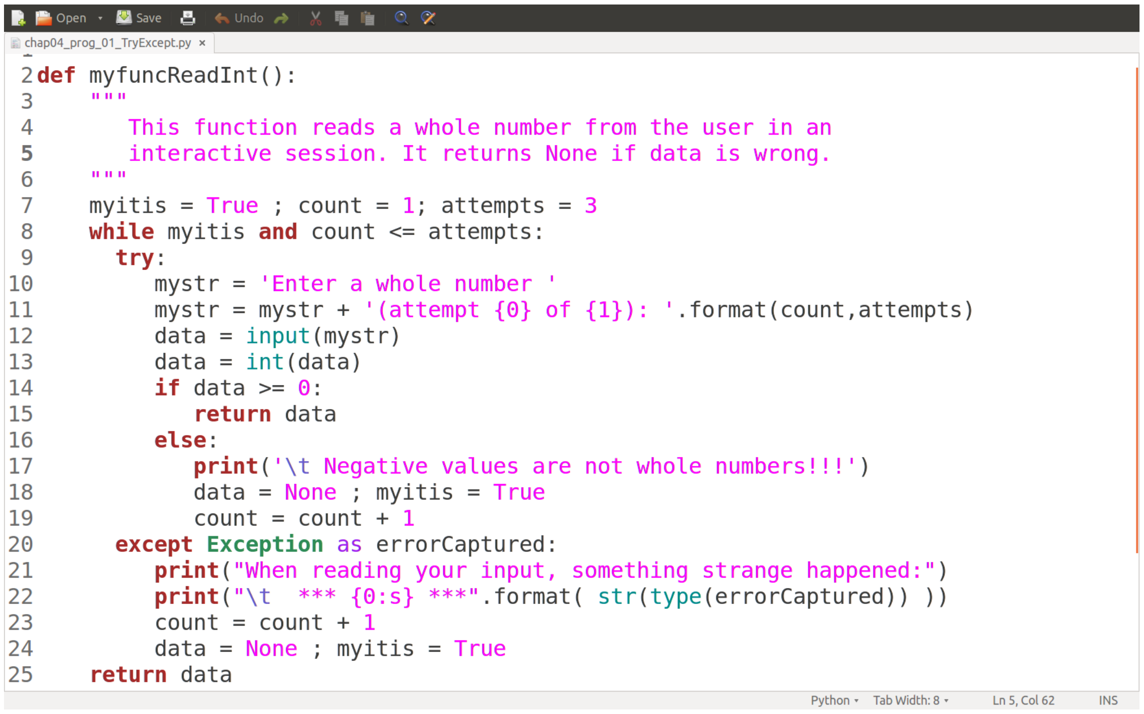
Via the references listed at the end of this chapter, on page 168, you are encourage to explore the many capabilities allowed by the statement *try--except* to handle exceptions. We will limit our use of exception handling to print a message telling the user that something strange happened with the given input and also printing what was the mistake as captured by *Python*.

```
try:
    Instructions to be execute
except Exception as myerrorCaptured:
    What to do if an error happen in the body of try
```

Used in this way, the flow of the code enters the body of the *try* statement and if something goes wrong considered as an *exception* in *Python* the flow of the code will go to execute the body of the *except Exception as*, assigning to the variable *myerrorCaptured* whatever *exception* had happened. Consequently, the programmer can apply a defensive programming step to tell why and where the code is falling.

An example of using the *try--except* instruction is shown in Figure 4.1, on page 153. You can find this function in the directory named `chapter_04` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap04_prog_01_TryExcept.py`.

As mentioned, the function in Figure 4.1 is used to read a whole number entered by the user of the function via the keyboard. The function assumes that the user could make mistakes while typing the required output and sets (via the variable *attempts*) three options to do that. A *while* loop is used to do the reading having as control variables the result of combining the states of the boolean variable *myitis* with the result of verifying whether $count \leq attempts$. The loop



```
 2def myfuncReadInt():
 3    """
 4       This function reads a whole number from the user in an
 5       interactive session. It returns None if data is wrong.
 6    """
 7    myitis = True ; count = 1; attempts = 3
 8    while myitis and count <= attempts:
 9      try:
10         mystr = 'Enter a whole number '
11         mystr = mystr + '(attempt {0} of {1}): '.format(count,attempts)
12         data = input(mystr)
13         data = int(data)
14         if data >= 0:
15            return data
16         else:
17            print('\t Negative values are not whole numbers!!!')
18            data = None ; myitis = True
19            count = count + 1
20      except Exception as errorCaptured:
21         print("When reading your input, something strange happened:")
22         print("\t  *** {0:s} ***".format( str(type(errorCaptured)) ))
23         count = count + 1
24         data = None ; myitis = True
25    return data
```

Figure 4.1: Function using *try--except* to read whole number.

ends whenever that result is *False*. Inside the loop, lines 10 − − 11 illustrates combinations of strings. The value is read via the instruction on line 12 (using the *input* function) and the result will be tried to be converted to an integer (using the function *int*) on line 13. In case something wrong happen in this type casting instruction, the flow of the program will continue with the execution of the body of the *except Exception as* instruction, where the error forking the flow of the program to this level is printed to the screen and the *counter* and *myitis* variables are reassigned. The while loop condition is evaluated and, according to the new state, the user receives or not a new opportunity to enter the required input value. If no error happen on line 13 (meaning that he user entered and accepted number), then the *if--else* statement verifies that the number is effectively a whole number, returning it to the calling program in line 15 or reassigning values to the variables *counter* and *myitis* in case the given number is negative (not a whole number). In the later case, The while loop condition is evaluated and, according to the new state, the user receives or not a new opportunity to enter the required input value. At the end, when the *while* loop if exited, the program returns to the calling program which receives either a whole number or the value *None* (in case the user could not entered an accepted value after the given three attempts). The following *IPython* session illustrates how to use the function:

Chapter 4, IPython session 6

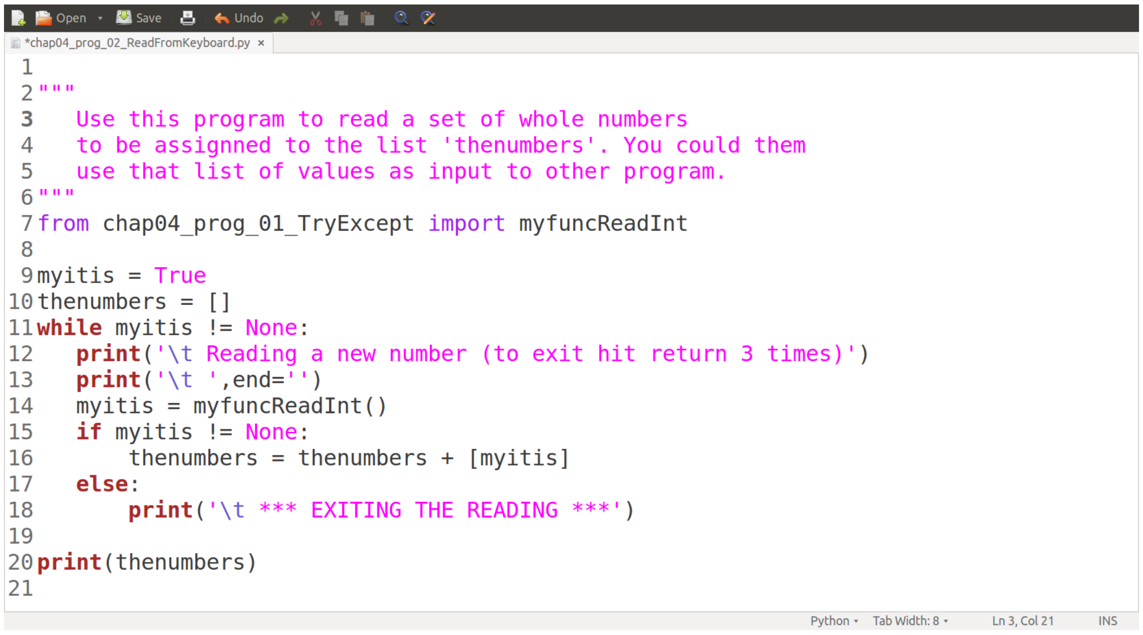
```
In [1]: from chap04_prog_01_TryExcept import myfuncReadInt

In [2]: ans = myfuncReadInt()
Enter a whole number (attempt 1 of 3): asa
When reading your input, something strange happened:
    *** <class 'ValueError'> ***
Enter a whole number (attempt 2 of 3): 2.8
When reading your input, something strange happened:
    *** <class 'ValueError'> ***
Enter a whole number (attempt 3 of 3): -3
   Negative values are not whole numbers!!!

In [3]: ans

In [4]: print(ans)
None

In [5]: ans = myfuncReadInt()
Enter a whole number (attempt 1 of 3): 234

In [6]: print(ans)
234

In [7]: ans = myfuncReadInt()
Enter a whole number (attempt 1 of 3): 0

In [8]: print(ans)
0

In [9]:
```

We encourage you to further explore the functionality of this function. Illustrative examples using this function will be shown in the following sections. First, in the next subsection we will use the function to read a set of values entered via the keyboard, and then, in the continuing section, we will use the function as part of a program to play with the computer the *guess two digits* game of section 2.8, whose discussion is presented starting on page 47.

4.3.1 Reading data entered via the keyboard

An example to further illustrate the usage of our function to read whole numbers entered via the keyboard and shown in Figure 4.1, on page 153, we will use it to read a set of whole number and assign them to a *list* object. These values can them be used to feed the programs to compute, for example, the statistical measures we studied in section 3.5, on page 78.

The program for reading the data set is shown in Figure 4.2, on page 155. You can find this program in the directory named `chapter_04` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap04_prog_02_ReadFromKeyboard.py`.



```
 1
 2"""
 3   Use this program to read a set of whole numbers
 4   to be assignned to the list 'thenumbers'. You could them
 5   use that list of values as input to other program.
 6"""
 7from chap04_prog_01_TryExcept import myfuncReadInt
 8
 9myitis = True
10thenumbers = []
11while myitis != None:
12   print('\t Reading a new number (to exit hit return 3 times)')
13   print('\t ',end='')
14   myitis = myfuncReadInt()
15   if myitis != None:
16       thenumbers = thenumbers + [myitis]
17   else:
18       print('\t *** EXITING THE READING ***')
19
20print(thenumbers)
21
```

Figure 4.2: Program to read a set of whole numbers entered via the keyboard.

As we have already discussed, the function in Figure 4.1 returns the *Python* special value *None* if the function receives (in the three options the user has) a wrong entry. As seen in input line 11 of Figure 4.2 that value is used to control the end for given data to the program. We are confident you can understand by know the rest of the lines of code in that program. Here we show a working example executing the program from a system shell or terminal:

Chapter 4, System shell command 1

```
$ python chap04_prog_02_ReadFromKeyboard.py
   Reading a new number (to exit hit return 3 times)
   Enter a whole number (attempt 1 of 3): 23
   Reading a new number (to exit hit return 3 times)
   Enter a whole number (attempt 1 of 3): 10
   Reading a new number (to exit hit return 3 times)
   Enter a whole number (attempt 1 of 3): 0
   Reading a new number (to exit hit return 3 times)
   Enter a whole number (attempt 1 of 3): 22
   Reading a new number (to exit hit return 3 times)
   Enter a whole number (attempt 1 of 3): 23
   Reading a new number (to exit hit return 3 times)
   Enter a whole number (attempt 1 of 3): 54
   Reading a new number (to exit hit return 3 times)
   Enter a whole number (attempt 1 of 3): 54
   Reading a new number (to exit hit return 3 times)
   Enter a whole number (attempt 1 of 3): 67
   Reading a new number (to exit hit return 3 times)
   Enter a whole number (attempt 1 of 3): 98
   Reading a new number (to exit hit return 3 times)
   Enter a whole number (attempt 1 of 3):
When reading your input, something strange happened:
    *** <class 'ValueError'> ***
Enter a whole number (attempt 2 of 3):
When reading your input, something strange happened:
    *** <class 'ValueError'> ***
Enter a whole number (attempt 3 of 3):
When reading your input, something strange happened:
    *** <class 'ValueError'> ***
   *** EXITING THE READING ***
[23, 10, 0, 22, 23, 54, 54, 67, 98]
```

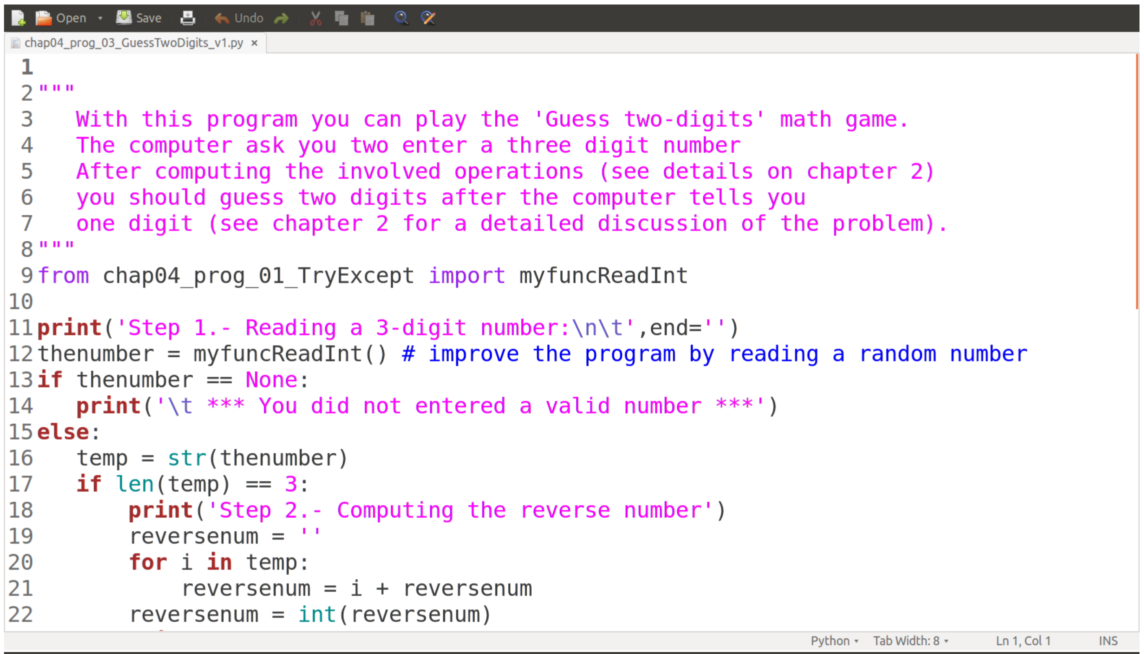
Notice that at the end of the program the entered numbers are printed to the computer screen, but as we have already mentioned they can be used to feed any other of the programs we have studied in the preceding sections and chapters requiring a set of values.

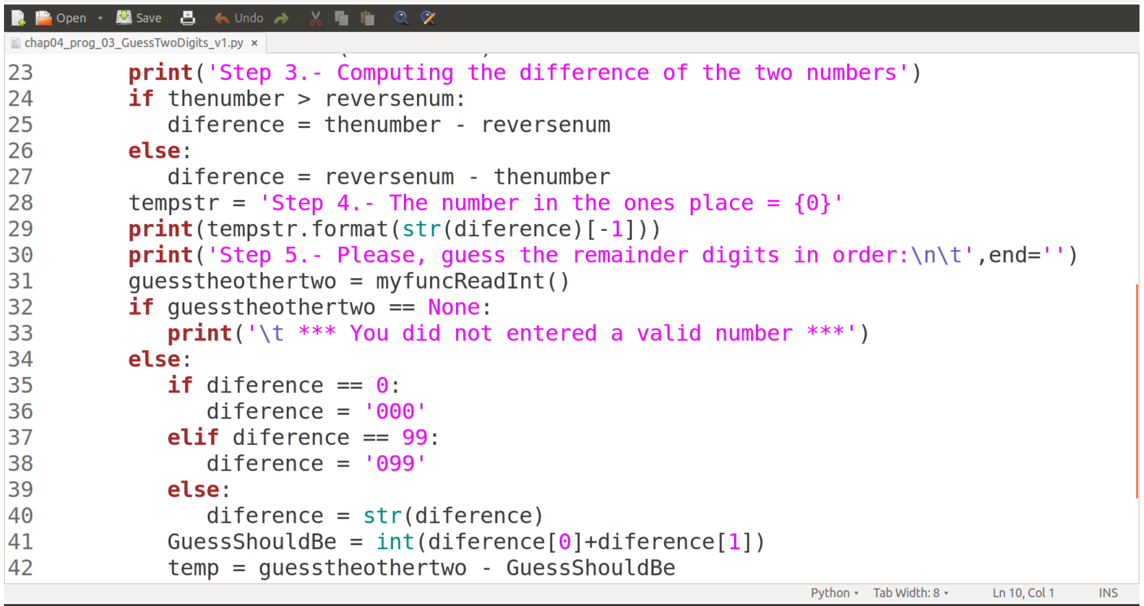
4.4 Programing the *guess two-digits* game

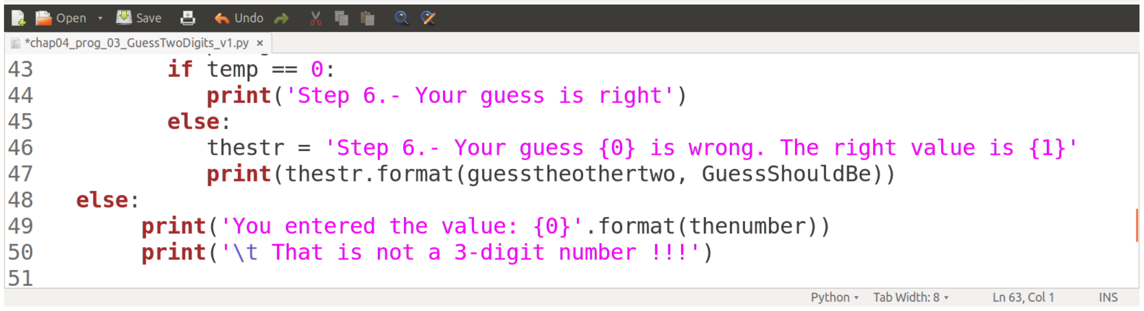
In this section we will discuss a program you can use to play the *guess two digits* game of section 2.8, whose discussion starts on page 47.

The program is shown in Figure 4.3, on page 157. You can find this function in the directory named `chapter_04` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap04_prog_03_GuessTwoDigits.py`.

Exploring the program you will see that only *Python* instructions that we have studied so far are used in this code (but see exercises 4.3 and 4.4, for some ways to improve the program). Following the flow of code (according to the instructions of the game) you will not have any major trouble understanding it.







```python

"""
   With this program you can play the 'Guess two-digits' math game.
   The computer ask you two enter a three digit number
   After computing the involved operations (see details on chapter 2)
   you should guess two digits after the computer tells you
   one digit (see chapter 2 for a detailed discussion of the problem).
"""
from chap04_prog_01_TryExcept import myfuncReadInt

print('Step 1.- Reading a 3-digit number:\n\t',end='')
thenumber = myfuncReadInt() # improve the program by reading a random number
if thenumber == None:
   print('\t *** You did not entered a valid number ***')
else:
   temp = str(thenumber)
   if len(temp) == 3:
       print('Step 2.- Computing the reverse number')
       reversenum = ''
       for i in temp:
           reversenum = i + reversenum
       reversenum = int(reversenum)
       print('Step 3.- Computing the difference of the two numbers')
       if thenumber > reversenum:
          diference = thenumber - reversenum
       else:
          diference = reversenum - thenumber
       tempstr = 'Step 4.- The number in the ones place = {0}'
       print(tempstr.format(str(diference)[-1]))
       print('Step 5.- Please, guess the remainder digits in order:\n\t',end='')
       guesstheothertwo = myfuncReadInt()
       if guesstheothertwo == None:
          print('\t *** You did not entered a valid number ***')
       else:
          if diference == 0:
             diference = '000'
          elif diference == 99:
             diference = '099'
          else:
             diference = str(diference)
          GuessShouldBe = int(diference[0]+diference[1])
          temp = guesstheothertwo - GuessShouldBe
          if temp == 0:
             print('Step 6.- Your guess is right')
          else:
             thestr = 'Step 6.- Your guess {0} is wrong. The right value is {1}'
             print(thestr.format(guesstheothertwo, GuessShouldBe))
   else:
        print('You entered the value: {0}'.format(thenumber))
        print('\t That is not a 3-digit number !!!')

```

Figure 4.3: Program of the two-digit game, section 2.8, page 47.

4.5 Programing the *guess a number* game

The *guess a number* is a popular game to help students of the elementary school to practice their understanding of the order of magnitude of numbers (as well as to improve their ability to follow instructions).

The game can be programmed in the computer using what is called the *bisection* method. This a powerful searching method which allows to divide a searching interval in half and then (after testing the search condition) pay attention to only one half to search what we are looking for. Then this new interval is again divided in half and then (after testing the search condition) pay attention only one half to search what we are looking for. This process is repeated again and again after one finds what we are looking for.

In the case of the *guess a number* game, you ask your partner to think of a whole number within a range (i.e. zero and a hundred). Then you ask your partner to tell you whether the number she or he have thought is lower/higher than the middle point of the chosen interval or neither (meaning you have guessed correctly the number). Then, if your guess is lower, you take as a new guess the middle point between zero and the previous middle point. If your guess is higher, you take as a new guess the middle point between the previous middle point and a hundred. After checking that the thought number is lower/higher or neither than this new guess, the process is repeated over and over until the guess is obtained.

We let as exercise 4.5 (page 166) for you to write a recipe (algorithm) to program this game so you can play it with the computer (which is the one that is going to guess the number you have thought). Test that the computer guess correctly the lower and upper limit in your interval. Also remember that the *Python*3 operator for integer division is double forward slash (//).

