Python from Scratch

Programming for absolute beginners with Python

Nilo Ney Coutinho Menezes

LogiKraft

novatec

Authorized English translation of the Portuguese edition of Introdução à Programação com Python 4ª Edição, ISBN 97875228869 © 2024 Novatec Editora Ltda. This translation is published and sold by permission of Novatec Editora Ltda, the owner of all rights to publish and sell the same. All rights reserved.

This edition is published by LOGIKRAFT SRL under exclusive license from the author, Nilo Ney Coutinho Menezes, with the express authorization of Novatec Editora Ltda, the original publisher and rights holder of the Portuguese edition.

Publisher: Rubens Prates
Translator: Nilo Ney Coutinho Menezes
Proofreader: Ariane L. Smith and Kelsey Yurek
Cover designer: Olinart
Cover illustration: Olinart

Paperback: 978-85-7522-949-1

Hardcover: 978-85-7522-950-7

Ebook: 978-85-7522-951-4

Novatec Editora Ltda.
Rua Luís Antônio dos Santos 110
02460-000 – São Paulo, SP – Brasil
Tel.: +55 11 2959-6529
Email: *novatec@novatec.com.br*
Site: *https://novatec.com.br*
Twitter: *twitter.com/novateceditora*
Facebook: *facebook.com/novatec*
LinkedIn: *https://linkedin.com/company/novatec-editora/*

LogiKraft SRL
Rue de la Grande Campagne, 40
7340 – Colfontaine
Belgium
Email: *contact@logikraft.be*
Website: *https://logikraft.be*

DISCLAIMER

Every effort has been made to ensure that the information provided in this book is as accurate and complete as possible. However, such information is provided "as is" and without warranty of any kind, either expressed or implied. The author, publisher, distributors, and any entity directly or indirectly involved in its dealings assume no liability whatsoever for any loss or damage, direct or indirect, arising from the information contained in this book.

Table of Contents

Acknowledgments

This book wouldn't be possible without the encouragement of my wife, Emília Christiane, and the understanding of my children, Igor, Hanna, and Iris. Writing in summer isn't always easy for those who live in a cold country like Belgium.

I am also grateful for the support I received from my parents and grandparents during my studies and for their valuable advice, understanding, and guidance.

I couldn't forget my brother, Luís Victor, for his help with the images in Portuguese.

Thanks to my wife Chris and my daughter Hanna for their help reviewing the book, asking about and marking errors, reading and rereading until the text is ready to be revised again.

Luciano Ramalho and colleagues from the python-brasil list. Luciano, thank you for the encouragement to publish this book and for the more than pertinent comments. To the python-brasil community for their efforts and proof of civility in keeping the discussions at the highest level, pleasing beginners, curious people, and computer professionals.

Not to forget the team from Editora Novatec, with whom I have been working for over ten years and who never cease to amaze me in terms of their professionalism and seriousness, operating in a competitive and globalized market without ever losing the conviction to publish technical works in Portuguese.

Thanks also to colleagues, friends, and students at the Matias Machline Foundation, where I had the opportunity to study and work as a teacher of programming logic. Thank you to friends and colleagues from the La Salle Educational Center and the Paulo Feitoza Foundation (FPF Tech), where I taught programming logic and Python courses for many years.

Preface

I learned to program using the BASIC language back in the mid-1980s. I remember building small drawing programs, phone books, and games. Data storage was only available on cassette tapes. Before the internet and living in northern Brazil, learning to program involved reading books and, of course, programming. The most common way to obtain new programs was through programming magazines, which were dedicated to the new community of microcomputer users, a term used at the time to differentiate home computers from bigger commercial alternatives, called mainframes or mini-computers, used by big companies and costing thousands of dollars. They included complete listings of programs written in BASIC or *Assembly*. At a time when downloading was barely a distant dream, typing these programs was the only solution to run them. The experience of reading and typing the programs was essential for learning to program, but unfortunately, few technical magazines today are accessible to beginners. The complexity of today's programs is also much much greater, requiring more time studying than in the past. While the internet greatly helps, following a planned learning order is very important.

When starting to teach programming, the challenge was always finding a book that students could read in high school or early in higher education. Although several works fulfilled this need, handouts were always necessary since the order in which the new concepts were presented was almost always more designed for a dictionary than for teaching programming. Concepts important to the beginner were forgotten entirely, and a greater focus was given to more complex subjects.

Following an idea that my high school teachers presented and used, programming logic is more important than any language, so those who learn to program once are better equipped to learn other programming languages. This book focuses on presenting Python resources whenever possible. The purpose is to initiate the reader into the world of programming and prepare them for more advanced courses and concepts. After reading and studying this book, I believe you will be able to read other programming texts and learn new languages on your own.