A sample session of yor program might looks like:

Chapter 4, System shell command 2

```
$ python chap04_prog_03_GuessNumber_exercise.py
   Please, think of a whole number between 0 and 100
   Hit return to continue
Is your secret number 50?.
  Enter 'h' to indicate the guess is too high.
  Enter 'l' to indicate the guess is too low.
  Enter 'n' to indicate neither of the above.
l
Is your secret number 75?.
  Enter 'h' to indicate the guess is too high.
  Enter 'l' to indicate the guess is too low.
  Enter 'n' to indicate neither of the above.
h
Is your secret number 62?.
```

```
   Enter 'h' to indicate the guess is too high.
   Enter 'l' to indicate the guess is too low.
   Enter 'n' to indicate neither of the above.
h
Is your secret number 56?.
   Enter 'h' to indicate the guess is too high.
   Enter 'l' to indicate the guess is too low.
   Enter 'n' to indicate neither of the above.
l
Is your secret number 59?.
   Enter 'h' to indicate the guess is too high.
   Enter 'l' to indicate the guess is too low.
   Enter 'n' to indicate neither of the above.
l
Is your secret number 60?.
   Enter 'h' to indicate the guess is too high.
   Enter 'l' to indicate the guess is too low.
   Enter 'n' to indicate neither of the above.
l
Is your secret number 61?.
   Enter 'h' to indicate the guess is too high.
   Enter 'l' to indicate the guess is too low.
   Enter 'n' to indicate neither of the above.
n
Game over. Your secret number was: 61
```

4.6 Writing and reading text files in *Python*

An importan aspect of a programming language is the ability to write and read files. There is a plethora of *Python* packages that offer versatile modules to execute these operation in many file formats (i.e. text and binary files, databases, and whatever other data type might be out there). In this book we will be using the buil-in *Python* function *open* to write and read text files. You are wellcome to find out about performing these tasks in other file formats perusing the references listed at the end of this chapter, on page 168. Since we already know how to capture data from the keyboard, we will start learning how it can be written to a (text) file, that then we want to read from a *Python* program.

4.6.1 Writing text files in *Python*

Once we have generated data using our program, we want to be able to save it to a file for further use of it. We will limit our discussion to the writing of *text* files that can later be opened with any text editor, like *gedit* discussed in section 3.2.1 (page 66). For that purpose, *Python* offer the build-in function *open* with the following basic syntax:

```
with open(filename, operation) as filehandler:
    filehandler.write(text1)
    filehandler.write(text2)
     ...
     ...
     ...
```

In this setup, *with*, *open*, and *as* are reserved *Python* keywords. The *filename* is any string of characters that can be used as the name of a file in your operating system. It is the name of the file were the data will be saved or written. The *operation* can take (in this book) the string value `'wt'` or `"wt"` (single or double quotes included) for writing (or overwriting) a text file. It can also take the string value `'at'` or `"at"` (single or double quotes included) for appending (not overwriting) data to an existing text file. The *filehandler* is any *Python* variable name that is used to access the file (via its *filename*) for writing, as is shown in the body of the *with* instruction, represented by the indented lines under it. Using the *with* instruction has the advantage of properly closing the file when finishing executing its last (indented) line of code.

You could explore more about the *open* function via executing the instruction `open?` or `open??` in any *IPython* input cell. In the reference section of this chapter, on page 168 you will find pointers to other sources discussing the topic.

The program in Figure 4.4, on page 161, illustrates the use of this approach. You can find this program in the directory named `chapter_04` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `gedit_my_program_04_writingTest.py`.

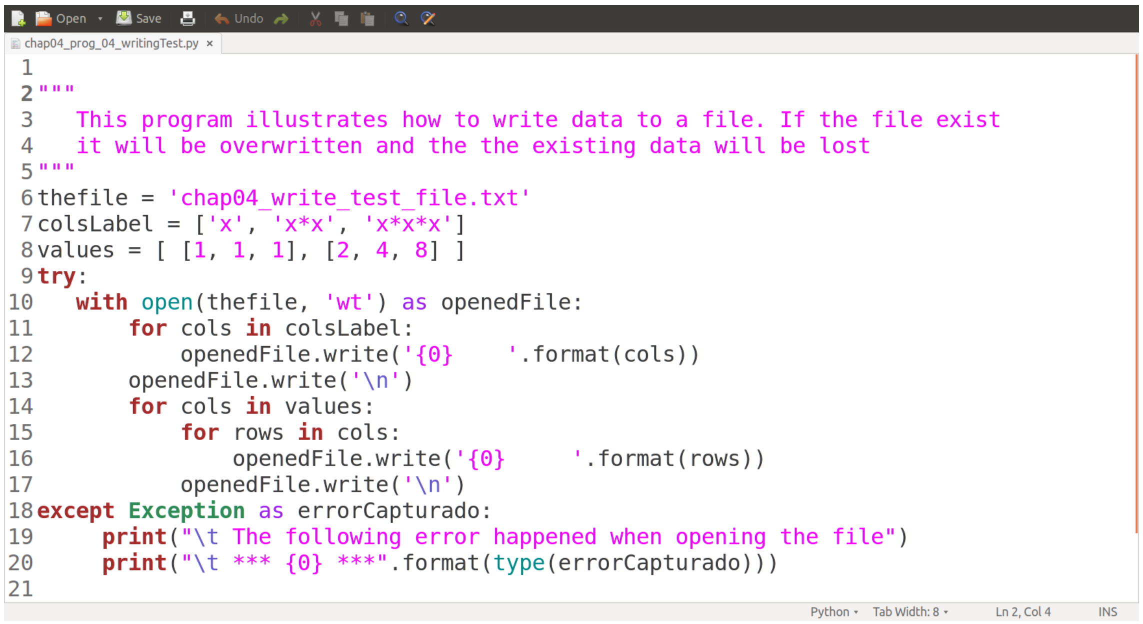
Executing this program from the system (terminal) shell (or from an *IPython* console) will create (in the same directory where the code is executed) the file named `chap04_write_test_file.txt`, whose contents is shown below:

Chapter 4, System shell command 3

```
$ python chap04_prog_04_writingTest.py
$ more chap04_write_test_file.txt
x   x*x   x*x*x
1    1    1
2    4    8
```

A few comments about the program are in order:

- Notice that the written file contains three space separated columns and three rows, being the first row a header containing the names of each column.
- Strings and numeric values can be written to the same file using the same instruction.
- In case something goes wrong opening the file, it is reported via the *except Exception as* instruction. In there, you need to devise any other set of actions to properly save your data, specially if your program has been running for quite some time.
- The lines of code 13 and 17 adds the newline character (`'\n'`) after finishing writing the last value in the respective row. When reading the data, each line will have its particular place. You can distinguish them via that newline character. We will see that in the next section.
- Pay attention to the extra white spaces included in the writing instructions in lines 12 and 16. This is what makes (already mentioned) the space separation between columns in the written file.
- This program is general enough that it is worthwhile making it a function (see exercise 4.7, on page 166).



```
 1
 2 """
 3    This program illustrates how to write data to a file. If the file exist
 4    it will be overwritten and the the existing data will be lost
 5 """
 6 thefile = 'chap04_write_test_file.txt'
 7 colsLabel = ['x', 'x*x', 'x*x*x']
 8 values = [ [1, 1, 1], [2, 4, 8] ]
 9 try:
10    with open(thefile, 'wt') as openedFile:
11        for cols in colsLabel:
12            openedFile.write('{0}    '.format(cols))
13        openedFile.write('\n')
14        for cols in values:
15            for rows in cols:
16                openedFile.write('{0}     '.format(rows))
17            openedFile.write('\n')
18 except Exception as errorCapturado:
19     print("\t The following error happened when opening the file")
20     print("\t *** {0} ***".format(type(errorCapturado)))
21
```

Figure 4.4: Program showing how to write data to a file.

4.6.2 Reading text files in *Python*

So far we have been working with data entered via the computer keyboard. This is required for small data set of data. Most of the time, the processing power of the computer is used to operate on large data sets, stored in a variety of file formats.

In our case, We will limit our discussion to reading *text* files that can be opened with any text editor, like *gedit* discussed in section 3.2.1 (page 66). For that purpose, *Python* offer the build-in function *open* with the following basic syntax:

```
with open(filename, 'rt') as filehandler:
   filehandler.read(text1)
   filehandler.read(text2)
       ...
       ...
       ...
```

In this setup, *with*, *open*, and *as* are reserved *Python* keywords. The *filename* is any string of characters that can be used as the name of a file in your operating system. It is the name of the file from which the data will be read. Notice that the *operation* of reading text files is specified by the string value `'rt'`.

The *filehandler* is any *Python* variable name that is used to access the the file (via its *filename*) for reading, as is shown in the body of the *with* instruction, represented by the indented lines under it. Using the *with* instruction has the advantage of properly closing the file when finishing executing its last (indented) line of code.

You could explore more about the *open* function via executing the instruction `open?` or `open??` in any *IPython* input cell. In the reference section of this chapter, on page 168 you will find pointers to other sources discussing the topic.

The program in Figure 4.5, on page 165, illustrates the reading of the file written using our program of page 161. You can find the reading program in the directory named `chapter_04` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface. In there, find the file named `chap04_prog_05_readingTest.py`.

Before continuing, let's point out that reading data from a file requires knowing its structure, so you can read it directly to be used to perform right away computations on it. Other files might require extra pre-processing before obtaining the data ready to operate on it (we will not be concerned with such files in this text). Consequently, before reading data from a file we nee to understand how data has been organized on it.

Consequently, when perusing the reading code of Figure 4.5 (page 165), keep in mind (see page 160) that the written file (that we named `chap04_write_test_file.txt`) contains three space separated columns and three rows, being the first row a header containing the names of each column. This means that it should be read as a string type variable. The other rows contains numerical values and they can be read and converted to numbers of type integers.

Executing this program from the system (terminal) shell (or from an *IPython* console) shows the following output:

Chapter 4, System shell command 4

```
$ python chap04_prog_05_readingTest.py
Read line from file  = x   x*x   x*x*x
strip-split processed row = ['x', '', '', '', 'x*x', '', '', '',
   'x*x*x']
removed '' processed row = ['x', 'x*x', 'x*x*x']


Read line from file  = 1    1    1
strip-split processed row = ['1', '', '', '', '', '1', '', '', '',
   '', '1']
removed '' processed row = ['1', '1', '1']


Read line from file  = 2    4    8
strip-split processed row = ['2', '', '', '', '', '4', '', '', '',
   '', '8']
removed '' processed row = ['2', '4', '8']


Data read in a list:
['x', 'x*x', 'x*x*x']
[1, 1, 1]
[2, 4, 8]
```

A few comments about the program are in order:

- Let's emphazise that the this program works for text files with data in space separated columns, having the first row as a header (not a data entry to be used in any further computation). If the first row is also numerical data, then you need to change the line of code 25 to read $row = 0$. Otherwise, the first row will be considered a header.
- The *print* statements in lines 11, 13, 16, and 18 are unnecessary. You need to comment or delete them from the program. They are there to show what is happening at intermediated steps.
- The *for loop* in line 10 indicates the file is being read line by line. Each line is captured in the variable *row* controling the execution of the loop. From the output on page 162 you can see that each line of the file is read as a *string Python* type.
- The body of the loop is a set of preprocessing steps to put the data in a format ready to perform numerical comptations on it. First, in line of code 12 the *row* is cleaned out of the newline character at the end of the row via the string *strip()* method. Then, the row is divided into pieces separated by white spaces `' '` and asigned to a list. We use this

strategy because at least one white space caracter should be within columns. Via lines of code `14--15` the row is further cleaned from this extra non-data characters. Finally, the row is appended to the *list* named *data*. This process continues with the next *row* in the file and goes on until the last line of the file is read and appended to *data*.
- In case something is wrong opening the file, it is reported via the *except Exception as* instruction. In there, this time we included the instruction *sys.exit(1)* from the module *sys*, imported to the current computational environment in line of code 20. If executed, this line of code makes the program to finish execution. This line of code is included because there is no sense continuing the execution of the program if the data can not be read from the file.
- Lines of code `25--31` represents the additional preprocessing of converting the necessary data to numrical values (integers in this case). These lines of code can be converted in fewer lines using the *Python* built-in *map* function (see exercise 4.8 (page 166).

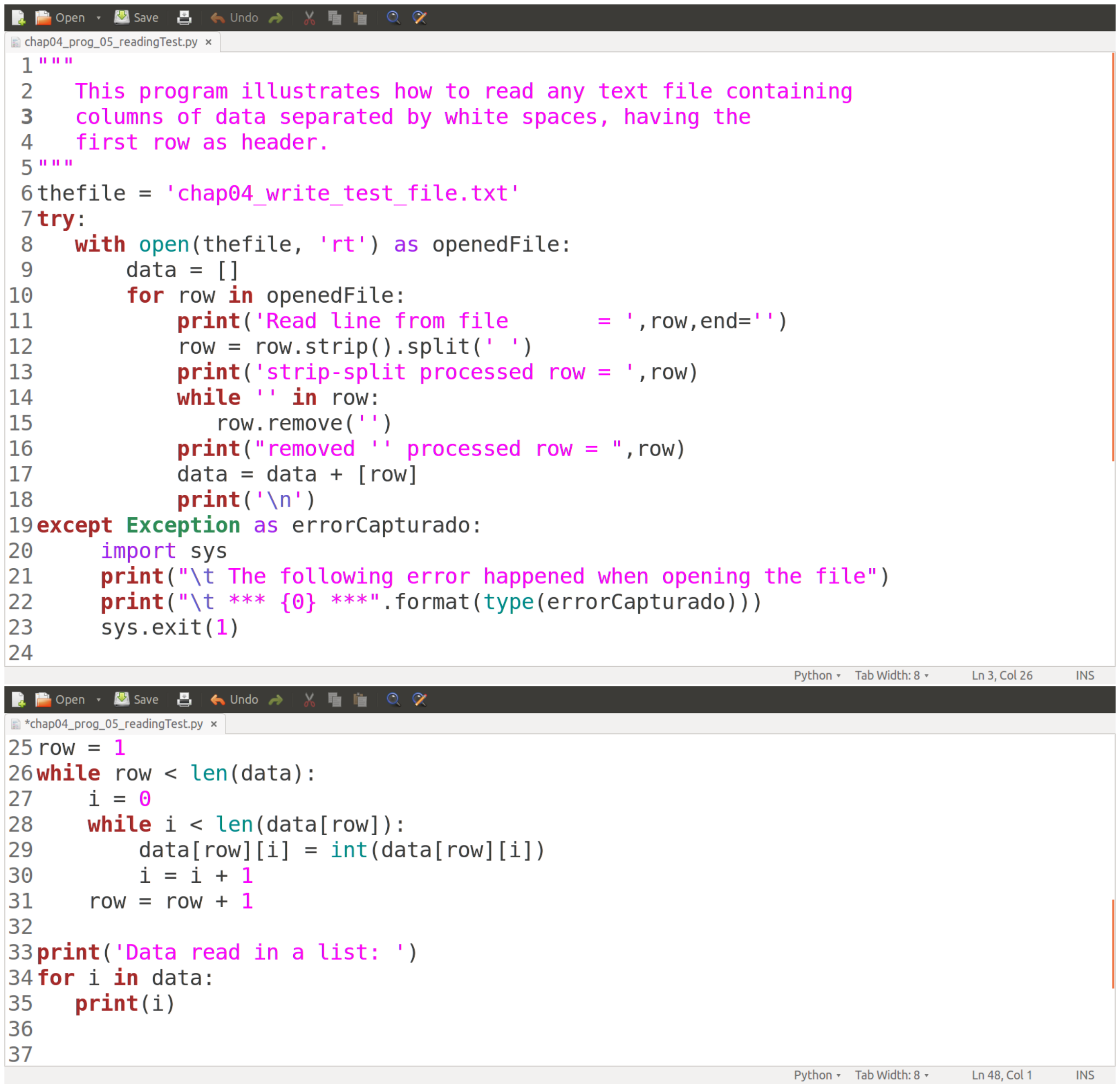
4.7 Chapter Summary

Reaching the end of this chapter you are now equipped with the important issue that bring your *Python* computing power to a higher stage. As time goes, you'll learn more advance topics on file handling via a more appropriated learning and trying setup. Remember that your major interest should be Prealgebra so you can apply its computing power in other fields (physics, chemistry, biology, engineering, poetry, human rights, you continue naming it).

Being able to read and write text files allows you to perform computations in large data sets. Perhaps you can help your school department (or your teacher in a small research endeavor) in computing the average grades of the whole school and discriminate them by age, gender, social habits, and so forth. The school don't need to spent unnecessary money in private software to do so.

Coming back to what we have learned in this chapter, being capable of reading and writing text files is more than enough to perform high level computing explorations in your Prealgebra course work. As mentioned elsewhere in the chapter, we have made major effort in applying what the basics of programming common to any programming language (*if--else* statements, *while* and *for* loops, *variables* and *function* definitions) which you can further practice trying non-traditional computing problems in your Prealgebra course work (without any doubt, you could also understand the flow of a code in any other computer language that will allow you to enrich your *Python* constructions).

In the next chapter we will learn about the elements of the nuances of the *Python float* data type as the way of representing real numbers in the computer.



```python
"""
   This program illustrates how to read any text file containing
   columns of data separated by white spaces, having the
   first row as header.
"""
thefile = 'chap04_write_test_file.txt'
try:
   with open(thefile, 'rt') as openedFile:
       data = []
       for row in openedFile:
           print('Read line from file       = ',row,end='')
           row = row.strip().split(' ')
           print('strip-split processed row = ',row)
           while '' in row:
              row.remove('')
           print("removed '' processed row = ",row)
           data = data + [row]
           print('\n')
except Exception as errorCapturado:
     import sys
     print("\t The following error happened when opening the file")
     print("\t *** {0} ***".format(type(errorCapturado)))
     sys.exit(1)

row = 1
while row < len(data):
    i = 0
    while i < len(data[row]):
        data[row][i] = int(data[row][i])
        i = i + 1
    row = row + 1

print('Data read in a list: ')
for i in data:
   print(i)


```

Figure 4.5: Program showing how to read space separated columns data from a file.

Exercises of Chapter 4

Exercise 4.1 *Modify the function of Figure 4.1, on page 153 to read whole numbers such that the user can only has two attempt to type correctly the number.*

Exercise 4.2 *Modify the function of Figure 4.1, on page 153 to read and return any entry (the function should never return* None*) to the calling program. Is it helpful to have a function to perform this task?*

Exercise 4.3 *The program in Figure 4.3, on page 157, can have a few lines less if we replace the lines of code* 19--21 *by the instruction* `reversenum = temp[::-1]` *followed by the other (which you could make one, if you wish)* `reversenum = int(reversenum)`. *In addition, you could replace the lines of code* 24--27 *by the instruction* `difference = abs(reversenum - thenumber)` *(why?). Please make those changes and make sure you understand them.*

Exercise 4.4 *The program in Figure 4.3, on page 157, can be further improved by setting the line of code* 12 *via randomly drawing the required three digit whole number. This way, the computer will be the one setting up the beginning of the game. Please, perform such changes. For this you might one to reread section 3.6.*

Exercise 4.5 *Write a recipe (algorithm) to program the guess a number game as described in section chap04:sec:GuessAnumber (page 158). Implement in* Python *your recipe so you can play the game with the computer (which is the one that is going to guess the number you have thought). Test that the computer guess correctly the lower and upper limit in your interval. Also remember that the* Python*3 operator for integer division is a double forward slash (//). What happen is the user answer incorrectly any of the questions? Could you write code instructions to handle those cases?*

Exercise 4.6 *Make the program of Figure 4.4 (page 161) to append data to an existing file. After modifying the program, you can test it in the same data file by executing the program consecutively twice or several times.*

Exercise 4.7 *Make the program of Figure 4.4 (page 161) a function taking as arguments, passed by the user, the filename, the labels to each column of data, the data in a list properly organized to be written in columns, and the writing mode (whether the writing operation will overwrite and existing file or append data to it).*

Exercise 4.8 *Modify the program of Figure 4.5 (page 165) replacing lines of code* `25--31` *by the lines:*

```
row = 1
while row < len(data):
    data[row] = list(map(int,data[row]))
    row = row + 1
```

Verify the code works as expected.

Exercise 4.9 *Make the program of Figure 4.5 (page 165) a function taking as arguments, passed by the user, the filename to be read. The function should return the data as printed in lines of code* `34--35`*.*

References of Chapter 4

Books and/or Articles

- **Marecek, L. and Smith, M. A.** (2017). Prealgebra, Rice University, OpenStax `https://openstax.org`.
  Book available for free at: `http://cnx.org/content/col11756/1.9`

References on the WEB

- **Python 3 tutorial**:
  `https://docs.python.org/3/tutorial/index.html`
- **Input/output in Python**:
  `https://docs.python.org/3/tutorial/inputoutput.html`
  `https://docs.python.org/3/library/functions.html#open`
  `https://docs.python.org/3/reference/compound_stmts.html#the-with-statement` `https://docs.python.org/3/library/sys.html`
- **Input/output alternatives in Python**:

  `http://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html`
  `http://docs.scipy.org/doc/numpy/reference/generated/numpy.genfromtxt.html`
  `http://pandas.pydata.org/`
  `https://wiki.python.org/moin/DatabaseInterfaces`
  `FINDHDF5inpythonhttp://hdf.ncsa.uiuc.edu/HDF5/doc/PSandPDF/`
- **Errors and exceptions in Python**:
  `https://docs.python.org/3/tutorial/errors.html` `https://docs.python.org/3/library/exceptions.html`

Integers and Rational numbers in *Python*

*“ **Mathematics is not just a tool by means of which phenomena can be calculated, it is the main source of concepts and principles by means of which new theories can be created.**”*
Freeman Dyson

5.1 Introductory remarks

Before starting, let’s remind ourselves where we are standing. Studying whole numbers and its basic operations, we have introduced a good deal of *Python* programming instructions which includes assignments to variable, boolean comparisons, repeatability via looping constructs, function abstraction, and input from and output to text files. We have also introduced some *Python* built-in functions, the most important of which is the *Python list* object which allows the encapsulation of different data types in one place facilitating its uses for performing computations on them. This basic knowledge allows us to basically compute anything that is computable, but we still needs to study the representation of numbers in *Python* beyond whole numbers.

Nevertheless, in the context of the restricted field of whole numbers, the introduced *Python* programming instructions have given you the ability to write programs to deal with non-traditional Prealgebra course work problems (i.e. the sailors, the coconuts, and the monkeys problem). But we also wrote some *Python* programs to perform specific computations from your Prealgebra course work (i.e. *mean*, *median*, *mode*, and some others) taking as input whole numbers and given as output whole numbers. We mentioned that some of the programs were general enough to accept as input other types of numbers (integers, reals, or even complex type ones) to perform the desired (meaningful) computations on them (keeping an eye that some computations have only meaning in the context of whole numbers).

In reading this chapter, you are suppose to have read and (surely you had) understood the preceding chapters as we will using in here basically all of the *Python* instructions we introduced in these preceding chapters. We will also include a few new *Python* built in functions as we need them to deal with the numbers in this chapter.

To be specific, in this chapter we will give a quick introduction to integer and how to operate with them in *Python*, continuing then with a more lengthy discussion about fractions and fractional expressions.

You are suppose to get acquainted with the theoretical background about these topics in your

Prealgebra course work. As is the characteristic in this book, *Python* will be used as a means to enhance and enrich your understanding of these topics by allowing you explore non-traditional course work applications as they might be too long or tedious to be performed manually (using pen and paper). Just keep in mind that if you do not understand basic Prealgebra operations and their relationship you could be trap in just doing operations mechanically. Your programming experience is suppose to help your thinking become increasingly abstract by building on concrete understandings via conscious problem-solving.

Accordingly, after finishing this chapter, you'll be equipped with a basic set of *Python* tools that will allow you explore computations in a much wider sense, including integers and fractions. In the next chapter you will go to a higher level by understanding how to work with real numbers in *Python*.

We will not rest in repeating that you should always keep in mind that a major goal of your investment reading this book is (as an independent learner) to enhance your skills in applying what you have learned to tackle new first (unfamiliar) met situations (not only in Prealgebra course work but also in any other of the subjects in your educational track like Physics, Chemistry, Biology, Sociology, Psychology, and so on)). The successful acquisition of such desired outcome requires your engagement in an effective learning involvement of constantly applying the programming stages of *designing* (actions), *implementing* (the actions in the required order), and *assessing* (the performance of such actions) to any other real world situations.

5.2 Computing with integers in *Python*

Whole numbers, the ones studied in preceding chapter, covers only zero and numbers greater than zero. As you know from daily language like "some degrees below zero," or "some feet below sea level" that whole numbers are a subset of a much wider set of numbers that needs to be used to quantitatively represents the meaning of the referred expressions. This numbers are the *integers*, which includes the sign plus (+) or minus ($-$) to represent respectively greater than zero (positive) and less than zero (negative) numbers. The zero value is unsigned.

In symbols, for any integer number m, if it is negative it is represented using the relational operator ($<$) in the form $m < 0$; for m positive, it is represented using the relational operator ($>$) in the form $m > 0$; and for m zero, it is represented using the relational operator ($=$) in the form $m = 0$ (recall that in programming the equal sign means assignment, while the equality sign is represented by a double equal $==$ sign). Another common representation is given in terms of the symbols ($\leq$) and ($\geq$). Whenever m is less or equal to some other number n is represented in the form $m \leq n$. Whenever m is greater or equal to some other number n is represented in the form $m \geq n$. These representations in *Python* are expresses in the form ($<=$), for less and equal to, and by ($>=$), for greater or equal to.

A positive number can be indicated by placing the plus sign (+) in front of the number, like +4. Usually the sign is omitted and it is understood that the number is a positive number. Any positive number is always greater than zero and it is said to lie at the right of zero on the numbered line. A negative number is indicated by placing the negative sign ($-$) in front of the number, like -4. The sign for negative numbers must always be written. Any negative number is always less than any positive number. Similarly, any negative number is always less than zero and they are said to lie at the left of zero on the numbered line.

The following *IPython* session shows how to express this relations in *Python*:

Chapter 5, IPython session 1

```
In [1]: -5 >= 2
Out[1]: False

In [2]: -5 >= -4
Out[2]: False

In [3]: -2 > 0
Out[3]: False

In [4]: 2 > 0
Out[4]: True

In [5]: 5 > 4
Out[5]: True

In [6]: -2 == -2
Out[6]: True

In [7]: -2 == 2
Out[7]: False

In [8]:
```

5.2.1 Operations with integers

As in the case of whole numbers, the basic operations of addition, subtraction, multiplication, division, and exponentiation are implemented in *Python* via the operators we are familiar with from section 2.3, page 25. The following *IPython* session illustrates them:

Chapter 5, IPython session 2

```
In [1]: a = -2

In [2]: b = 29
In [3]: c = - 5

In [4]: -a
Out[4]: 2

In [5]: -a == -1*a
Out[5]: True

In [6]: a + b + c == (a + b) + c
Out[6]: True

In [7]: a**3
Out[7]: -8

In [8]: a**(-3)
Out[8]: -0.125

In [9]: b // c
Out[9]: -6

In [10]: b % c
Out[10]: -1

In [11]: b / c
Out[11]: -5.8

In [12]: b*c
Out[12]: -145

In [13]: a*(b+c)
Out[13]: -48

In [14]: c
Out[14]: -5

In [15]: abs(c)
Out[15]: 5

In [16]:
```

Notice that on input cell `In [15]:` we have used the *Python* built-in function *abs* [`https://docs.python.org/3/library/functions.html#abs`]. which is used to make a negative number positive (it is equivalent to multiplying a negative number by negative one, while leaving the number without change if it is positive). In other words, the function *abs* is the pythonic way of getting the *absolute value* of any number. We can use this function to guarantee that a given input is of the positive type required by a program. For instance, since a distance or a length can never be a negative number, we can use this function to ensure that it always happen that such a measurements are always a positive number or zero.

From the operational point of view, you see that operating with integers implies to keep in mind that they could take negative values.

By the way, you might remember that when working with *Python list* object negative integers are used to to refer to elements in the *list* from right to left, being the index -1 the rightmost element in the *list*. The following *IPython* session recalls the fact:

Chapter 5, IPython session 3

```
In [1]: alist = [0, 1, 2, 3, 4, 5]

In [2]: alist[5]
Out[2]: 5

In [3]: alist[-1]
Out[3]: 5

In [4]: alist[-2]
Out[4]: 4

In [5]: alist[-6]
Out[5]: 0

In [6]: alist[0]
Out[6]: 0

In [7]:
```

What follows is an *IPython* session on which we can check some properties of working with integers:

Chapter 5, IPython session 4

```
In [11]: -12/-4
Out[11]: 3.0

In [12]: -12//-4
Out[12]: 3

In [13]: 12//-4
Out[13]: -3

In [14]: -12//4
Out[14]: -3

In [15]: -3*-4
Out[15]: 12

In [16]: 3*-4
Out[16]: -12

In [17]: -3*4
Out[17]: -12

In [18]: 2*(-3)*(-5)*(-7)
Out[18]: -210

In [19]:
```

5.2.2 The *least common multiple* (LCM) of natural numbers in *Python*

A previous knowledge to work with fractions requires computing the *least common multiple* (LCM) and the *greatest common divisor* (GCD) (also known as *greatest common factor* (GCF)) of a set of natural numbers (the later has been discussed in section 3.8.5, page 114). The multiples of a number a are the values multiplying a by 1, 2, 3, and so forth. For example, the multiples of 3 are $3 \times 1 = 3$, $3 \times 2 = 6$, $3 \times 3 = 9$, $3 \times 4 = 12$, etc.

A number that is a multiple of two or more numbers is a common multiple of those numbers. This let to the idea that the least common multiple (LCM) is the smallest common multiple of two or more natural numbers. For example the LCM of 3 and 5 is 15, and between 6 and 8 is 24. Recipes to computing the LCM can be given in terms of listing the factors of each number and then finding the smallest factor common to each listing. Another alternative goes by finding the prime factorization of each number and them taking as the LCM of them the product of the prime factors with the highest power that they appear. Instead of writing our own program to do that, we will use *SymPy* already programmed function to do that.

Regarding the GCD (or GCF) of a set of numbers (section 3.8.5, page 114) let just recall that it is is the largest common factor of two or more numbers (remember that a factor of a number is any number that divides the former evenly. For instance the factors of 15 are the numbers 1, 3, 5, and 15).

SymPy contains functions to perform LCM and GCD of natural numbers:

Chapter 5, IPython session 5

```
In [9]: from sympy.core.numbers import ilcm

In [10]: ilcm?
Signature: ilcm(*args)
Docstring:
Computes integer least common multiple.
...
... extra output deleted
...
In [11]: ilcm(5, 10)
Out[11]: 10

In [12]: ilcm(7, 3)
Out[12]: 21

In [13]: ilcm(5, 10, 15)
Out[13]: 30

In [14]: from sympy.core.numbers import igcd

In [15]: igcd?
Signature: igcd(*args)
Docstring:
Computes nonnegative integer greatest common divisor.
...
... extra output deleted
...
In [16]: igcd(5, 10, 15)
Out[16]: 5

In [17]:
```

You need to review your Prealgebra instructional notes to refresh your knowledge of these two common devices (LCM and GCD/GCF). Make sure you understand the procedure to find them. Remember that *SymPy* hides them from you. Exercise 5.5 (page 201) ask you to write and implement an algorithm to compute the LCM of two or more natural numbers.

5.2.3 Solving equations nvolving integers via *SymPy*

As we did with whole numbers (page 119, we can use *SymPy* to solve equations whose solutions can be any type of numbers, including integers. Consequently, after reading this section, you might want to read also the corresponding sections about equations with whole numbers (page 119), fractions (page 195), and decimals (page 230). You could also practice your skills in solving equations via the one variable equation solver described on page 236.

Let's start by posing the problem of finding the solution of:

$$7 = x + 9 \tag{5.1}$$

Surely your Prealgebra textbook has plenty of similar equations stated just like that, and perhaps you might be wondering how can you get that (u other similar) equation in a real world experience? After a while of talking with your classmates, you might ends up with a word problem leading to equation 5.1 in the context of a real world experience imagining that you have at home a black box with some *stuff* in it. One of your buddies comes to visit and put 9 units of extra *stuff* in the black box, and by some measurement (perhaps by weighing the black box using a lever balance) your buddy determined that such adding resulted in 7 units of total *stuff* in the black box. Since you did know how much *stuff* were in the black box at the beginning, you set up equation refchap05:eq:01 to find out how much *stuff* X was in the black box initially.

A more realistic way of phrasing the problem is in a story involving a first grade class, were, during the break period, two children mixed their toys of the same type. After the break time is over, the teacher was helping the children to get back theirs toys. No much convincing, one of them tells the teacher that he brought to the game 9 toys, while the other child smiling with innocence happily tells the teacher that after mixing the toys they counted the toys and found that only 7 toys were in the pile. The first child was OK with the recount of the second child and even told the teacher how they were having fun counting the toys after mixing them. The teacher knew there was a problem with their story, but wanted to use it to reinforce on the children the sense of ordering. By calling x the number of unknown toys, the teacher ends up with equation 5.1, which she applied with the toys to tell the children that something was wrong with theirs counting stories.

As you can see, we can always find a context in which an equation could make sense. Perhaps you could make up your own story to put this equation in another context. But let's now see how to find the solution of the equation via *SymPy*. Just keep in mind that *SymPy* will hide from you how it finds the solution of the equation. Consequently, you are encourage to workout the exercise following by hand (using pencil and paper) the procedures for solving equations presented in your Prealgebra course work. This way you will develop your intuition on whether the solution found make or not sense and how to check it.

A piece of code solving this equation (which you can find under the name chap05_prog_01_SympyWithIntegers.py in the directory named chapter_05 of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface of this book) is as follows:

```
from sympy import symbols, Eq, solveset
x = symbols('x')

LHS = 7
RHS = x + 9

thesol = list( solveset( Eq(LHS, RHS), x) )
print('thesol =', thesol)

newLHS = LHS - RHS #rearrange the equation to read: LHS - RHS = 0
print('newLHS =', newLHS)

newLHS = newLHS.subs(x, thesol[0])
if newLHS.simplify() == 0:
   print('The solution of {0} = {1}, is x =
      {2}'.format(LHS,RHS,thesol[0]))
```

After executing these lines of code you'll get:

Chapter 5, System shell command 1

```
$ python chap05_prog_01_SympyWithIntegers.py
thesol = [-2]
newLHS = -x - 2
The solution of 7 = x + 9, is x = -2
```

If you have trouble understanding this code, please go back and read section 3.9, starting on page 119.

Coming back to the found solution, what in the world could be thinking the teacher after getting this solution? Does it make sense in the case of the *stuff* interpretation in the black box? What king of *stuff* the friends were dealing with? Could it be be raw potatoes? What about smash potatos? Were they mixing up electrons and protons? Again, remember that after solving an equation one needs to make sense of the solution, we need to think if it make or not sense.

To further enhance your programming skills, knowing that the solution is an integer, exercise 5.2 (page 201) ask you to write a program to find the solution of the equation 5.1 not using *SymPy*.

5.3 Fractions and how to represent them in *Python*

After dealing with *whole numbers* and *integers*, your next experience with our current numerical system in your Prealgebra course work is with fractions, which you can think of dividing the unity in portions (like when dividing your birthday cake).

Python provides at least two approaches to deal with fractions as an indicated (not performed) division of the numerator and the denominator when the result is not an integer (like in 3/8). One is via the module *SymPy* and the other is via the module *fractions*. We will show both options, but we will work more on *SymPy* as it is the *Python* module for symbolic computations. Working with fractions is a bit cumbersome, but we will get use to it.

Before continuing, let's mention that we will not made distinction between proper and improper fractions. Computationally, both are the same (check exercises 5.3--5.4, on page 201). Related to this point, be aware that when *SymPy* receives a fraction, it is automatically simplified (or reduced), performing the division of common factors between the numerator and denominator.

We have seen that *Python* has two special symbols for division: the standard (in *Python*3) forward-slash (/) which performs the normal division operation between the numerator and the denominator (as your are familiar with your calculator) and the double-forward slash (//) which returns the integer part of dividing the numerator by the denominator. In standard *Python* operations, when using those symbols we will get a number, as shown in the following *IPython* session:

Chapter 5, IPython session 6

```
In [1]: 8/3
Out[1]: 2.6666666666666665

In [2]: 8//3
Out[2]: 2

In [3]: 3/8
Out[3]: 0.375

In [4]: 3//8
Out[4]: 0
```

```
In [5]: 3/8 + 8/3
Out[5]: 3.0416666666666665

In [6]:
```

To work with fractions in the standard way as presented in the Prealgebra course work we need to use either the *SymPy* module or the *fractions* module, but we need to define the numbers in a special way to avoid that the indicated division be executed by the *Python* interpreter.

5.3.1 Representing fractions using *sympy* functions *S* and *Rational*

Let's see how to represent fractions in *SymPy*:

Chapter 5, IPython session 7

```
In [6]: from sympy import S

In [7]: S('8')/S('3')
Out[7]: 8/3

In [8]: S('8')//S('3')
Out[8]: 2

In [9]: S('3')//S('8')
Out[9]: 0

In [10]: S('3')/S('8')
Out[10]: 3/8

In [11]: S('3')/S('8') + S('8')/S('3')
Out[11]: 73/24

In [12]: S('3')/18
Out[12]: 1/6

In [13]: 3/S('8')
Out[13]: 3/8

In [14]: 3./S('8')
Out[14]: 0.375000000000000

In [15]: (3./S('8')).n()
```

```
Out[15]: 0.375000000000000

In [16]: a = 3/S('8')

In [17]: a
Out[17]: 3/8

In [18]: a.n()
Out[18]: 0.375000000000000

In [19]: a.p
Out[19]: 3

In [20]: a.q
Out[20]: 8

In [21]: from sympy import Rational

In [22]: b = Rational(3,8)

In [23]: b
Out[23]: 3/8

In [24]: a - b
Out[24]: 0

In [25]: type(a)
Out[25]: sympy.core.numbers.Rational

In [26]: type(b)
Out[26]: sympy.core.numbers.Rational

In [27]:
```

Let's first notice that we need the special *SymPy* function S (called to the current computational environment on input cell `In [6]:`). Second, at least one of the numerical values forming the fraction must be represented as a string passed as argument to the function S (i.e *S('number')*). Third, the division symbols retains theirs meanings. Consequently, the forward slash symbol is the one to use to operate with fractions. The input cells `In [14]:` and `In [15]:` shows two alternatives to get the division performed in case we need it. The special method *.n()*, on input cell `In [15]:`, is particularly useful when dealing variables, as shown on input cell `In [18]:`. Defining fractions assigned to a variable seems to be the most affordable way to work with fractions. Input cells `In [19]:` and `In [20]:` hows how to extract the numerator and denominator of a fraction. Input cells `In [21]:`--`In [23]:` show an alternative way to represent fractions in *SymPy* via the *Rational* function, which in turns is equivalent (both represent the same type of *SymPy* objects) to the way using the S function as seen from the output of the function *type* (applied to the objects a and b) in cells `Out[25]:` and `Out[26]:`.

5.3.2 Representing fractions using the module *fractions*

Let's now see how to represent fractions using the *fractions* module, and then we return with *SymPy* to the rest of the chapter:

Chapter 5, IPython session 8

```
In [27]: from fractions import Fraction

In [28]: Fraction(8,3)
Out[28]: Fraction(8, 3)

In [29]: Fraction(8,3) + Fraction(3,8)
Out[29]: Fraction(73, 24)

In [30]: Fraction(8,3) + 3/S('8')
Out[30]: 73/24

In [31]: b = Fraction(8, 3)

In [32]: b
Out[32]: Fraction(8, 3)

In [33]: a
Out[33]: 3/8

In [34]: a + b
Out[34]: 73/24

In [35]: c = Fraction(3,8)

In [36]: b + c
Out[36]: Fraction(73, 24)

In [37]: type(c)
Out[37]: fractions.Fraction

In [38]: type(a)
Out[38]: sympy.core.numbers.Rational
```

```
In [39]: type(b)
Out[39]: fractions.Fraction

In [40]: c = a + b

In [41]: c
Out[41]: 73/24

In [42]: type(c)
Out[42]: sympy.core.numbers.Rational

In [43]:
```

As you can see, the functionality of the module *fractions* provides the function *Fraction* which take as argument the numerator and the denominator that forms the fraction we want to wok with. Results of operations are returned in the same representation of *Fraction*, unless mixed with the *SymPy* way of defining fractions. As shown on output cells `Out[30]:`, `Out[34]:`, and `Out[41]:` the result returned ha been changed to the *SymPy* representation. In other words, when mixing *Fraction* type objects with the *SymPy* representation of fractions, the resulting object is a *SymPy Rational* object. This can be seen explicitly on output cell `Out[42]:`. We will not use the module *fractions* beyond this section.

5.4 Computing with fractions in *Python*

As already mentioned, we will stick with the way of representing fractions in *Python* via the *SymPy* module using the representation using the *S* function presented in the previous section. From your Prealgebra course work you know that the operations defined on integers are also defined on fractions, which are numbers represented as an indicated (not executed) division of two integers (i.e. $a/b = \frac{a}{b}$, $b \neq 0$) on which now the dividend a is called the numerator, while the divisor b is called the denominator. Neither the quotient and the remainder are computed as the operation of dividing the numbers is not executed. Nevertheless, when computing with fractions, we need to have a sense of the result. If the numerator is greater than the denominator, the result (quotient) will be greater than one. Otherwise, the result will be less than one. This knowledge can also be used when deciding about the result of multiplying or dividing by numbers greater or less than one. We let you use *Python* to reinforce what you have learned so far on this sort of numerical operations. Keep in mind that no matter what numbers you use, **dividend** $=$ **quotient** $\times$ **divisor** $+$ **remainder**.

As you have study in your Prealgebra course work, the basic operations of addition, subtraction, multiplication, division, and exponentiation with fractions has special rules to be followed to do them correctly. We will use the symbolic capabilities of *SymPy* to illustrate them. You might read the following sections with extra profit if you refresh on your mind these operations. If you still have not study them in your Prealgebra course work, don't worry. They will make sense.

Before continuing, let's comment the preamble of a typical *SymPy* session to perform symbolic simplification of expressions.

Chapter 5, IPython session 9

```
In [1]: from sympy import symbols

In [2]: from sympy import together, factor, collect, expand, simplify,
   expand

In [3]: collect?
Signature: collect(expr, syms, func=None, evaluate=None, exact=False,
   distribute
_order_term=True)
Docstring:
Collect additive terms of an expression.

This function collects additive terms of an expression with respect
to a list of expression up to powers with rational exponents.
...
...
...
In [4]:
```

On input cell `In [2]:` the *SymPy* functions *together*, *factor*, *collect*, *expand*, and *simplify* are made available into the current *Python* computational environment. You might read about them in the *SymPy* documentation, listed in the reference section at the end of the chapter (page 203) or reading the inline documentation by executing in any *IPython* input cell the instruction formed by the name of the function followed by the question mark symbol (i.e. `collect?`) as shown on input cell `In [3]:`.

A brief description of these (and a few other) *SymPy* functions is as follows:

- **together**: this function combines the given expression into a single fraction by denesting and combining rational subexpressions.
- **factor**: this function computes the factorization of the given expression.
- **collect**: this function collects additive terms of an expression with respect to a list of expression up to powers with rational exponents.