More than thirteen years have passed since the first Brazilian edition of this book. The book was written to help people learn to program independently at home, with little or no help. The task is not simple; each person has a different experience when learning to program. One of the factors that I find most interesting is that difficulties occur at different times and places. Each revision and new edition reflects the comments I collect in various Internet groups and when talking to readers of the book, especially those who send questions by email. Without this exchange of experiences, it would be even more challenging to improve and correct the book.

The Python language grew tremendously during that period, and the easy and powerful nature of the language has been confirmed several times. Today, Python is highly regarded in academic, scientific, and professional circles. The purpose of this book has always been to teach programming, with Python as a valuable bonus gained in the process. I believe that the rules of programming are independent of the language itself. Still, when choosing Python, I knew that it would be a great investment of the reader's time, and they would simultaneously learn new techniques and a language that they could continue using after finishing the book.

As time has gone on, the book has become more Pythonic. The goal is to first show the concept without using everything from Python a few times. As you move toward the end of each chapter, you will encounter programs that increasingly use the resources of the language. Some sections have been revised to contain more details about mathematics and links to get more information. The same was done with more advanced resources and formal aspects of computing that would not fit in this book but are mentioned to allow the reader to continue learning more complex concepts in other texts.

No book or course can teach you everything about programming since it is a vast area that continues to expand rapidly. Thus, this book aims to create a solid base for programming students, enabling them to read other texts and select other courses that they want to take. Learning programming is an infinite process, and even though I have been programming for over 30 years, I continue to learn every day. My objective remains the same: to teach how to program with Python.

I hope you continue to send your questions, suggestions, and critiques by email or Telegram. Do not hesitate to comment if you like the changes or if anything is unclear.

Introduction

This book was written with the programming beginner in mind. Although the Python language is very powerful and rich in modern programming resources, this work aims to teach the logic and programming techniques needed to program in any language. Some Python language resources have not been used or have been postponed in favor of programming logic exercises. The goal was to prioritize these exercises and better prepare the reader for other languages. That choice didn't stop powerful language features from being presented, but this book doesn't intend to cover everything about Python.

The chapters are organized in such a way as to present the basic programming concepts progressively. It is highly recommended that you read the book near a computer with the Python interpreter open to facilitate experimentation with the proposed examples. Some readers have found success with Python interpreters installed on their cell phones, but a computer is recommended.

Each chapter includes exercises organized to explore the content presented. Some exercises only modify the examples in the book, while others require the application of the concepts presented to create new programs. Try to solve the exercises as they are given. While it's impossible not to talk about math in a programming book, the exercises are designed for a high school student's level of knowledge and use business or everyday problems. This choice was not made to avoid studying mathematics but to prevent mixing the introduction of programming concepts with new mathematical concepts. It is highly recommended that the reader solve each exercise before moving on to the next chapter.

Organizing the generated programs in one folder (directory) per chapter is recommended, preferably by adding the example or exercise number to the file names. Some exercises alter others, even in different chapters. Ensuring these files are well organized will facilitate your study.

Appendix A has been prepared to help understand the error messages that the Python interpreter can generate. Use it whenever you find new errors in your programs. With practice, you will learn how to recognize these errors and locate them in your programs.

Python also allows students and teachers to use the operating system of their choice, whether Windows, Linux, macOS, or those found on even mobile phones. All examples in the book require only the standard language distribution, which is available free of charge.

While every effort has been made to avoid errors and omissions, the book is not guaranteed to be error-free. If you find flaws in the content, please email `errors@pythonfromscratch.com`. If you have any questions, although I can't guarantee a response to every email, please send your message to `questions@pythonfromscratch.com`. Tips, critiques, and suggestions can be sent to `teachers@pythonfromscratch.com`. The source code, solved exercises, videos, and possible corrections to this book can be found at *https://pythonfromscratch.com*.

The author is also available via Telegram on the channel dedicated to the book (*https://t.me/pythonfromscratchbook*), but the response time may vary depending on the author's free time; don't expect immediate answers, except from other readers on the same channel.

A summary of each chapter's content is presented below:

- **Chapter 1 — Motivation**: Aims to present the challenge of learning and stimulating the study of computer programming, presenting everyday problems and applications.
- **Chapter 2 — Preparing the environment**: Installation of the Python interpreter, introduction to the text editor, presentation of IDLE, execution environment, how to type programs, and how to do the first tests with arithmetic operations in the interpreter. Optionally, you can also install Visual Studio Code to type your programs. A quick review of mathematical concepts has been included in this chapter to facilitate understanding and solving the exercises.
- **Chapter 3 — Variables and data entry**: Types of variables, properties of each type, operations, and operators. It introduces the concept of a program over time and a simple debugging technique. Keyboard data entry, data type conversion, and common errors.
- **Chapter 4 — Conditions**: Conditional structures, the concept of the block, and the selection of lines to execute based on evaluating logical expressions.
- **Chapter 5 — Repetitions**: **`while`** loop, counters, and accumulators. The execution of a block and nested loops.
- **Chapter 6 — Lists, dictionaries, tuples, and sets**: Operations with lists, sorting using the bubble sort method, searching, using lists as stacks and queues. Examples of using dictionaries, tuples, and sets.
- **Chapter 7 - Working with strings**: Presents advanced operations with strings. Explores the Python string class. The chapter also includes a simple game to reinforce the concepts of string manipulation.
- **Chapter 8 — Functions**: Notion of function and transfer of execution flow, recursive functions, **`lambda`** functions, parameters, and modules. It presents random numbers.
- **Chapter 9 — Files**: Creating and reading files on disk. Generation of HTML files in Python, operations with files and directories, command-line parameters, paths, and manipulation of dates and times. Creating and reading JSON files.
- **Chapter 10 - Classes and objects**: Introduction to object orientation. Explains the concepts of class, objects, methods, and inheritance. It prepares the student to continue studying the topic and better understand it.
- **Chapter 11 — Database**: Introduction to the SQL language and the SQLite database.
- **Chapter 12 — Patterns**: Pattern recognition, regular expressions, and Python's **`match`** statement.

- **Chapter 13 — Graphical interface**: Introduction to programming graphical interfaces with `tkinter`. Create simple interfaces with buttons, a drawing utility, and a site manager.
- **Chapter 14 — Next steps**: The final chapter lists the next steps in various topics, such as games, web systems, functional programming, graphical interfaces, and databases. It aims to present books and open-source projects that students can use to continue studying, depending on their area of interest.
- **Appendix A — Error messages**: Explains Python's most frequent error messages, showing their causes and how to resolve them.
- **Appendix B — Bitwise operators**: Supplementary material for computer science and electronic engineering students interested in bit manipulation in Python.

Contacts and groups

By email:

errors@pythonfromscratch.com

questions@pythonfromscratch.com

teachers@pythonfromscratch.com

Telegram:

https://t.me/pythonfromscratchbook

Book's website

Address: *https://pythonfromscratch.com*

The book's website contains all the book listings as well as all the solved exercises. You will also find videos and possibly corrections (errata).

CHAPTER 4
Conditions

To execute or not to execute? That's the question.

Here's the thing...

Not every line of our program will be executed. It will often be more interesting to decide which parts of the program should be executed based on the result of a condition. The basis of these decisions will consist of logical expressions that allow us to represent decisions in our programs.

The conditional execution of parts of the program is one of the fundamentals of computer programming. These conditional structures allow your program to work differently, depending on the values entered by the user.

In this chapter, you will encounter programs that are not executed in the same sequence as their lines. We will see how to skip some parts, depending on values and conditions. Logical expressions will represent conditions and decisions that must be made before executing a part of the program. The conditional structures we'll examine in this chapter will allow your program to react and produce different outputs for different data inputs.

4.1 if

The conditions serve to select when a part of the program should be activated and when it should simply be ignored. In Python, the conditional structure is the **`if`**. Its format is presented below:

```
if <condition>:
    True block
```

`if` is nothing more than our "if" as in English. We can then understand it as follows: if the condition is **`True`**, do something. Remember that condition is a logical expression, as we saw in Chapter 3.

Let's look at an example of a program that reads two values and prints which one is larger:

```
# Program 4.1: Reads two values and prints which one is the highest
a = int(input("First value: "))
b = int(input("Second value: "))
if a > b: ❶
    print("The first value is the highest!") ❷
if b > a: ❸
    print("The second value is the highest!") ❹
```

In ❶, we have the condition `a > b`. This logical expression will be evaluated, and if its result is `True`, line ❷ will be executed. If **`False`**, line ❷ will be ignored. The same is true for condition `b > a` on line ❸. If your result is **`True`**, line ❹ will be executed. If it's **`False`**, it will be ignored.

The program execution sequence changes according to the values entered as the first and second values. Type in Program 4.1 and run it twice. The first time, type a larger value first and a smaller value after it. The second time, invert these values and verify that the message on the screen also changed.