- **expand**: this function expands the given expression, expressing it as a sum of individual terms.
- **simplify**: this function via a set of sophisticated symbolic manipulation functions attempts to arrive at the simplest form of the given expression.
- **sympify**: this function Converts an arbitrary expression to a type that can be used inside *SymPy*. It is very useful to read any collection of symbols entered via the keyboard as strings which are them translated by **sympify** as valid *SymPy* expression.
- **nsimplify**: this function finds a simple representation for a number, converting (by default) Floats to Rationals.
- **S**: this function helps to define numerical fractions (rational numbers) in *SymPy*. For instance, to have 3/4 we could write any of the following: $S('3')/S('4')$, $S('3')/4$, or $3/S('4')$.
- **Eq**: this function is used to represent that two objects are equal. It is helpful in defining an equation, receiving as argument the left and right hand side of the equation.
- **symbols**: this function Transform an string into a *SymPy* symbol of the same name as the given string.
- **solveset**: this function Solves a given inequality or equation with a set as output.

We will use these functions to operate with fractions and to perform symbolic polynomial operations later in the book.

5.4.1 Addition of fractions

The following *IPython* session illustrates general symbolic computation with the addition of fractions:

Chapter 5, IPython session 10

```
In [1]: from sympy import symbols

In [2]: from sympy import together, factor, collect, expand, simplify,
   expand

In [3]: a, b, c, d = symbols ('a, b, c, d')

In [4]: u = a/b + c/d

In [5]: u
Out[5]: a/b + c/d

In [6]: u = together(u)

In [7]: u
```

```
Out[7]: (a*d + b*c)/(b*d)

In [8]:
```

On input cell `In [4]:` we entered the addition of two generic fractions a/b ($b \neq 0$) and c/d ($d \neq 0$). Output cell `Out[5]:` shows that *SymPy* does not automatically combine symbolic fractions (we will see shortly that it does so if numbers are used). To combine the symbolic fractions, in this case using the function *together* does the operation. From input cell `Out[5]:` and `Out[7]:` you might recognize the general result from your Prealgebra course work for adding fractions, summarized in the equation:

$$\frac{a}{b} + \frac{c}{d} = \frac{1}{bd}(ad + bc)\,\frac{ad + bc}{bd},\ b \neq 0 \,\text{and}\, d \neq 0. \tag{5.2}$$

The particular case when $b = d$ can be readily obtained from this relation as follows:

Chapter 5, IPython session 11

```
In [8]: u.subs(d,b) # in the u expression changes d by b
Out[8]: (a*b + b*c)/b**2

In [9]: simplify(Out[8])
Out[9]: (a + c)/b

In [10]:
```

On input cell `In [8]:` we applied the method *subs* to replace d by b in the existing expression of u. After doing the replacing, *SymPy* does not perform the obvious simplification unless it is instructed to do so via the *simplify* function on input cell `In [9]:`. Now you might recognize the result stated in your Prealgebra course work:

$$\frac{a}{b} + \frac{c}{b} = \frac{a + c}{b},\ b \neq 0. \tag{5.3}$$

Both results satisfy the rule that before fractions can be added, the fractions must have the same denominator. To add fractions with different denominators, we first rewrite the fractions as equivalent fractions with a common denominator, which is taken as the least common multiple (LCM) of the denominators of the fractions.

Let's now see some numerical examples (keep in mind that *SymPy* will always simplify your given numerical fractions to its simplest form):

Chapter 5, IPython session 12

```
In [10]: from sympy import S

In [11]: S('3')/6 + 7/6 + 5/6 # this way Python uses standard division
Out[11]: 2.50000000000000

In [12]: S('3')/6 + S('7')/6 + S('5')/6
Out[12]: 5/2

In [13]: S('3')/6 + S('7')/3 + S('5')/8
Out[13]: 83/24

In [14]: a = S('3')/6 ; b = S('7')/3 ; c = S('5')/8

In [15]: a + b + c
Out[15]: 83/24

In [16]:
```

We let you to study and understand the previous *IPython* session at your own peace. By the way, we can also use relational operators on fractions:

Chapter 5, IPython session 13

```
In [16]: a
Out[16]: 1/2

In [17]: b
Out[17]: 7/3

In [18]: c
Out[18]: 5/8

In [19]: a > c
Out[19]: False

In [20]: a == c
Out[20]: False

In [21]: b > a
Out[21]: True
```

```
In [22]: b > c
Out[22]: True

In [23]: S('2')/4 == S('8')/16
Out[23]: True

In [24]:
```

5.4.2 Subtraction of fractions

The following *IPython* session illustrates general symbolic computation with the subtraction of fractions:

Chapter 5, IPython session 14

```
In [1]: from sympy import symbols

In [2]: from sympy import together, factor, collect, expand, simplify,
   expand

In [3]: a, b, c, d = symbols ('a, b, c, d')

In [4]: u = a/b - c/d

In [5]: u
Out[5]: a/b - c/d

In [6]: u = together(u)

In [7]: u
Out[7]: (a*d - b*c)/(b*d)

In [8]:
```

On input cell `In [4]:` we entered the subtraction of two generic fractions a/b ($b \neq 0$) and c/d ($d \neq 0$). Output cell `Out[5]:` shows that *SymPy* does not automatically combine symbolic fractions (we will see shortly that it does so if numbers are used). To combine the symbolic fractions, in this case using the function *together* does the operation. From input cell `Out[5]:` and `Out[7]:` you might recognize the general result from your Prealgebra course work for the subtraction of fractions, summarized in the equation:

$$\frac{a}{b} - \frac{c}{d} = \frac{1}{bd}\,(ad - bc)\,\frac{ad - bc}{bd},\ b \neq 0\,\text{and}\,d \neq 0. \tag{5.4}$$

The particular case when $b = d$ can be readily obtained from this relation as follows:

Chapter 5, IPython session 15

```
In [8]: u.subs(d,b) # in the u expression changes d by b
Out[8]: (a*b - b*c)/b**2

In [9]: simplify(Out[8])
Out[9]: (a - c)/b

In [10]:
```

On input cell `In [8]:` we applied the method *subs* to replace d by b in the existing expression of u. After doing the replacing, *SymPy* does not perform the obvious simplification unless it is instructed to do so via the *simplify* function on input cell `In [9]:`. Now you might recognize the result stated in your Prealgebra course work:

$$\frac{a}{b} - \frac{c}{b} = \frac{a - c}{b},\ b \neq 0. \tag{5.5}$$

Both results satisfy the rule that to subtract fractions, the fractions must have the same denominator. To subtract fractions with different denominators, we first rewrite the fractions as equivalent fractions with a common denominator, which is taken as the least common multiple (LCM) of the denominators of the fractions.

Let's now see some numerical examples (keep in mind that *SymPy* will always simplify your given numerical fractions to its simplest form):

Chapter 5, IPython session 16

```
In [10]: from sympy import S

In [11]: S('3')/6 - S('5')/6
Out[11]: -1/3

In [12]: S('7')/3 - S('5')/6
Out[12]: 3/2
```

```
In [13]: - S('7')/3 - S('5')/6
Out[13]: -19/6

In [14]: a = -S('3')/6 ; b = S('7')/3 ; c = -S('5')/8

In [15]: a + b - c
Out[15]: 59/24

In [16]: S('5')/12 - S('3')/8
Out[16]: 1/24

In [17]:
```

We let you to study and understand the previous *IPython* session at your own peace.

5.4.3 Multiplication of fractions

The following *IPython* session illustrates general symbolic computation with the Multiplication of fractions:

Chapter 5, IPython session 17

```
In [1]: from sympy import symbols

In [2]: a, b, c, d = symbols ('a, b, c, d')

In [3]: u = a/b * c/d

In [4]: u
Out[4]: a*c/(b*d)

In [5]:
```

On input cell `In [3]:` we entered the multiplication of two generic fractions a/b ($b \neq 0$) and c/d ($d \neq 0$). On output cell `Out[4]:` you might recognize the general result from your Prealgebra course work for the multiplication of fractions, summarized in the equation:

$$\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd},\, b \neq 0 \,\text{and}\, d \neq 0. \tag{5.6}$$

The result satisfy the rule that to multiply fractions, multiply the numerators and multiply the denominators.

SymPy automatically recognize that a is the inverse of $1/a$:

Chapter 5, IPython session 18

```
In [5]: a*(1/a)
Out[5]: 1

In [6]: (1/a)*a
Out[6]: 1

In [7]:
```

Let's now see some numerical examples (keep in mind that *SymPy* will always simplify your given numerical fractions to its simplest form):

Chapter 5, IPython session 19

```
In [7]: from sympy import S

In [8]: S('3')/8 * S('16')/9
Out[8]: 2/3

In [9]: -S('3')/8 * S('16')/9
Out[9]: -2/3

In [10]: -S('3')/8 * -S('16')/9
Out[10]: 2/3

In [11]: -S('3')/8 * (-S('16')/9)*-S('3')/4
Out[11]: -1/2

In [12]: S('3')/8 * 16/9
Out[12]: 2/3

In [13]:
```

We let you to study and understand the previous *IPython* session at your own peace.

5.4.4 Division of fractions

The following *IPython* session illustrates general symbolic computation with the Division of fractions:

Chapter 5, IPython session 20

```
In [1]: from sympy import symbols

In [2]: a, b, c, d = symbols ('a, b, c, d')

In [3]: u = (a/b) / (c/d)

In [4]: u
Out[4]: a*d/(b*c)

In [5]:
```

On input cell `In [3]:` we entered the division of two generic fractions a/b ($b \neq 0$) and c/d ($d \neq 0$). On output cell `Out[4]:` you might recognize the general result from your Prealgebra course work for the division of fractions, summarized in the equation:

$$\left(\frac{a}{b}\right) / \left(\frac{c}{d}\right) = \frac{\frac{a}{b}}{\frac{c}{d}} = \frac{ad}{bc}, \, b \neq 0 \, c \neq 0 \, \text{and} \, d \neq 0. \tag{5.7}$$

The result satisfy the rule that to divide two fractions, multiply by the reciprocal of the divisor:

Chapter 5, IPython session 21

```
In [5]: ur = (a/b) * 1/(c/d)

In [6]: ur
Out[6]: a*d/(b*c)

In [7]: ur == u
Out[7]: True

In [8]:
```

Let's now see some numerical examples (keep in mind that *SymPy* will always simplify your given numerical fractions to its simplest form):

Chapter 5, IPython session 22

```
In [8]: from sympy import S

In [9]: ( S('3')/8 ) / ( S('16')/9 )
Out[9]: 27/128

In [10]: (-S('3')/8 ) / ( S('16')/9 )
Out[10]: -27/128

In [11]: (-S('3')/8) / (-S('16')/9)
Out[11]: 27/128

In [12]: (-S('3')/8) / 16
Out[12]: -3/128

In [13]: ( a/S('3') ) / ( b/S('9') )
Out[13]: 3*a/b

In [14]:
```

We let you to study and understand the previous *IPython* session at your own peace.

5.4.5 Exponential operations with fractions

When working with powers of numerical values, *SymPy* will do it correctly. When working with symbols, you need to be aware of the type of values the symbols could take.

Exponentiation rules of specific validity are as follows (this does not mean that these are the only cases on which these rules are true):

$$x^a x^b = x^{a+b} \quad \text{for all real } x, a, b \tag{5.8}$$

$$x^a y^a = (xy)^a \quad x, y \geq 0 \text{ and } a \text{ real.} \tag{5.9}$$

$$(x^a)^b = x^{a\,b} \quad \text{for all real } x, a, \text{ and } b \text{ any integer.} \tag{5.10}$$

Let's see how to get them in *SymPy*, which provides a special function *powsimp* to handle powers:

Chapter 5, IPython session 23

```
In [1]: from sympy import symbols, powsimp

In [2]: x, y = symbols('x, y', positive=True)

In [3]: a, b = symbols('a, b', real=True)

In [4]: u = x**a*x**b

In [5]: u
Out[5]: x**a*x**b

In [6]: powsimp(u)
Out[6]: x**(a + b)

In [7]: u = x**a*y**a

In [8]: u
Out[8]: x**a*y**a

In [9]: powsimp(u)
Out[9]: (x*y)**a

In [10]: u = (x**a)**b

In [11]: u
Out[11]: x**(a*b)

In [12]:
```

These lines of code explains what they do on their own. We let you to follow them on your own. The *SymPy* tutorial contains a thorough discussion of the function *powsimp*. Let's now turn to exponentiation in fractions:

Chapter 5, IPython session 24

```
In [1]: from sympy import symbols, powsimp, expand

In [2]: x, y = symbols('x, y', positive=True)

In [3]: a, b, c, d = symbols('a, b, c, d', real=True)
In [4]: u = (x/y)**a

In [5]: u
Out[5]: (x/y)**a

In [6]: powsimp(u)
Out[6]: (x/y)**a

In [7]: expand(u)
Out[7]: x**a*y**(-a)

In [8]: u = (x/y)**a * (x/y)**b

In [9]: u
Out[9]: (x/y)**a*(x/y)**b

In [10]: powsimp(u)
Out[10]: (x/y)**(a + b)

In [11]: u = (x/y)**a * x**b

In [12]: u
Out[12]: x**b*(x/y)**a

In [13]: powsimp( expand(u) )
Out[13]: x**(a + b)*y**(-a)

In [14]:
```

We let you to continue exploring how *SymPy* handle other symbolic expressions. In particular, try using numerical values in the exponent. Let's turn now to numerical fractions:

Chapter 5, IPython session 25

```
In [1]: from sympy import S

In [2]: u = S('3')/4

In [3]: u
Out[3]: 3/4
```

```
In [4]: v = S('5')/7

In [5]: u**2*v**3
Out[5]: 1125/5488

In [6]: (u/v)**2
Out[6]: 441/400

In [7]: (u/v)**2*((u/v)**(-2))
Out[7]: 1

In [8]: (u/v)**2*((v/u)**2)
Out[8]: 1

In [9]:
```

We encourage you to try the examples of your course work to become more familiar on how *SymPy* performs exponentiation. For further discussion of this subject read the tutorial listed on the reference section (page 203).

5.4.6 Solving equations involving fractions via *SymPy*

As we did with whole numbers 3.9 (page 119) and integers 5.2.3 (page 176), we can also use *SymPy* to solve equations containing fractions and (in case a non-integer solution is obtained) get the solution expressed as a fraction. After reading this section, you will find instructive to re-read sections about equations with whole numbers (page 119) and integers (page 176), and read the section about equations with decimals (page 230). You could also practice your skills in solving equations via the one variable equation solver described on page 236

Let's show it by finding the solution of:

$$\frac{7}{2} = 5x + 9 \tag{5.11}$$

Surely your Prealgebra textbook has plenty of similar equations stated just like that. In the previous sections about equations, we have stressed the fact that whenever you try to solve an equation, you need to spent sometime thinking about what that equation could represent. In section 5.2.3 (page 176) about solving equations with integers we gave some ideas on how you can rephrase the wording presenting equation 5.1 (page 195) to make it look as if it were the result of a word problem. You were able to see that the same equation could be assigned to more than one word problem representing different situations. What makes unique the problems is the meaning of the equations in each respective context: perhaps negative solutions are not allowed in a particular case; perhaps the point of interest in another case is in the set of numbers greater than the one that solve the equation. The are plenty of possibilities. You named it!. The point is that one need to look at equations beyond the usual "find the solution of ⋯" and think of the equations in your Prealgebra course works with those that you see in your Physics, Biology, Chemistry, subjects. Go beyond to simple learning the methods to solve it. As excite as interesting this discussion might be, we need to continue with our subject of finding the solution of the posed equation. You are encourage to check the references at the end of Chapter 3, starting on page 144.

Just keep in mind that *SymPy* will hide from you how it finds the solution of the equation. Consequently, you are encourage to workout the exercise following by hand (using pencil and paper) the procedures for solving equations presented in your Prealgebra course work. This way you will develop your intuition on whether the solution found make or not sense and how to check it.

A piece of code solving this equation (which you can find under the name `chap05_prog_02_Sympy_SolEquation.py` in the directory named `chapter_05` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface of this book) is as follows:

```python
from sympy import symbols, Eq, solveset, S
x = symbols('x')

LHS = S('7')/2
RHS = 5*x + 9

thesol = list( solveset( Eq(LHS, RHS), x) )
print('thesol =', thesol)

newLHS = LHS - RHS #rearrange the equation to read: LHS - RHS = 0
print('newLHS =', newLHS)

newLHS = newLHS.subs(x, thesol[0])
if newLHS.simplify() == 0:
   print('The solution of {0} = {1}, is x =
      {2}'.format(LHS,RHS,thesol[0]))
```

After executing these lines of code you'll get:

Chapter 5, System shell command 2

```
$ python chap05_prog_02_Sympy_SolEquation.py
thesol = [-38/25]
```

```
newLHS = -5*x - 38/5
The solution of 7/5 = 5*x + 9, is x = -38/25
```

If you have trouble understanding this code, please go back and read section 3.9, starting on page 119.

Notice that the trick to get the solution expressed as a rational number is to, at least, write one of the numbers in the equation (no matter which one) using the *SymPy* *S* function. Nevertheless, to avoid obtaining weird results because of unexpected changes in the future from the *SymPy* developers (as those make by the *Python* developers every time they release a major *Python* version, breaking stright forward portability across *Python* version), you can write all of the constants in your equation using the S function.

At this point, you have know the knoeledge of fractions which can help you start trying solutions to more complicated equations, whose solution are more general than integers. You can write the numbers in the equation as fractions in order to find the solution expressed in terms of a fraction. We hope you enjoy the the end of this trip finding solutions of end of chapter exercises in your Prealgebra textbook.

To help you in this endeavour, we are including a code for solving any one variable equation (which you can find under the name `chap05_prog_03_Sympy_SolInputEquation.py`, in the directory named `chapter_05` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface of this book).

We let you read and understand the code at your own peace, the listing of which is shown in figure 5.1, on page 199. In the next chapter we will study how to represent real values in *Python*, which will explain the use of line of code 57.

Here is a sample of executing it:

Chapter 5, System shell command 3

```
$ python chap05_prog_03_Sympy_SolInputEquation.py
Enter the variable name (i.e. x, y, z): x
 *=========
 * Input the part of your equations as requested
 * If your equation looks like: (3/2)x - 8 = 7x + 5
 * LHS = (3/2)x - 8, and you should enter for it: (3/2)*x - S('8')
 * RHS =   7x + 5, and you should enter for it: 7*x + S('5')
 * If you write your equation in a text editor, copy and paste might
    work.
 *=========
Enter the LHS of the equation: (3/2)*x - S('8')
   You entered LHS = (3/2)*x - S('8')
```

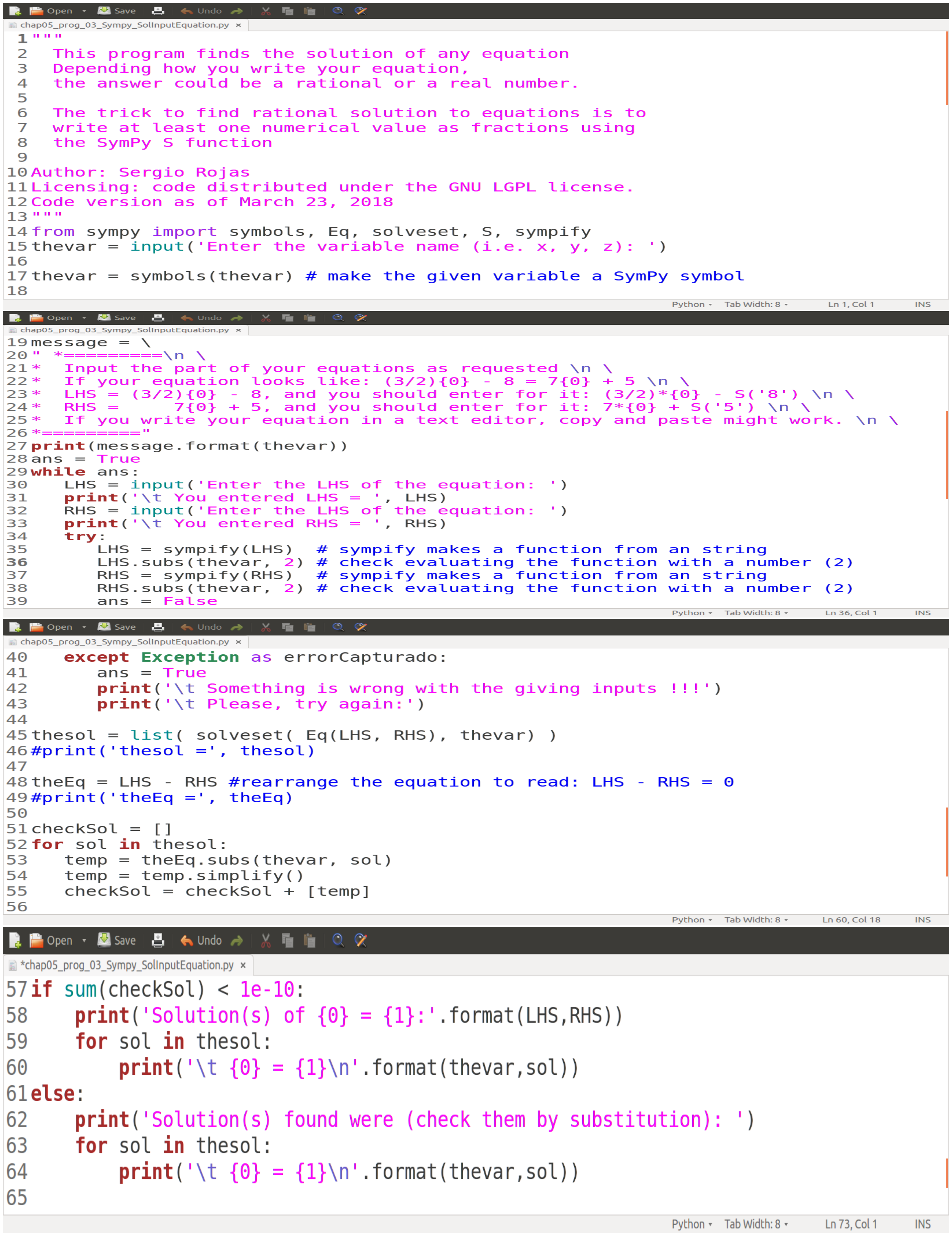
```
Enter the LHS of the equation: 7*x + S('5')
   You entered RHS = 7*x + S('5')
Solution(s) of 3*x/2 - 8 = 7*x + 5:
   x = -26/11
```

We find it instructive (see also exercise 5.6, on page 201) to show a sample running the code using an equation containing real values on the entered equation (this explains the use of line of code 57):

Chapter 5, System shell command 4

```
$ python chap05_prog_03_Sympy_SolInputEquation.py
Enter the variable name (i.e. x, y, z): z
 *=========
 * Input the part of your equations as requested
 * If your equation looks like: (3/2)z - 8 = 7z + 5
 * LHS = (3/2)z - 8, and you should enter for it: (3/2)*z - S('8')
 * RHS =   7z + 5, and you should enter for it: 7*z + S('5')
 * If you write your equation in a text editor, copy and paste might
   work.
 *=========
Enter the LHS of the equation: 5.0*z**3 - 4.0*z + 8.0
   You entered LHS = 5.0*z**3 - 4.0*z + 8.0
Enter the LHS of the equation: 0
   You entered RHS = 0
Solution(s) of 5.0*z**3 - 4.0*z + 8.0 = 0:
   z = -1.39525701255756

   z = 0.697628506278782 - 0.812438673573761*I

   z = 0.697628506278782 + 0.812438673573761*I
```

Let's end this section mentioning that, in general, when finding numerical solution to equations, we might need to use other *Python* alternatives better suitable than *SymPy*, like *SciPy*.



```python
"""
  This program finds the solution of any equation
  Depending how you write your equation,
  the answer could be a rational or a real number.

  The trick to find rational solution to equations is to
  write at least one numerical value as fractions using
  the SymPy S function

Author: Sergio Rojas
Licensing: code distributed under the GNU LGPL license.
Code version as of March 23, 2018
"""
from sympy import symbols, Eq, solveset, S, sympify
thevar = input('Enter the variable name (i.e. x, y, z): ')

thevar = symbols(thevar) # make the given variable a SymPy symbol

message = \
" *==========\n \
*  Input the part of your equations as requested \n \
*  If your equation looks like: (3/2){0} - 8 = 7{0} + 5 \n \
*  LHS = (3/2){0} - 8, and you should enter for it: (3/2)*{0} - S('8') \n \
*  RHS =      7{0} + 5, and you should enter for it: 7*{0} + S('5') \n \
*  If you write your equation in a text editor, copy and paste might work. \n \
*=========="
print(message.format(thevar))
ans = True
while ans:
    LHS = input('Enter the LHS of the equation: ')
    print('\t You entered LHS = ', LHS)
    RHS = input('Enter the LHS of the equation: ')
    print('\t You entered RHS = ', RHS)
    try:
        LHS = sympify(LHS)   # sympify makes a function from an string
        LHS.subs(thevar, 2)  # check evaluating the function with a number (2)
        RHS = sympify(RHS)   # sympify makes a function from an string
        RHS.subs(thevar, 2) # check evaluating the function with a number (2)
        ans = False
    except Exception as errorCapturado:
        ans = True
        print('\t Something is wrong with the giving inputs !!!')
        print('\t Please, try again:')

thesol = list( solveset( Eq(LHS, RHS), thevar) )
#print('thesol =', thesol)

theEq = LHS - RHS #rearrange the equation to read: LHS - RHS = 0
#print('theEq =', theEq)

checkSol = []
for sol in thesol:
    temp = theEq.subs(thevar, sol)
    temp = temp.simplify()
    checkSol = checkSol + [temp]

if sum(checkSol) < 1e-10:
    print('Solution(s) of {0} = {1}:'.format(LHS,RHS))
    for sol in thesol:
        print('\t {0} = {1}\n'.format(thevar,sol))
else:
    print('Solution(s) found were (check them by substitution): ')
    for sol in thesol:
        print('\t {0} = {1}\n'.format(thevar,sol))

```

Figure 5.1: Program to find solutions to any one variable equation via *SymPy*.

5.5 Chapter Summary

Ending this chapter you have done great! By know you know how to handle computational operations of your Prealgebra involving integers and rational numbers. We learned about the *S* function from *SymPy* (and about the *Fraction* from the module *fractions*) to properly represent fractions in *Python*. You also did some algebraic symbolic operations (as well as numerical ones) with equations involving integers and rational numbers, ending the chapter with a general program to find numerical solutions to any one variable equation.

In the next chapter we will learn about the fundamental aspects of representing real numbers in *Python* and how to perform computations with them.

Exercises of Chapter 5

Exercise 5.1 *Find the* mean, median, *and* mode *of the following set of values:*

```
TheValues = [-70, 50, -100, 0, 70, 50, -30, 8, -50, 90, 110, 40, -50]
```

Exercise 5.2 *Knowing that the solution is an integer, write a program to find the solution of the the equation 5.1 (page 176) without using* SymPy*. You need o write a recipe (algorithm), implement, and test it (for that, one way to go is thinking in defining and interval on which the solution can be found and in the bisection method to search for it).*

Exercise 5.3 *Write an algorithm to find the mixed number representation of an improper fraction. Implement the algorithm in* Python *and write a few test cases.*

Exercise 5.4 *Write an algorithm to find the improper fraction representation of a mixed number. Implement the algorithm in* Python *and write a few test cases.*

Exercise 5.5 *Write an algorithm to find the least common factor (LCD) of two or more natural numbers (you could do it by listing the multiples of the numbers). Implement the algorithm in* Python *and write a few test cases. (you might one to reread our algorithm to find the greatest common divisor of a set of whole numbers discussed on section 3.8.5, page 114).*

Exercise 5.6 *Execute the code of figure 5.1, page 199 to find the solution of* $5z^3 - 4z + 8 = 0$ *by entering (after z as variable) one of the following in each run:*

```
LHS = 5*z**3 - 4*z + 8
LHS = 5*z**3 - 4*z + S('8')
LHS = 5*z**3 - 4*z + 8.0
RHS = 0
```

Explain what you notice (including the executing time).

Exercise 5.7 *Improve the code of figure 5.1, page 199 to properly handle wrong input of the variable.*

References of Chapter 5

Books and/or Articles

- **Marecek, L. and Smith, M. A.** (2017). Prealgebra, Rice University, OpenStax `https://openstax.org`.
  Book available for free at: `http://cnx.org/content/col11756/1.9`
-

References on the WEB

- ***Python* tutorial**:
  `https://docs.python.org/3/tutorial/index.html`
- **SymPy tutorial**:
  `http://docs.sympy.org/latest/tutorial/index.html`

6

Decimal numbers in *Python*

"God made the natural numbers; all the rest is the work of man."
Leopold Kronecker

6.1 Introductory remarks

We have study integers (which includes whole numbers) and fraction representation of numbers in *Python*. We learned that *Python* deals directly with integers using infinity precision, meaning that we can carry out precise computations with integer values (no matter how big they could be) in *Python*. Integers are represented as objects of type *int* in *Python*. We also learned that to deal with fractions (as an indicated division between two integers) we need to use the special function *S* from the module *SymPy* (or the function *Fraction* from the module *fractions*). Via either functions (though we will stick with the *S* function), we can perform infinity precision computations with fractions. For *SymPy*, fractions are represented as objects of type *Rational*.

We know turns to the study of real numbers and its operations in *python*. This will allow us to wider our use of *Python* instructions to write more complex programs as we can know execute computations mixing the different type of numbers we are familiar with. As learned in your Prealgebra course work, just keep in mind that some computations have sense in a particular numerical system. For instance, when forming study groups in your classroom, each group must have a positive integer as the number of individuals forming the group. Each group is a fraction of the total class size, which in turn is a positive integer.

In reading this chapter, you are suppose to have read and (surely you had) understood the preceding chapters as we will using in here basically all of the *Python* instructions we introduced in these preceding chapters. We will also include a few new *Python* built in functions as we need them to deal with the real numbers.

To be specific, in this chapter we will give a lengthy discussion in working with real numbers in *Python*. So, take your shoes off and open your mind to see some unusual behavior when computing using real number representations in any computer.

After finishing this chapter, you'll be equipped with a almost complete functional set of *Python* tools that will allow you explore computations in a much wider sense. Graphing of the results will be left to the next chapter.

We will not rest in repeating that you should always keep in mind that a major goal of your investment reading this book is (as an independent learner) to enhance your skills in applying what you have learned to tackle new first (unfamiliar) met situations (not only in Prealgebra course work but also in any other of the subjects in your educational track like Physics, Chemistry, Biology, Sociology, Psychology, and so on)). The successful acquisition of such desired outcome requires your engagement in an effective learning involvement of constantly applying the programming stages of *designing* (actions), *implementing* (the actions in the required order), and *assessing* (the performance of such actions) to any other real world situations.

6.2 Computing with decimal numbers in *Python*

As learned in your Prealgebra course work, decimal numbers are written following the notation of a **signed integer** followed by a **dot** followed by a **whole number** (representing the decimal part) like 23.98, 5.2345, and 345.98765001. Some usual representations for decimals uses the power of ten notation. You might be familiar with the *scientific notation*, in which decimals are represented in the form $d_1.d_2d_3d_4\cdots d_kd_{k+1}\cdots \times 10^m$, where $d_1 \neq 0$ and $0 \leq d_i \leq 9$. Example are 1.08, -1.05, $1.\bar{3}$, etc. You certainly have been using such representations when performing computations in your calculator, becoming familiar with the fact that in the computer, however, decimals are represented using only a finite number of digits, given rise to the so called *finite digit arithmetic*, whose consequences (perhaps already known by you) will be mentioned shortly.

Decimal numbers are represented in *Python* as *float* objects (from the *floating number* representation) which, in essential, is (contrary to integers and fractions) an **approximated** representation of these numbers in the computer. This comes from the fact that (generally) computers uses the binary (or base two) system to represent numbers. For decimals, this internal representation is organized such that the given decimal number is converted into binary where (in general) the fractional part is truncated to a fixed length in bits following the IEEE 754 binary floating point standard representation. In *Python* this standard defaults to the 64 bit representation. This means that every number must fit into 64 binary digits of storage, meaning that there can not be more than $2^{64} \approx 10^19$ distinct decimal numbers represented in the computer in the form **sign**$\times$mantissa$\times 2^{\text{exponent}}$, where*mantissa* takes 52 bits of precision and the exponent (also called the characteristic) 11 bits. An in depth discussion on how numbers are represented in the computer will take us beyond the Prealgebra topics. Please, check out the references at the end of the chapter (page) for extra details in this fascinating subject.

In practical operational terms, the 64 bits representation allows upto 15 digits of precision to represent a minimal absolute value of about $2^{-1023} \approx 10^{-308}$ and the maximum value of about $2^{+1023} \approx 10^{+308}$. The exact values for your system can be obtained executing the commands:

Chapter 6, IPython session 1

```
In [1]: import sys
In [2]: sys.float_info.dig
Out[2]: 15

In [3]: sys.float_info.min
Out[3]: 2.2250738585072014e-308

In [4]: sys.float_info.max
Out[4]: 1.7976931348623157e+308

In [5]: sys.float_info.epsilon # is the smallest 52-bits mantissa 2.0**(-52)
Out[5]: 2.220446049250313e-16

In [6]:
```

The variable *sys.float_info* contains more information about the internal representation of the *Python* object *float* on your machine If you want the full listing contained in the variable *sys.float_info*, just execute the following lines of code:

```
import sys
u = str(sys.float_info).split(')')[0].split('(')[1].split(',')
for val in u:
    print(val)
```

The fact that only a subset of the decimal numbers can be represented in the computer put a strong limitation on the computations we can do using numbers outside the range range. But we have even more restrictions when computing with the allowed decimals as the computer can not represent them with infinity precision. Nevertheless, with such limitations, the computer helps to send rockets to the space, to have satellites for communications, to have video games, and so many other things around our technological world. The main point here is that when using the computer, we need to be aware of its limitations in terms of computing with decimals. A few of which are presented below, in section 6.2.3, page 211.

6.2.1 Operations with decimals

As in the previous cases of integers and fractions, the basic operations of addition, subtraction, multiplication, division, and exponentiation with decimals are implemented in *Python* via the operators ($+$, $-$, $*$, $/$, and $**$) we are familiar with from section 2.3, page 24. The following *IPython* session illustrates them:

Chapter 6, IPython session 2

```
In [1]: a = -2.208

In [2]: b = 29.786

In [3]: c = - 5.675432

In [4]: a + b + c
Out[4]: 21.902568000000002

In [5]: a*b + c
Out[5]: -71.44292000000002

In [6]: a/b
Out[6]: -0.07412878533539247

In [7]: (c*b)/a
Out[7]: 76.5617833115942

In [8]: a**3
Out[8]: -10.764582912000003

In [9]: a**2
Out[9]: 4.8752640000000005

In [10]: a**c
Out[10]: (0.005843891513628068+0.009507377106479155j)

In [11]: abs(a*b + c)
Out[11]: 71.44292000000002

In [12]: (2**(0.5))**2
Out[12]: 2.0000000000000004

In [13]: (2.**2)**(0.5)
Out[13]: 2.0

In [14]: (1.0/49.0)*49.0
Out[14]: 0.9999999999999999

In [15]:
```

Notice that some operations does not returns the expected result as in output cells `Out[12]:` and `Out[14]:`. These kind of errors are consequence of the finite number arithmetic that any computer uses to operate with decimals, and are called *round-off errors*. When operating with decimals, you need to be aware of this sort of errors.

Before discussing some nuances of working with the finite number arithmetic of the computer, in the next section we will introduce a way to quantify its approximation errors.

6.2.2 Relative error: a way to quantify round-off errors (optional)

As mentioned, the finite representation of decimal numbers in the computer introduces round-off errors which, eventually, we would like to quantify. A commonly used measure is called the *relative error* defined as follows: let N^{exact} be the exact number which is represented in the computer by the number N^{approx} containing less digits than N^{exact}, then the {emphrelative error (RE) is defined as:

$$RE = \left| \frac{N^{\text{exact}} - N^{\text{approx}}}{N^{\text{exact}}} \right| , \tag{6.1}$$

where he vertical bars means absolute value (the RE is always a positive number).

As illustrative example, let's consider that we have the number $N^{\text{exact}} = 5/3$, which, via *chopping* or *truncation*, could be represented to three decimal digits $N^{\text{approx}} = 1.666$. The same number could also be approximated, via *rounding*, to three decimal digits $N^{\text{approx}} = 1.667$. The RE for each approximation is:

$$RE_{\text{truncating}} = \left| \frac{\frac{5}{3} - \frac{1666}{1000}}{\frac{5}{3}} \right| = \frac{1}{2500} = 0.0004 \tag{6.2}$$

$$RE_{\text{rounding}} = \left| \frac{\frac{5}{3} - \frac{1667}{1000}}{\frac{5}{3}} \right| = \left| -\frac{1}{5000} \right| = 0.0002 \tag{6.3}$$

We see that rounding (in this case) gives a smaller error that truncating. To have a better sense of what this relative error means, we can compute the percentage of the actual number that this error represents. This quantity is computed in the form $(RE/N^{\text{exact}}) \times 100$.

In the case of $RE_{\text{truncating}}$, this error is 0.024%. You are left as an exercise to compute this quantity for RE_{rounding}.

These approximations could have been computed using *SymPy* as follows (in these computations the only new operations are the way how we get the rounded and truncated approximations of 5/3 in, respectively, input cells `In [6]:` and `In [12]:`, and the use of the *SymPy* function *nsimplify* to convert a *Python float* object into a *SymPy Rational* object):

Chapter 6, IPython session 3

```
In [1]: from sympy import S, N, nsimplify

In [2]: a = S(5)/3

In [3]: a
Out[3]: 5/3

In [4]: type(a)
Out[4]: sympy.core.numbers.Rational

In [5]: N(a)
Out[5]: 1.66666666666667

In [6]: a_approx_round = float(str(N(a,4)))

In [7]: a_approx_round
Out[7]: 1.667

In [8]: type(a_approx_round)
Out[8]: float

In [9]: a_approx_round = nsimplify( a_approx_round )

In [10]: a_approx_round
Out[10]: 1667/1000

In [11]: type(a_approx_round)
Out[11]: sympy.core.numbers.Rational

In [12]: a_approx_trunc = float(str(N(a))[0:5])

In [13]: a_approx_trunc
Out[13]: 1.666

In [14]: type(a_approx_trunc)
Out[14]: float

In [15]: a_approx_trunc = nsimplify( a_approx_trunc )

In [16]: a_approx_trunc
Out[16]: 833/500

In [17]: type(a_approx_trunc)
```

```
Out[17]: sympy.core.numbers.Rational

In [18]: REtruncating = abs( (a - a_approx_trunc)/a )

In [19]: REtruncating
Out[19]: 1/2500

In [20]: N(REtruncating)
Out[20]: 0.000400000000000000

In [21]: N(REtruncating,1)
Out[21]: 0.0004

In [22]: RErounding = abs( (a - a_approx_round)/a )

In [23]: RErounding
Out[23]: 1/5000

In [24]: N(RErounding)
Out[24]: 0.000200000000000000

In [25]: N(RErounding,1)
Out[25]: 0.0002

In [26]:
In [26]: TruncationPercentError = (REtruncating/a)*100

In [27]: TruncationPercentError
Out[27]: 3/125

In [28]: N(TruncationPercentError)
Out[28]: 0.0240000000000000

In [29]: N(TruncationPercentError,2)
Out[29]: 0.024

In [30]:
```

This discussion is useful for situations where we know what the output would be. This is the case of test values used to verify that our program is computing correctly. In those cases we can also check the relative error for known situations on which the result is not exact. When performing numerical computations, you will use equation 6.1 to compute an overall estimation error of your computations relative to the average value of the computed quantity. Similar use will be given when analyzing laboratory measurements from your Physics, Chemistry, or Biology courses in your educational track.

6.2.3 Cautionary tales about operations with decimals

How, you might be wondering, the computer finite digit arithmetic actually affect our computations.? Let's consider we are given to compute the quantity $((A+B)^2 - 2AB - B^2)/A^2$ with $A = 0.0254$ and $B = 9788.0$. You can readily do it using an ipython console:

Chapter 6, IPython session 4

```
In [1]: A = 0.0254

In [2]: B = 9788.0

In [3]: ApproxValue_1 = ( (A+B)**2 - 2*A*B - B**2)/A**2

In [4]: ApproxValue_1
Out[4]: 1.0000241431738204

In [5]:
```

But your project classmate (working late night too) does the computation switching the assigned values to A and B, in the form:

Chapter 6, IPython session 5

```
In [10]: A = 9788.0

In [11]: B = 0.0254

In [12]: ApproxValue_2 = ( (A+B)**2 - 2*A*B - B**2)/A**2

In [13]: ApproxValue_2
Out[13]: 1.0000000000000002

In [14]:
```

In the morning when discussing the obtained results from the seemingly trivial exercise to be presented in class, which value would you choose to be reported? The one on output cell

`Out[4]:` or the one on output cell `Out[13]:`? What would you do with the other result? It is wise to hide it?

In this case, it turns out that the value of 1 (in cell `Out[13]:`) is the right value in this operation. In fact, with a little algebra you can easily verify that

$$\frac{(A+B)^2 - 2AB - B^2}{A^2} = 1. \tag{6.4}$$

An after the fact computation of the relative error shows that the second computation is much closer to the correct result than the first computation, as you can see in what follows:

Chapter 6, IPython session 6

```
In [5]: ExactValue = 1.0

In [6]: RelError_1 = abs( (ExactValue - ApproxValue_1)/ExactValue )

In [7]: RelError_1
Out[7]: 2.4143173820379005e-05

In [8]: PercentError_1 = (RelError_1/ExactValue)*100

In [9]: PercentError_1
Out[9]: 0.0024143173820379005

In [15]: RelError_2 = abs( (ExactValue - ApproxValue_2)/ExactValue )

In [16]: RelError_2
Out[16]: 2.220446049250313e-16

In [17]: PercentError_2 = (RelError_2/ExactValue)*100

In [18]: PercentError_2
Out[18]: 2.220446049250313e-14

In [19]:
```

What went wrong, you might be wondering, with the first result of output cell `Out[4]:`? It turns out that the trouble here comes from the subtlety of subtracting (in the numerator) two very similar numbers and then dividing the result by the small value of A^2, as can be seen inspecting the magnitude of each term:

Chapter 6, IPython session 7

```
In [19]: A = 0.0254

In [20]: B = 9788.0

In [21]: (A+B)**2
Out[21]: 95805441.23104517

In [22]: 2*A*B + B**2
Out[22]: 95805441.2304

In [23]: A**2
Out[23]: 0.00064516

In [24]: (A+B)**2 - (2*A*B + B**2)
Out[24]: 0.0006451755762100022

In [25]:
```

In the second case this situation does not happen, as can also be checked out inspecting the magnitude of each term:

Chapter 6, IPython session 8

```
In [25]: A = 9788.0

In [26]: B = 0.0254

In [27]: (A+B)**2
Out[27]: 95805441.23104517

In [28]: 2*A*B + B**2
Out[28]: 497.23104515999995

In [29]: A**2
Out[29]: 95804944.0

In [30]: (A+B)**2 - (2*A*B + B**2)
Out[30]: 95804944.00000001

In [31]:
```

This issue is not difficult to happen in practice, as we can easily (not being aware of the mentioned subtlety) program the left hand side of equation 6.4 as a two parameter function and then use it to make additional computations.

The general issue here is that any error of a finite digit representation gets enlarged by dividing by a number with small magnitude or via multiplication by a number with large magnitude.