When the first value is greater than the second, we have the following lines running: ❶, ❷, ❸. When the first value is smaller than the second, we have another sequence: ❶, ❸, ❹. It is important to understand that the line with the condition itself with the **`if`** is executed even when the result of the expression is **`False`**.

Let's see how this program behaves. The executed lines are marked with a gray background.

Figure 4.1 shows the execution (on a gray background) when `a` is smaller than `b`. The code with the white background was not executed. In the example, `a` is `1` and `b` is `3`. In this case, the second value is the largest.

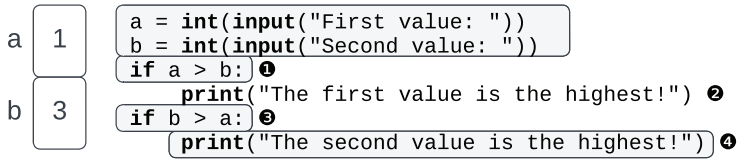

Figure 4.1: Execution example (in gray) when a is smaller than b

TRIVIA

Graphical code representation can help with understanding algorithms; a long time ago, we used flowcharts to learn how to program. Flowcharts are graphical representations of a program. They worked relatively well in "unstructured" programming when a program's execution could jump to any line. In the 1970s and 1980s, structured programming (with blocks and functions) became more used, and flowcharts were left out. A flowchart offers a lot of freedom when creating a program since you can pass lines to any part of the program. Structured programming organizes the program in blocks, as we did with **`if`** and does not allow jumps from one part of the program to another, except when following well-established rules.

To learn more about structured programming: *https://en.wikipedia.org/wiki/Structured_programming*.

And what happens when `a` is greater than `b`?

Figure 4.2 shows the lines executed when a is greater than b. Notice that now ❷ is executed but ❹ is not.

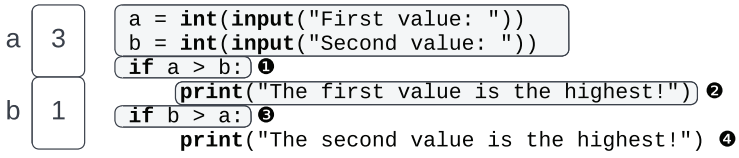

Figure 4.2: Execution example (in gray) when a is greater than b

Notice that lines ❶ and ❸ ended with the colon (:). When that happens, we have the announcement of a block of lines to follow. In Python, a block is represented by moving the beginning of the line to the right. The block continues to the first line with a different offset. To move the text to the right, use spaces, usually four spaces for each level of indentation. See Figure 4.3 for the two blocks of code that we created by advancing the text to the right.

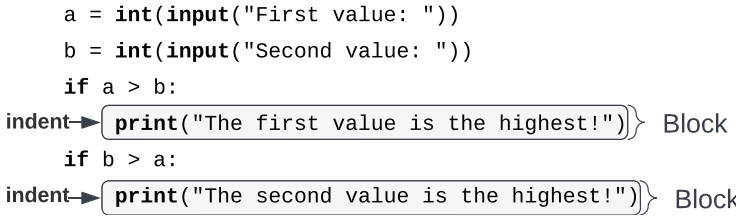

Figure 4.3: Blocks and text feed

Notice that we started writing line ❷ a few more characters to the right of the previous line ❶ that started the block. As line ❸ was written farthest to the left, we say that the block of line ❷ has been completed.

TRIVIA

Python is one of the few programming languages that marks the beginning and end of a block by moving text to the right (indentation). Other languages have special words for this, such as BEGIN and END in Pascal or the famous curly braces ({ and }) in C and Java. While offsetting to the right using white spaces makes our code more elegant, typing our programs requires greater care.

Exercise 4.1 Analyze Program 4.1. What happens if the first and second values are the same? Explain.

Let's look at another example, where we will request the age of the user's car and then write new (if the car is less than three years old) or, otherwise, old.

```
# Program 4.2: New or old car, depending on age
age = int(input("Enter the age of your car: "))
if age <= 3:
    print("Your car is new") ❶
if age > 3:
    print("Your car is old") ❷
```

The first condition is `age <= 3`. This condition decides whether the line with the **`print`** ❶ function will be executed. As it is a simple condition, we can understand that we will only display the new car message for ages 0, 1, 2, and 3. The second condition, `age > 3`, is the inverse of the first. If you look closely, there isn't a single integer that would make both true simultaneously. The second decision is responsible for deciding to print the old car message ❷. Run Program 4.2 and check what appears on the screen. You can run it multiple times with the following ages: 1, 3, and 5.

While it's obvious that a car can't have negative values for `age`, the program doesn't address that issue. We'll change it later to check for invalid values.