This last issue is shown below (compare output cells `Out[10]:` and `Out[14]:`):

Chapter 6, IPython session 9

```
In [8]: A = 9788.0

In [9]: B = 0.0254

In [10]: ((A+B)**2 - 2*A*B - B**2)*B**2 # Expanding will give (A*B)**2
Out[10]: 61809.517671040005

In [11]: (A*B)*(A*B)
Out[11]: 61809.51767103999

In [12]: A = 0.0254

In [13]: B = 9788.0

In [14]: ((A+B)**2 - 2*A*B - B**2)*B**2 # Expanding will give (A*B)**2
Out[14]: 61811.00994896889

In [15]: (A*B)*(A*B)
Out[15]: 61809.51767103999

In [16]:
```

It is let as an exercise to compute the errors in these computations (exercise 6.1, page 241).

The limitations of the computer in representing decimals also impose some restrictions in the addition of decimals. From your Prealgebra course work you know that for any number a, it is always true that $a + 1 = 1$ only and only if $a = 0$.

Since the computer can not represent arbitrarily small decimal numbers, then the machine epsilon is defined as the largest number ϵ (represented in the computer) such that $1.0 + \epsilon = 1.0$ In other words the values less or equal to ϵ are considered zeros when added to one. Let's see it in action:

Chapter 6, IPython session 10

```
In [18]: import sys

In [19]: e = sys.float_info.epsilon #the machine epsilon

In [20]: e
Out[20]: 2.220446049250313e-16

In [21]: e == 2.**(-52)
Out[21]: True

In [22]: 1e-22 < e
Out[22]: True

In [23]: 1e-22 + 1.0 == 1.0
Out[23]: True

In [24]: 1e-16 < e
Out[24]: True

In [25]: 1e-16 + 1.0 == 1.0
Out[25]: True

In [26]: e < e
Out[26]: False

In [27]: e + 1.0 == 1.0
Out[27]: False

In [28]: 1e-15 < e
Out[28]: False

In [29]: 1e-15 + 1.0 == 1.0
Out[29]: False
```

```
In [30]:
```

Another true operation from your Prealgebra course work is that $a+1 \neq a$ (or that $a+1 > a$). But due to limitations in the representation of decimals, there are many numbers for which $a+1 = a$. Here is a code to find the minimal value for which that happen:

```
a = 1.0
i = 0
while (a + 1.0) != a:
   i = i+1
   a = a*2.0
print("for i >= {}, it happens that (2.0)**i + 1 = (2.0)**i".format(i))
```

Executing this code you will find that $i = 53$, for which we have:

Chapter 6, IPython session 11

```
In [27]: (2.0)**52 + 1.0 == (2.0)**52
Out[27]: False

In [28]: (2.0)**53 + 1.0 == (2.0)**53
Out[28]: True

In [29]: (2.0)**54 + 1.0 == (2.0)**54
Out[29]: True

In [30]:
```

Since $2^53 \approx 10^16$, using powers of ten with exponents greater than or equal to 16 gives similar results as above:

Chapter 6, IPython session 12

```
In [70]: 10.**14 + 1.0 == 10.**14
Out[70]: False

In [71]: 10.**15 + 1.0 == 10.**15
Out[71]: False
```

```
In [72]: 10.**16 + 1.0 == 10.**16
Out[72]: True

In [73]: 10.**17 + 1.0 == 10.**17
Out[73]: True

In [74]:
```

You can recall that, back in chapter 2, page 31, we were dealing with the wheat problem involving the computation of a big whole number. Back then we were able to perform the computation thanks to the infinite precision of *Python* to represent integers. The number we are talking about is $2^{64}-1$, which as an integer is readily computed in *Python*:

Chapter 6, IPython session 13

```
In [31]: 2**64 - 1
Out[31]: 18446744073709551615

In [32]:
```

If trying to do this computation using standard floating point computation, we will be in trouble (the last four digits to the right are lost):

Chapter 6, IPython session 14

```
In [32]: 2.0**64 - 1.0
Out[32]: 1.8446744073709552e+19

In [33]:
```

You can certainly explain this result based on what we just learned. In the next section we will come back to try perform correctly this computation using decimals (as we already know how to do it correctly using integers).

Another cautionary tale (derived in some cases from the previous one) involves the violation (in computer arithmetic with decimals) of the associative law of addition $(a+b)+c = a+(b+c)$.

This violation happens when one of the digits is very small compared with the others. Let's see it:

Chapter 6, IPython session 15

```
In [76]: a = 10.**16; b = -10.**16; c = 1.

In [77]: (a+b)+c == a+(b+c)
   ...:
Out[77]: False

In [78]: a = 10.**16; b = -10.**16; c = 2.

In [79]: (a+b)+c == a+(b+c)

Out[79]: True

In [80]: a = 1234.567 ; b = 45.67834 ; c = 0.0004

In [81]: (a +b)+ c
Out[81]: 1280.2457399999998

In [82]: a +(b +c)
Out[82]: 1280.24574

In [83]:
```

We also needs to be careful when performing computations with very small numbers, as any one that results in underflow are set to zero (as we already know). What follows are some operations with small numbers:

Chapter 6, IPython session 16

```
In [27]: 1e-323 + 1e-323 # OK
Out[27]: 2e-323

In [28]: 1e-324 + 1e-324 # round off error occur beyond exponent negative 323
Out[28]: 0.0

In [29]: 1e-162*1e-161 # OK
Out[29]: 1e-323
```

```
In [30]: 1e-162*1e-162 # round off error occur
Out[30]: 0.0

In [31]:
```

Computing with large numbers is also problematic. Any number that results in overflow are set to the special value *inf* and the program might not stop. Here are some operations with large numbers:

Chapter 6, IPython session 17

```
In [31]: 1.0*1e+308 # OK
Out[31]: 1e+308

In [32]: 1.0*1e+309 # overflow occur beyond exponent positive 308
Out[32]: inf

In [33]: 1e+154*1e+154 # OK
Out[33]: 1e+308

In [34]: 1e+154*1e+155 # overflow
Out[34]: inf

In [35]: 1e-309*1e309
Out[35]: inf

In [36]: 1e-309*1e+309
Out[36]: inf

In [37]: 1e-309*1e309 + 1e-309*1e309
Out[37]: inf

In [38]:
```

As mentioned, in *Python* the special representation *inf* is assigned to overflow results. Another special symbol (*nan*) is assigned to undefined results, standing for not-a-number, meaning an undefined mathematical operation. These symbols are used so when an overflow or undefined operation happen in a computation the program do not stop. In case we want to check if any of them has been given as output, we can use special functions from the *SciPy* module *isnan* and *isinf*. The following line of code illustrates this:

Chapter 6, IPython session 18

```
In [47]: a = [1e-309, 1e309, 2.3, 6]

In [48]: b = []

In [49]: for i in a:
   ...:     b = b + [i*i-i] + [i*i]
   ...:

In [50]: b
Out[50]: [-1e-309, 0.0, nan, inf, 2.9899999999999993,
   5.289999999999999, 30, 36]

In [51]: from scipy import isnan, isinf

In [52]: isnan(b[2])
Out[52]: True

In [53]: b[2]
Out[53]: nan

In [54]: isinf(b[2])
Out[54]: False

In [55]: isinf(b[3])
Out[55]: True

In [56]: b[3]
Out[56]: inf

In [57]:
```

Presenting the preceding cautionary computational subtleties has the main intention of making you aware that numerical computing with decimals has some important limitations that (if working with care) can be overcome (if not, look around you to see the many devices that work using finite floating point arithmetic). There are also examples stressing the fact that we need to be careful in extreme when computing results with decimals could put any life at risk (see the references for a few pointers). In this endeavor, when dealing with numerical computations,

✓ Be skeptical when confronted with numerical results until you can verify them by other methods with its respective error boundaries. This helps to minimize problems due to the effects of loss of precision by aggregation.

✓ Take a moment to evaluate how the computation was performed. Check the operational manual to find out which algorithm was implemented to perform the computation. This helps to avoid non-robust behaviors of the implementation.

✓ Check the result by using different hardware/operating system platforms to avoid cross-platform inconsistencies of the results.

6.2.3.1 Some observations when computing with decimals

Illustrated in the preceding section, numerical computing in any computer uses finite digit arithmetic, which leads to some inconsistencies with the rules of algebra you are learning in your Prealgebra course work that we need to be aware of:

1. Because of the rule of underflow, addition or subtraction of a very (not zero) small number might have no effect.
2. $a \times (\frac{1}{a})$ is not always 1.
3. $(a + b) + c$ in not always equal to $a + (b + c)$.
4. Subtracting two floating points close in value might result in roundoff errors that can be magnified by further division of the result by a small number or by multiplying the result by a big number.
5. No calculation is more accurate than its weaker portion.
6. Round-off errors can be reduced by reformulation of the calculation in such a way that the number of computations is lowered. For instance, instead of computing $x^3 + bx^2 + cx + d$ (four multiplications and three additions) it might be better to use $((x + b)x + c)x + d$ (two multiplications and three additions).
7. If ram memory is not a problem, avoid computing with decimals or use the most precise type of floating point avilable in your computations.

6.2.4 Computing with decimals using extended precision

In the preceding section we learned situations on which finite digit arithmetic implemented in practically any computer of daily use does not satisfy the rules of algebra, generating imprecise results (measured by the relative error defined via equation 6.1) when computing with decimals in *Python*. At the end of the section we mentioned that a way around this inconvenience is to avoid decimal operations using instead integer or fractions in computations. In large scale computational projects these alternatives could be very slow and we can also run out of ram memory.

To give you an idea of the ram memory usage in a computation, let's consider the case of 64 bits computing with *Python*floats (which, as you know, are the approximated representations of decimals in the *Python*), but this discussion is valid for any other programming language using this representation.

To store one 64 bit float, 8*bytes* are required. One byte (1B) is a storage computer measurement unit containing eight bits. In terms of more common units, we have that *one kilo bytes* (1KB) corresponds to 1024 bytes (1KB = 1024B), while *one mega bytes* (1MB) is equivalent to 1024KB (1MB = 1024KB), continuing with *one giga bytes* (1GB) that is equivalent to 1024MB (1GB = 1024MB), and so forth.

Consequently, to store a million (10^6) floats (not an unusual quantity in large scale computations), we need the amount of $(8B)(1000000) = 8{\times}10^6B \approx 7813KB \approx 7.63MB$. For infinity precision operations with integers and fractions, this requirement of memory could growth very quickly. But for small computational projects, today's standards in ram memory and hard drive storage can satisfy the requirements of not intensive computational needs.

Thus, you can take advantage of such possibility to perform your exact Prealgebra computational requirements with infinity precision via computing with integer and fractions in *Python* and using the *S* function of *SymPy*, as we did in the previous chapter. A further illustrative example follows, using the *nsimplify* function from the *SymPy* module to convert a float to a fraction (check the documentation of *nsimplify* for further examples on how to use it):

Chapter 6, IPython session 19

```
In [1]: A = 0.0254

In [2]: B = 9788.0

In [3]: ( (A+B)**2 - 2*A*B - B**2)/A**2
Out[3]: 1.0000241431738204

In [4]: from sympy import nsimplify

In [5]: A = nsimplify(A)  # convert A to a Rational object

In [6]: A
Out[6]: 127/5000

In [7]: B = nsimplify(B)  # convert B to an Integer object

In [8]: B
Out[8]: 9788

In [9]: ( (A+B)**2 - 2*A*B - B**2)/A**2
Out[9]: 1
```

```
In [10]: type(A)
Out[10]: sympy.core.numbers.Rational

In [11]: type(B)
Out[11]: sympy.core.numbers.Integer

In [12]:
```

See exercise 6.2 (page 241) for an extra practice in this issue.

By the way, to further enhance its computational capabilities, *Python* has available a set of tools for memory management which are very useful to handling efficiently that important computational resource in case of need (see the references at the end of this chapter, on page 242, for pointers to it.)

Now, in situations were working with infinite precision is not an option, in *Python* there are available options to perform floating point operations with extended precision, two of which will be present in what follows.

6.2.4.1 Using *SymPy* extended float precision

The module *SymPy* offer several ways for working with extended (or reduced) precision. For them to work, each number must be defined with the chosen precision to have the same accuracy in computing with such quantities. There at least two functions for this kind of operations in *SymPy*: *N* and *Float*. You can made them available in the current computational environment using the usual *import* instruction:

Chapter 6, IPython session 20

```
In [1]: from sympy import N

In [2]: N? # Hit return to read the documentation

In [3]: from sympy import Float

In [4]: Float? # Hit return to read the documentation

In [5]:
```

We will use the *N* function in what follows, leaving to you the curiosity to explore on your own the other function. This function *N* is used as follows:

```
N(number, precision)
```

where *precision* is optional. If not given, the default value of standard *Python* is used. Here are some examples:

Chapter 6, IPython session 21

```
In [63]: N(1)
Out[63]: 1.00000000000000

In [64]: N(100)
Out[64]: 100.000000000000

In [65]: a = N(10000000)

In [66]: a
Out[66]: 10000000.0000000

In [67]: type(a)
Out[67]: sympy.core.numbers.Float

In [68]:
```

Notice that the *type* of the object returned by N is *Float* (you might correctly guess it is of the same type returned by the other mentioned function to perform extended precision computations).

Let's now use this function to perform the computing of the familiar quantity $2^{64} - 1$ from chapter 2, page 31.

Chapter 6, IPython session 22

```
In [1]: a = 2

In [2]: type(a)
Out[2]: int

In [3]: exactVal = a**64 - 1

In [4]: exactVal
Out[4]: 18446744073709551615
```

```
In [5]: a = 2.0

In [6]: type(a)
Out[6]: float

In [7]: floatVal = a**64 - 1

In [8]: floatVal
Out[8]: 1.8446744073709552e+19

In [9]: from sympy import N

In [10]: a = N(a, 20)

In [11]: a
Out[11]: 2.0000000000000000000

In [12]: type(a)
Out[12]: sympy.core.numbers.Float

In [13]: sympyVal = a**64 - 1

In [14]: sympyVal
Out[14]: 18446744073709551615.

In [15]:
```

Our next example explores the computation of the imprecise result of subtracting two similar numbers and dividing the result by a small one:

Chapter 6, IPython session 23

```
In [25]: A = 0.0254 ; B = 9788.0

In [26]: A = N(A, 30) ; B = N(B, 30)

In [27]: ( (A+B)**2 - 2*A*B - B**2)/A**2
Out[27]: 1.00000000000000000000049606753

In [28]: A = N(A, 50) ; B = N(B, 50) # increase precision to verify last
    digits
```

```
In [29]: ( (A+B)**2 - 2*A*B - B**2)/A**2
Out[29]: 1.00000000000000000000000000000000000000000000000000

In [30]:
```

See exercise 6.3 (page 241) for getting the right answer of the multiplication problem of page 214. You are also encourage to change the precision figure to see its influence on the obtained output.

Our final example uses the triple 3987, 4365, and 4472 which according to a Simpson's episode (see the reference page for the YouTube video), that using power 12 the triple violates Fermat's last theorem, meaning that $3987^{12}+4365^{12}-4472^{12}=0$. Using *Python* integer infinite precision we can easily disprove that claim, but just for fun let's do the math in an ordinary (non-scientific) calculator (mimic here using *SymPy* reduced precision):

Chapter 6, IPython session 24

```
In [52]: x = 3987. ; y = 4365. ; z = 4472.

In [53]: x = N(x,9) ; y = N(y,9) ; z = N(z,9)

In [54]: x
Out[54]: 3987.00000

In [55]: y
Out[55]: 4365.00000

In [56]: z
Out[56]: 4472.00000

In [57]: x**12 + y**12
Out[57]: 6.39766563e+43

In [58]: z**12
Out[58]: 6.39766563e+43

In [59]: x**12 + y**12 - z**12
Out[59]: 0

In [60]: x**12 + y**12 - z**12 == 0
Out[60]: True
```

```
In [61]:
```

Your next exercise is disprove this equality (see exercise 6.4, page 241. Moreover, you could go back and redo the *IPython* sessions of the preceding section 6.2.3 (starting at page 211) using this way of increasing the precision and accuracy of a computation using the N function from the *SymPy* module.

As you might have already anticipated from the examples in this section on using the N function from the *SymPy* module to increasing the precision and accuracy of a computation, that is a usual technique to check the validity of a computation. That is, we perform a calculation with a particular precision and accuracy, and then we increase them to compare the obtained results (sometimes this is referred to increasing the order of the computation). This is particular useful in the production stage of our code, when we are in the process of generating new results (we have already used test cases in evaluating the correctness of the program being use).

Let's finish this section mentioning that *Python* offer another alternative via the *NumPy* module to increase the precision of a computation up to 128 bits, which can be used as a preliminary check for large scale computations because numerical computing via *NumPy* is more efficient than doing so with *SymPy*. Discussing it is outside the scope of your Prealgebra course work enhancement and enrichment journey.

6.2.5 Special mathematical functions and numbers in *Python*

In this section we will present how to compute with many mathematical functions available in *Python*. These include *logarithms*, *square roots*, *exponential*, *trigonometric*, inverse trigonometric, and the special numbers *pi* and *e*.

6.2.5.1 Special mathematical functions and numbers via *SymPy*

The following *IPython* session defines the special numbers *pi* and *e* using *SymPy*:

Chapter 6, IPython session 25

```
In [8]: import sympy as sp

In [9]: from sympy import N

In [10]: sp.exp(1)
Out[10]: E

In [11]: N(sp.E) # prints the value e using standard precision
Out[11]: 2.71828182845905
```

```
In [12]: N(sp.pi) # prints the value pi using standard precision
Out[12]: 3.14159265358979

In [13]: mypi = N(sp.pi, 50) # assigns to variable mypi 50 digits pi

In [14]: mypi
Out[14]: 3.1415926535897932384626433832795028841971693993751

In [15]: mye = sp.exp(N(1., 50)) # assigns to variable mye 50 digits of e

In [16]: mye
Out[16]: 2.7182818284590452353602874713526624977572470937000

In [17]: mye = N(sp.E,50) # assigns to variable mye 50 digits of e

In [18]: mye
Out[18]: 2.7182818284590452353602874713526624977572470937000

In [19]:
```

Using the previous setup, the trigonometric functions (sin, csc, cos, sec, tan, cot) are computed in the following way:

Chapter 6, IPython session 26

```
In [41]: sp.sin(mypi/4)
Out[41]: sqrt(2)/2

In [42]: sp.cos(mypi/4)
Out[42]: sqrt(2)/2

In [43]: sp.tan(mypi/4)
Out[43]: 1
```

and the inverse trigonometric functions (asin, acsc, acos, asec, atan, acot) are also defined:

Chapter 6, IPython session 27

```
In [54]: sp.acos(sp.sqrt(2)/2)
Out[54]: pi/4

In [55]: sp.asin(sp.sqrt(2)/2)
Out[55]: pi/4

In [56]: sp.atan(1)
Out[56]: pi/4

In [57]:
```

There are also some trigonometric identities

Chapter 6, IPython session 28

```
In [61]: from sympy import symbols

In [62]: x, y = symbols('x y')

In [63]: sp.trigsimp(sp.sin(x)**2 + sp.cos(x)**2)
Out[63]: 1

In [64]: sp.expand_trig(sp.sin(x + y))
Out[64]: sin(x)*cos(y) + sin(y)*cos(x)

In [65]: sp.expand_trig(sp.cos(x + y))
Out[65]: -sin(x)*sin(y) + cos(x)*cos(y)

In [66]: sp.expand_trig(sp.sin(2*x))
Out[66]: 2*sin(x)*cos(x)

In [67]:
```

Taking Logarithms is as follows:

Chapter 6, IPython session 29

```
In [88]: sp.log(sp.exp(1)) # natural logarithm of number E
Out[88]: 1

In [89]: sp.log(sp.E) # natural logarithm of number E
Out[89]: 1

In [90]: sp.log(sp.exp(1), 10) # base 10 logarithm of number E
Out[90]: 1/log(10)

In [100]: sp.log(sp.E, 10) # base 10 logarithm of number E
Out[100]: 1/log(10)

In [101]:
```

Other modules have also available the numerical versions of these functions. In particular, the modules *NumPy* and *SciPy* has them available.

6.2.6 Solving equations involving decimals via *SymPy*

When working with equations in the context of whole numbers 3.9 (page 119) or integers 5.2.3 (page 176), or fractions 5.4.6 (page 195), the fact that in each of these cases we have used the same setup in the use of *Python* to find (via *SymPy*) the solution of any equation (namely, identify the left and right side of the equation to give them as input to the solver), it can readily be anticipated that such a way for finding solutions of equations via *SymPy* can also work in finding solutions to equations containing decimals (as a more general representation of numbers which can not be expressed as fractions, like the irrational numbers such as π). You might want to reread the aforementioned sections bout equations to refresh yourself how the procedure works.

Since equations play a very important role in modeling the world out there (close and far away as you already know from your Physics, Chemistry and Biology classes), before using *SymPy* to find any solution, it is important that you get acquainted with the whole set of steps presented in your Prealgebra course work that you can take to find solution to equations because *SymPy* will hide them from you (unless it is programmed to show the steps, and we are not doing so here). In your Prealgebra course work you have been solving so many routine problems in order to learn the different methods applied to find solution to the equations you encounter at this level of your formal education. You could practice such procedures by solving step by step many of the exercises at the end of the corresponding chapter of your Prealgebra textbook using the program presented and described in the Appendix of this chapter, on page 236. The knowledge of *Python* you have reached at this point, will allow you to explore and understand the core *Python* function used to create that program (though you still need to still understand on your own the few basic elements of *Flask* that we used to show the functionality of the program in a web browser).

Besides that, it is also crucial at this stage of your formal education to think about equations beyond the literal "find the solution of $\cdots$". You need to start thinking about what the equality sign in an equation is telling you. The most straight forward understanding of it is as an indication to find the value of the unknown variable that makes equal the left and the right hand side of the equation (when the unknown is assigned a value, numerical or symbolic, that is a solution the whole expression becomes a true statement). But you could also think of the equal sign as indicative that you could use one side of the equation in place of (or equivalent to) the other side of the equation and vice-versa in any chain of reasoning in which either side of the equation appear.

Let's see an example. Suppose we are dealing with the equations:

$$ax + b = c \tag{6.5}$$

In the usual interpretation of *balancing* both sides of the equation (or solving for the unknown variable x) it is found that with $x = (c - b)/a$ (obviously a must be different from zero. Otherwise $b = c$) both sides of the equation are balanced. But the equality sign in equation 6.5 also means that in place of c one could use $ax + b$ or vice-versa, in place of $ax + b$ one could use c in any chain of reasoning associated with equation 6.5. This interpretation is particularly useful when solving system of equations or making proof. We did that in the Appendix of Chapter 2 (page 59), when replacing the left hand side of equation A.1 (page 59) appearing in the equation A.2 by the right hand side of equation A.1 (to see this in detail, go through the equations A.1--A.5).

On the other hand, equations are a guide to our thinking. Finding solutions of them activate thought processes of high order which should be reinforced with a method, like the general three steps one for writing algorithms: *designing--implementing--testing*. Designing goes in the heart of *thinking before implementing*. It involve asking questions: can I use induction of deduction? should I put everything in one side of the equation? what set of operations allows me to do so? Could a division appear in my chain of reasoning? If it happen, how can I deal with it.? Then, your *implementing* strategy needs to be guided by verifying you are getting equivalent equations, that you are not loosing solutions or that your are not adding new ones. A crucial point here is verifying that each given step is a valid step (you'll find that incorrect procedures leads to right results, like the misuse of the cancelation trick in computing $\frac{\cancel{6}5}{1\cancel{6}} = 5$. The answer (5) of the division is certainly right but the procedure, as your intuition says, is clearly wrong.) Finally, the *testing* stage could bring you to verify that your solution is correct, perhaps plugin the solution in the equation, but, does the found solution has meaning in the context of the equation? Does it makes sense to find 1000 as the age to a human being? In what units that answer could be true? By making this sort of questions your equation guides your thinking.

Let's continue by finding the solution of (notice that in the equation could now appear decimals, fractions and integers. We are not restricted on using a particular set of numbers as in the previous sections):

$$frac35 = 5.2 * x + 9 \quad \text{or} \quad 0.6 = 5.2 * x + 9.0 \tag{6.6}$$

Surely your Prealgebra textbook has plenty of similar equations stated just like that. In the previous sections about equations, we have stressed the fact that whenever you try to solve an equation, you need to spent sometime thinking about what that equation could represent. In section 5.2.3 (page 176) about solving equations with integers we gave some ideas on how you can rephrase the wording presenting equation 5.1 (page 232) to make it look as if it were the result of a word problem. You were able to see that the same equation could be assigned to more than one word problem representing different situations. What makes unique the problems is the meaning of the equations in each respective context: perhaps negative solutions are not allowed in a particular case; perhaps the point of interest in another case is in the set of numbers greater than the one that solve the equation. There are plenty of possibilities. You named it!. The point is that one need to look at equations beyond the usual "find the solution of $\cdots$" and think of the equations in your Prealgebra course works with the same look you do at those equations that you see in your Physics, Biology, Chemistry, subjects. You need to go beyond the overwhelming state of simple learning the methods to solve and equation (that needs to be learned) and find meaning on each equation (try to see what they might represent connecting your thought processes with other subjects like Physics, Chemistry or Biology). As exciting and interesting this discussion might be, it will take us beyond the scope of the book as we need to continue with our subject of finding the solution of the posed equation 6.6 via *Python*. To further explore the meaning of equations, you are encourage to check the references at the end of Chapter 3, starting on page 144.

Before continuing in finding the solution of equation 6.6 in *Python* via *SymPy*, just keep in mind that *SymPy* will hide from you how it finds the solution of the equation. Consequently, you are encourage to workout the exercise following by hand (using pencil and paper or via our program described in the Appendix of this chapter, on page 236) the procedures for solving equations presented in your Prealgebra course work. This way you will develop your intuition on whether the solution found make or not sense and how to check it.

A piece of code to find the solution of equation 6.6 (which you can find under the name `chap06_prog_01_Sympy_SolEquation.py` in the directory named `chapter_06` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface of this book) is as follows:

```
"""
  This program finds the solution of the equation
     3/5 = 5.2x + 9
"""
from sympy import symbols, Eq, solveset, S, sympify

```

```
x = symbols('x')

LHS = S('3')/5
RHS = 5.2*x + 9

thesol = list( solveset( Eq(LHS, RHS), x) )

# Check the found solution by substitution
newLHS = LHS - RHS #rearrange the equation to read: LHS - RHS = 0

newLHS = newLHS.subs(x, thesol[0])
if newLHS.simplify() < 1e-15:
   print('We verified that x = {2} is solution of {0} =
       {1}'.format(LHS,RHS,thesol[0]))
```

After executing these lines of code you'll get:

Chapter 6, System shell command 1

```
$ python chap06_prog_01_Sympy_SolEquation.py
We verified that x = -1.61538461538462 is solution of 3/5 = 5.2*x + 9
```

If you have trouble understanding this code, please go back and read section 3.9, starting on page 119.

To check on your onw that the show solution is correct, you are encourage to workout the exercise following by hand (using pencil and paper or via our program described in the Appendix of this chapter, on page 236) the procedures for solving equations presented in your Prealgebra course work.

At this point, you know (from the many examples given in the previous sections of this chapter) that working with decimals in the limited floting point representations of these numbers in the computer is prone to errors that propagates across associates computations using them. In case efficiency is not an issue, any decimal (with a finite set of digits) can be represented as a fraction, a task that can be done in *SymPy* via the function *nsimplify*. Consequently, any equation containing decimals can be converted to an equivalent equation containing rational numbers. By solving the later we will get an exact (rational) solution which can then be converted to a decimal value using the *sympy* *N* function.

This is illustrated in the following lines on *Python* code that find the solution of the equation 6.6, which you can find under the name `chap06_prog_02_Sympy_SolEquation.py` in the directory named `chapter_06` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface of the book:

```
"""
  This program finds the solution of the equation
     3/5 = 5.2x + 9 written as 0.6 = 5.2x + 9
"""
from sympy import symbols, Eq, solveset, N, nsimplify

x = symbols('x')

LHS = 0.6
RHS = 5.2*x + 9

# convert numerical values on each part of the equation to fractions
LHS = nsimplify(LHS)
RHS = nsimplify(RHS)

thesol = list( solveset( Eq(LHS, RHS), x) )
#print('The found solution is x =', thesol[0])

# Check the found solution by substitution
newLHS = LHS - RHS #rearrange the equation to read: LHS - RHS = 0
#print('newLHS =', newLHS)

newLHS = newLHS.subs(x, thesol[0])
if newLHS.simplify() == 0:
   print('x = {2} is solution of {0} = {1}'.format(LHS,RHS,thesol[0]))
   print('\t Which using decimals, is equivalent to saying that: ')
   print('x = {2} is solution of {0} =
       {1}'.format(N(LHS),N(RHS),N(thesol[0])))
```

After executing these lines of code you'll get:

Chapter 6, System shell command 2

```
$ python chap06_prog_02_Sympy_SolEquation.py
x = -21/13 is solution of 3/5 = 26*x/5 + 9
   Which using decimals, is equivalent to saying that:
x = -1.61538461538462 is solution of 0.600000000000000 = 5.2*x + 9.0
```

You can try solving the equation step by step via our program described in the Appendix of this chapter, on page 236, following the procedures for solving equations presented in your Prealgebra course work.

Let's end this section mentioning that, in general, when finding numerical solution to equations, we might need to use other *Python* alternatives better suitable than *SymPy*, like *SciPy*.

6.3 Chapter Summary

By reaching the end of this chapter you have done wonderful! After finishing this chapter you know how to handle in *Python* computational operations of your Prealgebra course work involving decimals. You learned how decimals are represented in *Python* using a limited 64 bit of precision. Accordingly, this limited representation of decimals force the computer to work with a set of rules known as *floating point arithmetic* which in many cases lead to contradicting the rules of algebra. We learned how to overcome such limitations working with *SymPy* fractions or using extended precision arithmetic.

We also learned about some special numbers like *pi* and *e* are represented in *Python*, in addition on how to perform computations involving logarithms, trigonometric functions and their inverse, square roots, and so forth. In addition, we also covered the solving of equations involving decimals in *Python* via *SymPy*. Moreover, the appendix of this chapter describes a *Python* solver equation that we wrote and that you could use to practice the operations you have learned in your Prealgebra course work to find solution of equations step by step. The solver is general enough that could handled equations with symbolic and/or numerical parameters.

More importantly, in the context of programming, by studying the afore mentioned application you could start building your own programming applications either to be executed in a system shell (terminal) or in a browser (though for this you need to learn about Flask or any other technology allowing the generation of dynamic output for the web).

In the next chapter we will learn about the basics of making plots of a data set in *Python* via *Matplotlib*.

Appendix of Chapter 6

A.1 A simple *Python* one variable, linear equation solver run on a browser via Flask

This appendix shows a one variable solver equation we have written with the working knowledge of *Python* you learned up to now. This application can guide your thoughts in writing your own ones. We have chosen to display it using a web browser and Flask (thus you need to learn a bit of it on your own or any other technology allowing the display of dynamic content on a browser).

Python has a few alternatives to write *graphical user interfaces*, but we found its instructive o show that it can be used to deploy applications over a web browser (which in turn can be used over the internet by properly configuring a web server). We are leaving as exercise 6.5 (page 241) for you to write a *text interface* of the main solver which you can find under the name `EquationSolver_funcs.py` in the directory named `chapter_06/PythonFlask_EquationSolver` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface of the book.

To use our solver, you need o go to the folder `PythonFlask_EquationSolver` under the directory `chapter_06` just mentioned. Once in there, you need to execute on a system shell (terminal) the instruction:

Chapter 6, System shell command 3

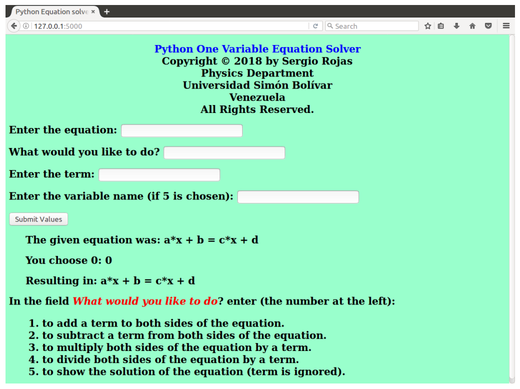
```
$ python EquationSolver.py
```

to which the system will output:

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 113-137-830
```

Now you need to open a web browser on your computer and point it to browse the address listed after “Running on” in the first line of the output shown above (which in our case is `http://127.0.0.1:5000/`). Whenever you want to end this program you need to hit **CTRL--C**, as mentioned in that same output line.

After pasting the given address in the respective place of the browser, it will show a page looking as shown in figure A.1 below.



Python One Variable Equation Solver
Copyright © 2018 by Sergio Rojas
Physics Department
Universidad Simón Bolívar
Venezuela
All Rights Reserved.

Enter the equation:

What would you like to do?

Enter the term:

Enter the variable name (if 5 is chosen):

The given equation was: a*x + b = c*x + d

You choose 0: 0

Resulting in: a*x + b = c*x + d

In the field *What would you like to do*? enter (the number at the left):

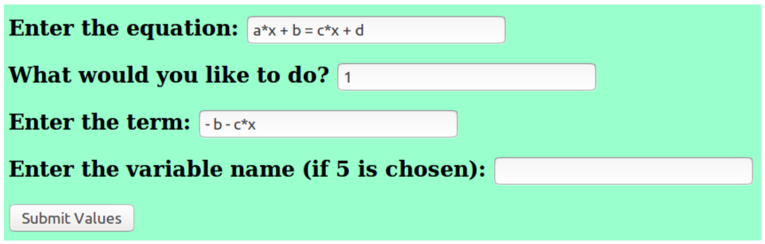
1. **to add a term to both sides of the equation.**
2. **to subtract a term from both sides of the equation.**
3. **to multiply both sides of the equation by a term.**
4. **to divide both sides of the equation by a term.**
5. **to show the solution of the equation (term is ignored).**

Figure A.1: General one variable equation solver in *Python*.

In what follows we will only showing relevant figures, leaving the top and bottom header of the page. We will show how to find step by step solution of the default shown equation, but you could use any other equation.

1. In the box at the right side of **Enter the equation**, we need to write down the equation whose solution we want to find. This equation needs to be entered using *Python* math symbology. In that box we write $a * x + b = c * x + d$ (any extra white space will be ignored)
2. In the box next to **What would you like to do?**, we need to enter a number from the options (1--5) listed at the bottom of the web page. we are going to use 1.
3. The option 1 in the previous item means to add a term to both sides of the equation. We could enter in the box next to **Enter the term**, any term to add to both sides of the equation that we think could help us to find a reduced equivalent equation. For instance, we can enter $-b$, or $-d$, or $-b - c * x$, or $-d - a * x$ or any other term that you like. To make this demonstration a bit faster we will enter $-b - c * x$. **We are assuming that** x **is the variable (unknown) in our equation**.
4. We leave empty the box next to **Enter the variable name (if 5 is chosen)** as it will be ignored in this case.

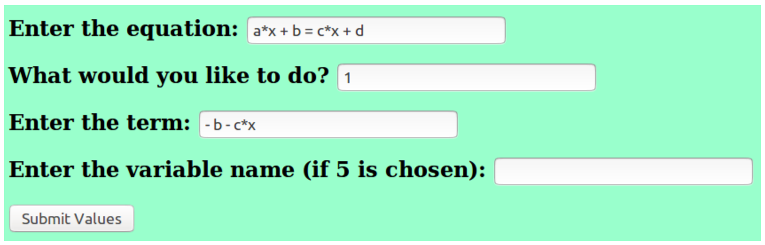
Figure A.2 below shows the above setup before hitting the **submit Values** bouton.



Enter the equation:	a*x + b = c*x + d
What would you like to do?	1
Enter the term:	- b - c*x
Enter the variable name (if 5 is chosen):	

Figure A.2: Filling the boxes of the equation solver of figure A.1.

After hitting the **submit Values** bouton the boxes becomes empty and the following output is shown below it:



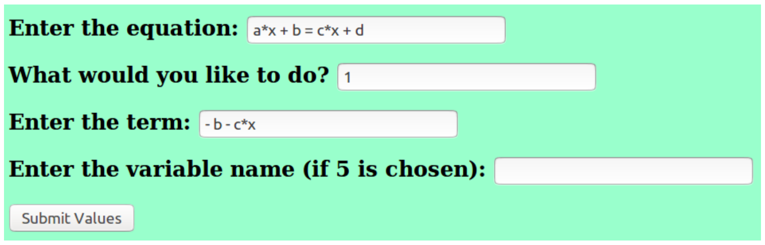
Enter the equation:	a*x + b = c*x + d
What would you like to do?	1
Enter the term:	- b - c*x
Enter the variable name (if 5 is chosen):	

Figure A.3: The output after hitting the **submit Values** bouton of figure A.2.

Our new equivalent equation is the one shown next to where it is said **Resulting in** (which in this case is the equation $a * x - c * x = -b + d$). Notice that the left hand side contains non gruped (unfactored) combinations of the uknown x while the right hand side contains only algebraic parameters. We repeat the previous step:

1. In the box at the right side of **Enter the equation**, we write our new equation $a * x - c * x = -b + d$.
2. In the box next to **What would you like to do?**, this time we enter 4.
3. The option 4 in the previous item means to divide by a term both sides of the equation. We will choose c (a could also be chosen) to further reduce the left hand side of the equation.
4. We leave empty the box next to **Enter the variable name (if 5 is chosen)** as it will be ignored in this case.

Figure A.4 below shows the above setup before hitting the **submit Values** bouton.


Enter the equation: a*x + b = c*x + d

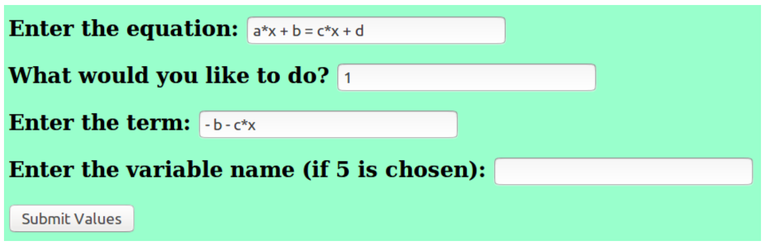
What would you like to do? 1

Enter the term: - b - c*x

Enter the variable name (if 5 is chosen):

Figure A.4: New filling the boxes of the equation solver.

After hitting the **submit Values** bouton the boxes becomes empty and the following output is shown below it:


Enter the equation: a*x + b = c*x + d

What would you like to do? 1

Enter the term: - b - c*x

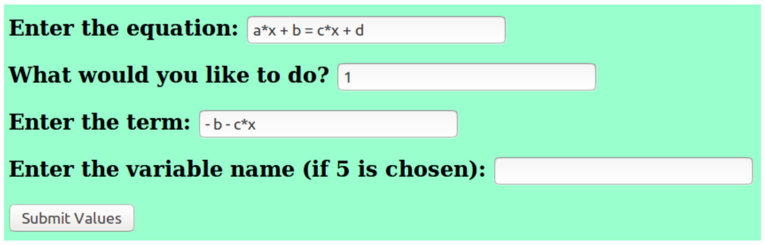
Enter the variable name (if 5 is chosen):

Figure A.5: The output after hitting the **submit Values** bouton of figure A.4.

Our new equivalent equation is the one shown next to where it is said **Resulting in** (which in this case is the equation $x * (a - c)/c = (-b + d)/c$). Notice that this time the left hand side contains gruped (factored) combinations of the uknown x while the right hand side contains only algebraic parameters. We repeat the previous step:

1. In the box at the right side of **Enter the equation**, we write our new equation $x * (a - c)/c = (-b + d)/c$.
2. In the box next to **What would you like to do?**, this time we enter 3.
3. The option 3 in the previous item means to multiply by a term both sides of the equation. We will choose the full necessary term $c/(a-c)$ to go a little faster, but you could continue the steps as if you were doing the using pencil and paper.
4. We leave empty the box next to **Enter the variable name (if 5 is chosen)** as it will be ignored in this case.

Figure A.6 below shows the above setup before hitting the **submit Values** bouton.



Enter the equation: a*x + b = c*x + d

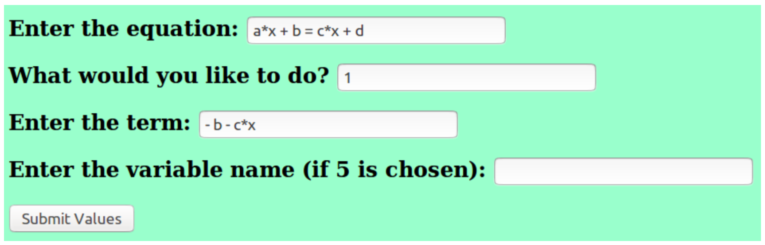
What would you like to do? 1

Enter the term: - b - c*x

Enter the variable name (if 5 is chosen):

Figure A.6: New filling the boxes of the equation solver.

After hitting the **submit Values** bouton the boxes becomes empty and the following output is shown below it:



Enter the equation: a*x + b = c*x + d

What would you like to do? 1

Enter the term: - b - c*x

Enter the variable name (if 5 is chosen):

Figure A.7: The output after hitting the **submit Values** bouton of figure A.6.

This time, as expected, we reached the solution of our equation $x = (-b + d)/(a - c)$, shown next to where it is said **Resulting in**.

We let you as an exercise to further explore the functionality of this solver, which needs to be further improved.

Exercises of Chapter 6

Exercise 6.1 *Compute the relative error and the percent error of the computations in page 214.*

Exercise 6.2 *Obtain the exact result of the operations on page 214 converting to fractions the values of A and B.*

Exercise 6.3 *Obtain the exact result of the operations on page 214 using extended precision for the values of A and B.*

Exercise 6.4 *Find the exact value of $x^{12}+y^{12}$ and compare it with z^{12} on the computations in page 226. How would you find the closes (non-integer) value for z to best satisfy the equality. What value did you find? Print some lines of code testing your result.*

Exercise 6.5 *Wrtie a text interface for the function `EquationSolver_funcs.py` we wrote to find step by step solutions a any one variable equation you will find in your Prealgebra course work (see the Appendix of this chapter for details, on page 236).*

References of Chapter 6

Books and/or Articles

- **Marecek, L. and Smith, M. A.** (2017). Prealgebra, Rice University, OpenStax `https://openstax.org`.
  Book available for free at: `http://cnx.org/content/col11756/1.9`
- **Burden, R. L. and Faires, J. D.** (2011). Numerical Analysis, 9th Edition, Brooks/-Cole.
-

References on the WEB

- The IEEE Standard for Floating-Point Arithmetic (IEEE 754)
  `https://en.wikipedia.org/wiki/IEEE_754`
- What Every Computer Scientist Should Know About Floating-Point Arithmetic:
  `https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`
- Floating Point Arithmetic: Issues and Limitations
  `https://docs.python.org/3/tutorial/floatingpoint.html`
  `https://en.wikipedia.org/wiki/Floating-point_arithmetic`
  `https://en.wikipedia.org/wiki/Machine_epsilon`
- Decimal fixed point and floating point arithmetic:
  `https://docs.python.org/3/library/decimal.html`
  `https://en.wikipedia.org/wiki/Extended_precision`
- Software bugs:
  `https://en.wikipedia.org/wiki/List_of_software_bugs`
  `https://en.wikipedia.org/wiki/Software_bug`
- *Python* Memory managegement `https://docs.python.org/3/c-api/memory.html`

- Computer prefixes:
  https://en.wikipedia.org/wiki/Binary_prefix
- Simon Singh: The Simpsons and Their Mathematical Secrets
  https://youtu.be/bk_Kjpl2AaA?t=912
- DOE steps further toward exascale computing (2018):
  https://physicstoday.scitation.org/doi/10.1063/PT.6.2.20180411a

  ***Python* tutorial**:
  https://docs.python.org/3/tutorial/index.html
- **SymPy tutorial**:
  http://docs.sympy.org/latest/tutorial/index.html

7

Graphing and data visualization in *Python*

"If I have seen farther than others, it is because I have stood on the shoulders of giants."
Issac Newton

7.1 Introductory remarks

A very important aspect of data analysis in any field of the organized knowledge is graphing and data visualization as it helps to uncover patterns or tendencies in the data behavior and can also guide our thoughts processes to new ways of quantitative analysis of the situation at hand.

Python offer (among others) the *Matplotlib* module to perform graphing and data visualization in two and three dimensional data sets. You can take a look of the sophistication by this module to create plots by exploring the *Matplotlib gallery* at `http://matplotlib.org/gallery.html`.

In this and last chapter of this book we will presenting the basic ideas to get you started in plotting two dimensional data that you will find in your Prealgebra course work. In the reference section of this chapter (page 254) you'll find a key literature that will guide you to deeper your understanding of this important *Python* library.

7.2 Graphing two dimensional data with *Matplotlib*

We will start by showing how to make any of the plots shown in Figure 7.1. Typically, the unjoined plot on Figure 7.1a (at the left) is an exploratory plot to verify how the data is configured. It also apply to unordered data or data that we don't know whether it can be joined by a line. The Figure 7.1b (at the right) is when we know that the data set can be joined by a line (In Table 7.1 we show a few of the style options that *Matplotlib* provides for lines joining the data set, as well as some of the available options for markers).

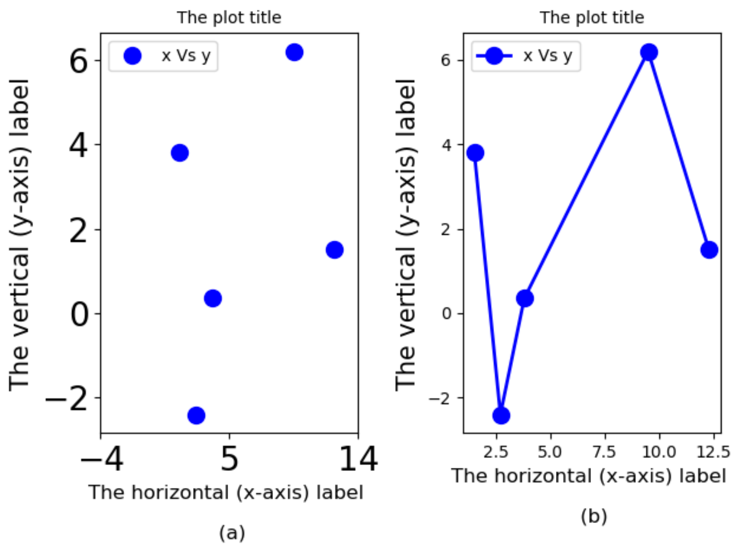
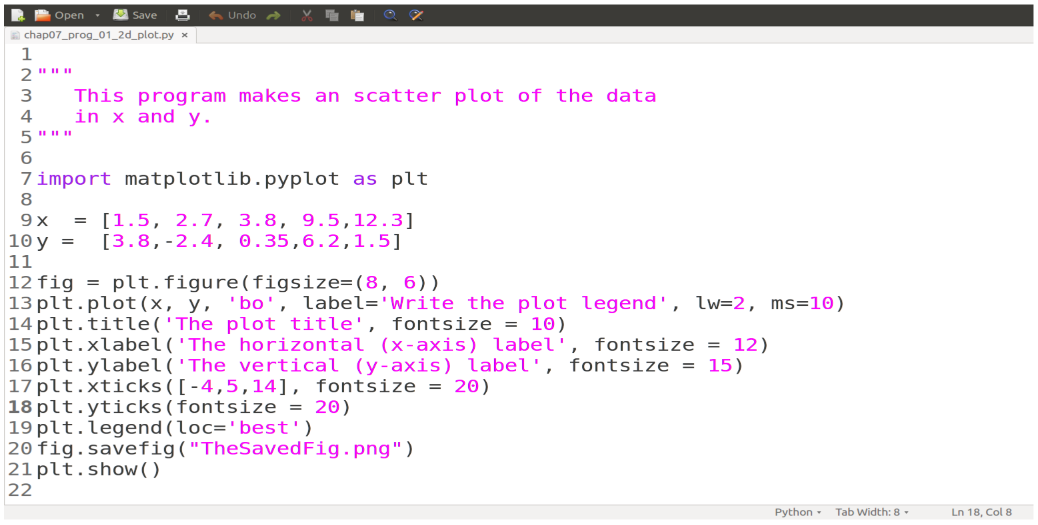


Figure 7.1: *Matplotlib* Example of basic two dimensional plots

Before making a plot, we need to have the data x (that goes to the horizontal *x-axis*) and the data y (that goes to the vertical *y-axis*) available i in separated corresponding (same length) *Python lists* (other ways are available, including the use of *NumPy* arrays, but you can learn about them perusing the listed references). The numbers can be of any type or a mixture of them. Accordingly, we will use the following data set in our example (feel free to add more data to check the generality of the presented procedure):

```
x = [1.5, 2.7, 3.8, 9.5, 12.3]
y = [3.8, -2.4, 0.35, 6.2, 1.5]
```

To make the plot of this set of x and y data points we can use the *Python* program listed in Figure 7.2.



```python

"""
   This program makes an scatter plot of the data
   in x and y.
"""

import matplotlib.pyplot as plt

x  = [1.5, 2.7, 3.8, 9.5,12.3]
y =  [3.8,-2.4, 0.35,6.2,1.5]

fig = plt.figure(figsize=(8, 6))
plt.plot(x, y, 'bo', label='Write the plot legend', lw=2, ms=10)
plt.title('The plot title', fontsize = 10)
plt.xlabel('The horizontal (x-axis) label', fontsize = 12)
plt.ylabel('The vertical (y-axis) label', fontsize = 15)
plt.xticks([-4,5,14], fontsize = 20)
plt.yticks(fontsize = 20)
plt.legend(loc='best')
fig.savefig("TheSavedFig.png")
plt.show()

```

Figure 7.2: *Matplotlib* Making two dimensional plot shown in Figure 7.1a

This program is available under the name `chap07_prog_01_2d_plot.py` in the directory named `chapter_07` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface of the book. Please, execute this code before continuing via issuing in a system shell or terminal the command:

Chapter 7, System shell command 1

```
$ python chap07_prog_01_2d_plot.py
```

After executing this code you will see on the computer screen a plot similar to that shown in Figure 7.3 which, after eploring each element of the plot, you will need to close to recover the prompt of the system shell or terminal.

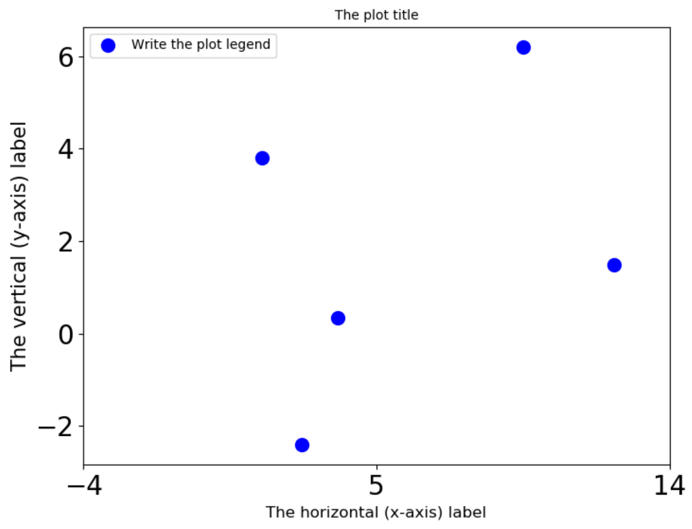


Figure 7.3: Plot generated by the code of Figure 7.2

The program starts (in line of code 7) by making available the *matplotlib.pyplot* method in our current *IPython* session with the name "plt". The code continues by defining the data we want to plot (but it can be defined previous to line 7 if we want to. We are adjusting ourself here to the conventional rule of making *Python* modules available at the beginning of the program).

To make the plot, we assign it to the name *fig*, in line of code 12. (notice the argument defining the size of the figure, 8 inches wide and 6 inches high). The actual plot is made in line of code 13, where the *plot* method is called to draw the plot. It takes as **mandatory** arguments the x and y already defined *Python lists* containing the values to be shown in the graph. The other arguments are optional (for a full discussion of them check the *Matplotlib* references listed on page 254). The string of characters *'bo'* means to graph the points in blue and use rounded bullet as the shape of the marker indicating each point in the data set. Table 7.1 list a few other values that can be used in place of the ones used in our example (they can also be combined in any way we want to). For instance, to make the plot of Figure 7.1b replace line of code 13 by the line (see exercise 7.2):

```
plt.plot(x, y, 'bo-', label='Write the plot legend', lw=2, ms=10)
```

The following lines of code (14--16) should be self evident what they do. We encourage you to make changes in the set of parameters in them and execute the code to see what they do. Lines of code 17 and 18 shows how to change the fontsize and the tick marks shown on each axis. The plot to the right (Figure 7.1b) has default values for these setting which you can obtain by commenting or deleting lines of code 17 and 18. Then follows line of code 19 which defines the way to set the *legend* of the plot defined by the *label* (optional parameter) of the

Line Styles	**Marker shape**	**Basic built-in colors**
-	x	b
–	*	g
-.	p	r
:	s	k
	+	m
	o	y
	.	w
		c

Table 7.1: Plot visualization optional parameters

plot method specified on line of code 13. Alternative values for the parameter of this line of code are listed in Table 7.2. Please try them in the provided code. To save the plot is line of

Keyword	**Numerical code**	**Keyword**	**Numerical code**
best	0	center left	6
upper right	1	center right	7
upper left	2	lower center	8
lower left	3	upper center	9
lower right	4	center	10
right	5		

Table 7.2: Legend positioning keywords

code 20. Here we are saving the plot in *png* format (included as the extension of the filename to save the plot). *Matplotlib* offer many other formats like *pdf*, *gif*, and *jpeg*. Just change the extension of the filename to get the plot in the format that you like. Finally, the code ends by issuing the command to show the plot in the computer screen (**you might need to close the plot to continue execution of the program**).

As you might have already anticipated, many of these lines of code are decorative. For a quick view of the plot we only needs to issue lines of code 7, 9, 10, 13 and 21 (you are encourage to try it on exercise 7.1, page 253). Let's mention that the lines of code 13-19 can be given in any order. Lines of code 20--21, however, must be given in that order if you want to save the plot to a file. If you show it to the screen before saving the plot, you'll get an empty file.

7.3 Fitting a curve to a two dimensional data (optional)

Once a data set has been shown in a two dimensional graph, one might have interest in fitting a curve to such data. The most common example is to find the parameters to fit a straight line to a data set and to draw the fit passing through the given points. *Python* offer many alternatives to do so. We will use the *NumPy* functions *polyfit* (to find the fitting parameters) and *poly1d* (to make a polynomial function using the already found fitting parameters).

In this section we will use the data shown in Table 7.3, which you can find in any Physics textbook or perusing the Planets astronomical data in Wikipedia [`https://es.wikipedia.org/wiki/Planeta`] The idea is to find a relationship (if any) between he two quantities.

Planet	**Orbital Period (T)** (Years)	**Orbit Radius (R)** (meters)
Mercury	88.0/365.3	5.79×10^{10}
Venus	224.7/365.3	1.08×10^{11}
Earth	365.3/365.3	1.49×10^{11}
Mars	687.0/365.3	2.28×10^{11}
Jupiter	11.86	7.78×10^{11}
Saturn	29.46	1.43×10^{12}
Uranus	84.01	2.87×10^{12}
Neptune	164.8	4.49×10^{12}
Pluto	248.7	5.91×10^{12}

Table 7.3: Planets astronomical average data respect the Sun

From your experience in Physics courses (or from driving) you know that the longer the radius of a circle the longer its take to go around it. Consequently, we might be looking for a relationship between the period (T) it takes a planet to go around the Sun versus the the average radius (R) of the orbit. In this case, we can start by looking how the data looks like in a two dimensional plot of T vs R. This is shown in Figure 7.4.

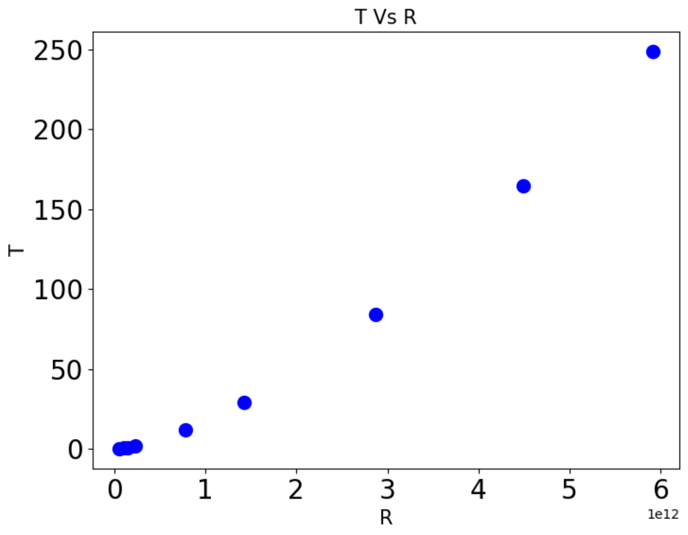


Figure 7.4: A plot of the data from Table 7.3

It is not hard to see in Figure 7.4 that there is a clear functional shape in the data. What it is hard to see is the kind of relationship we can draw from looking at the graph. To try to clarify a bit more the nature of the relationship, let's look at the plot of the natural logarithm of both quantities. That is, we take $\ln(T)$ and $\ln(R)$ to get the plot of $\ln(T)$ Vs $\ln(R)$. This could even help to resolve a bit more the shape around the points that cluster together near the origin (lower left corner) of the plot. The corresponding graph is shown in Figure 7.5.

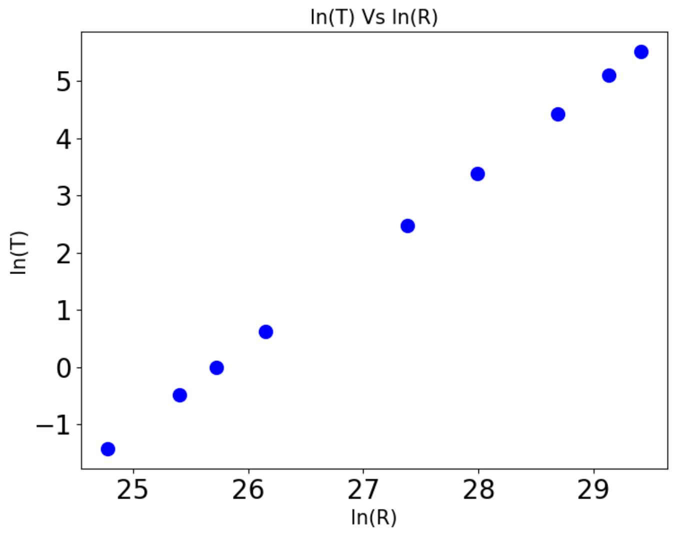


Figure 7.5: A log-log plot of the data from Table 7.3

It is clear from Figure 7.5 that a linear relationship relates $\ln(T)$ to $ln(R)$. Meaning that we could write $\ln(T) = mln(R) + b$. We need to find the values of m and b. Let's mention that in this case we have more interest on the value of m, as we'll explain shortly. The plot is shown in Figure 7.6.

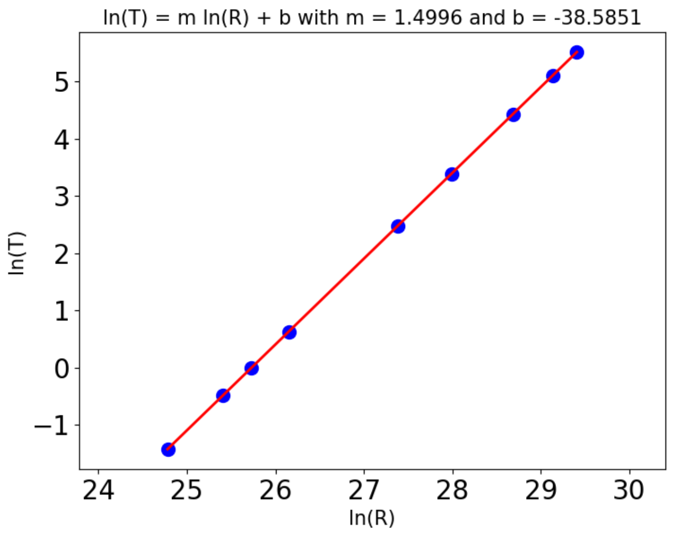


Figure 7.6: Fit of the log-log plot of the data from Table 7.3

What relationship can be inferred from this plot? Well, let's do a little bit of algebra. From the graph of Figure 7.6 it is clear that $\ln(T) = mln(R) + b$. From this relation we can write that $\ln(T) = \ln(R^m) + \ln(e^b) = \ln(e^b R^m)$ or that $T = aR^m$, where a is some constant (notice that the intercept b is absorbed into a). Now, since $m = 1.4996$ we can take it as $m = 3/2$, meaning that each planet period is proportional to the respective radius orbit to the power 3/2, a fact that you have learned when studying Kepler's third Law in your Physics class. We hope you enjoyed this journey!.

The program to make this analysis is available under the name `chap07_prog_02_planets.py` in the directory named `chapter_07` of the programs that comes with this book, that you can download from the respective companion web site mentioned in the Preface of the book. Please, execute this code before continuing via issuing in a system shell or terminal the command:

Chapter 7, System shell command 2

```
$ python chap07_prog_02_planets.py
```

After executing this code you will see on the computer screen a plot similar to that shown in Figure 7.4 which, after eploring each element of the plot, you will need to close to continue execution of the program. This time you will see a plot similar to that shown in Figure 7.5 which, again, after eploring each element of the plot, you will need to close to continue execution of the program. This time you will see a plot similar to that shown in Figure 7.6. The program ends after closing this last figure. Since saving each graph has been enabled, the three shown figures were saved in the current directory. You can see them by executing in a system shell or terminal the command:

Chapter 7, System shell command 3

```
$ ls *png
lnTvslnR_fit.png lnTvslnR.png TvsR.png
```

We are leaving to you to explore the full program on your own.

7.4 Chapter Summary

By reaching the end of this chapter you have done really great! By know you know how to handle via *Python* programming computational operations of your Prealgebra course work involving integers, rational numbers, decimals and basic two dimensional plots. Your knowledge of python is enough for you to cover on your own the many other applications presented in your Prealgebra course work. You could also can go to more advanced books on *Python* programming to take advantage of the splendid capabilities offered by this wonderful programming language. We hope you are ready to joint the trip. Thank you for reading this book.

Exercises of Chapter 7

Exercise 7.1 *Write (and execute) an inspection two dimensional plot program containing only lines* 7, 9, 10, 13 *and* 21 *of the code shown in Figure 7.2, of page 246.*

Exercise 7.2 *Reproduce Figure 7.1b (page 245) by making the change proposed on page 247 in the program of Figure 7.2 (page 246). You will need to comment (or delete) lines of code* 17 *and* 18*.*

References of Chapter 7

Books and/or Articles

- **Devert, A.** (2014). *Matplotlib* Plotting Cookbook, Packt Publishing.
- **Rojas, S.** (2017) Numerical and Scientific Computing with SciPy [Video course]. Packt Publishing.
- **Rojas, S.**, **Christensen, E. A.**, and **Blanco-Silva, F. J.** (2015) Learning SciPy for Numerical and Scientific Computing, Second Edition, Packt Publishing.

References on the WEB

- *Matplotlib* Examples:
  `http://matplotlib.org/1.5.1/examples/index.html`
- *Matplotlib* Faq/How-To:
  `http://matplotlib.org/faq/howto_faq.html`
- *Matplotlib* tutorial:
  `www.labri.fr/perso/nrougier/teaching/matplotlib/`
- *Matplotlib* Legend guide:
  `http://matplotlib.org/users/legend_guide.html`
- *Matplotlib*: plotting
  `http://www.scipy-lectures.org/intro/matplotlib/matplotlib.html`
- Three-dimensional Plotting in *Matplotlib*
  `https://www.oreilly.com/learning/three-dimensional-plotting-in-matplotlib`
- 3D plotting with Mayavi
  `http://www.scipy-lectures.org/packages/3d_plotting/index.html`

- Beautiful plots with Pandas and *Matplotlib*:

  `https://datasciencelab.wordpress.com/2013/12/21/beautiful-plots-with-pandas-and-matplotlib/`

- Overview of *Python* visualization tools:
  `http://pbpython.com/visualization-tools-1.html`

- mpltools: Tools for *Matplotlib*:
  `http://tonysyu.github.io/mpltools/index.html`

- How to make beautiful data visualizations in *Python* with *Matplotlib*:

  `http://spartanideas.msu.edu/2014/06/28/how-to-make-beautiful-data-visualizations-in-python-with-matplotlib/`

- prettyplotlib: Painlessly create beautiful matplotlib plots:
  `http://blog.olgabotvinnik.com/blog/2013/08/21/2013-08-21-prettyplotlib-painlessly-create-beautiful-matplotlib/`

- Plotting data on a map (Example Gallery):
  `http://matplotlib.org/basemap/users/examples.html`

- Visualization: Mapping Global Earthquake Activity:
  `http://introtopython.org/visualization_earthquakes.html`

- SciPy Cookbook:
  `http://scipy-cookbook.readthedocs.org/`

- *Python* scripting for 3D plotting:
  `http://docs.enthought.com/mayavi/mayavi/mlab.html`

- Example gallery:
  `http://docs.enthought.com/mayavi/mayavi/auto/examples.html`

Index