Exercise 4.2 Write a program that asks the speed of a user's car. If it exceeds 80 km/h, display a message stating that the user has been fined. In this case, show the amount of the fine, charging $5 per km above 80 km/h.

A block in Python can have more than one line of code though the last example shows only two blocks with one line in each. If you need two or more lines in the same block, write those lines in the same direction or the same column as the first row of the block.

4.1.1 Income tax calculation example

A common problem is when we must pay income tax. Normally, we pay income tax by salary range. Imagine that, for salaries lower than $1,000, we would have no tax to pay, a 0% rate. For salaries between $1,000 and $3,000, we would pay 20%. Above these amounts, the rate would be 35%. This problem would be very similar to the previous one unless the tax was not charged differently for each band. That is, those who earn $4,000 have the first $1,000 exempt from tax, 20% paid on the amount between $1,000 and $3,000, and 35% paid on the rest. Let's look at the solution:

```
# Program 4.3: Income tax calculation
salary = float(input("Enter the salary for tax calculation: "))
base = salary ❶
tax = 0
if base > 3000: ❷
    tax = tax + (base - 3000) * 0.35 ❸
    base = 3000 ❹
if base > 1000: ❺
    tax = tax + (base - 1000) * 0.20 ❻
print(f"Salary: ${salary:6.2f} Tax payable: ${tax:6.2f}")
```

Program 4.3 is very interesting. Try running it a few times and compare the printed amount with the amount you calculated. Trace the program and try to understand what it does before reading the next paragraph. Check what happens for salaries of $500, $1,000, and $1,500.

In ❶, we have the `base` variable receiving a copy of the `salary`. This is necessary because when we assign a new value to a variable, the previous value is replaced (and lost if we don't store it elsewhere). Since we are going to use the amount of the salary entered to display it on the screen, we cannot lose it; therefore, an auxiliary variable called `base` is needed here.

In ❷, we verify that the `base` is greater than $3,000. If true, we execute lines ❸ and ❹. In ❸, we calculated 35% of the amount greater than $3,000. The result is stored in the `tax` variable. As

this variable contains the amount to be paid for this amount, we will update the base amount to $3,000 ❹ since what exceeds this amount has already been charged.

In ❺, we verify that the `base` amount is greater than $1,000, calculating 20% tax in ❻, if `True`.

Let's look at the tracing for a salary of $500:

salary	base	tax
500	500	0

For a salary of $1,500:

salary	base	tax
1500	1500	~~0~~
		100

For a salary of $3,000:

salary	base	tax
3000	3000	~~0~~
		400

For a salary of $5,000:

salary	base	tax
5000	~~5000~~	~~0~~
	3000	~~700~~
		1100

Exercise 4.3 Write a program that reads three numbers and prints the largest and the smallest.

Exercise 4.4 Write a program that asks for the employee's salary and calculates the amount of the rise. For salaries above $1,250, calculate a raise of 10%. For those equal or lower, 15%.

4.1.2 Example of calculating a cell phone bill

The following program calculates the monthly fee for a cell phone operator called Bye. The operator in question has only two plans: LittleTalk and TalkMore. Under the LittleTalk plan, the consumer is entitled to 100 minutes, which are included in the $50 monthly plan. Each extra minute (in addition to those included) costs $0.20. In the TalkMore plan, 500 minutes are included for $99 and each extra minute costs $0.15.

Let's look at a program that asks for the plan and the number of minutes consumed. With this information and using what we just discovered about the plans, the program will calculate and print the price to pay. Note that if the plan is neither LittleTalk nor TalkMore, the program does not calculate any price.

```
# Program 4.4: Calculating the monthly fee for a cell phone plan from the operator Bye
plan = input("What's your cell phone plan? ")
if plan == "LittleTalk":
    minutes_on_plan = 100
    extra = 0.20
    price = 50
if plan == "TalkMore":
    minutes_on_plan = 500
    extra = 0.15
    price = 99
if plan != "LittleTalk" and plan != "TalkMore":
    print("I don't know this plan")
if plan == "LittleTalk" or plan == "TalkMore":
    minutes_consumed = int(input("How many minutes did you consume? "))
    print("You will pay:")
    print(f"Plan price ${price:10.2f}")
    supplement = 0
    if minutes_consumed > minutes_on_plan:
        supplement = extra * (minutes_consumed - minutes_on_plan)
    print(f"Supplement ${supplement:10.2f}")
    print(f"Total      ${price + supplement:10.2f}")
```

Let's look at the blocks highlighted in Figure 4.4:

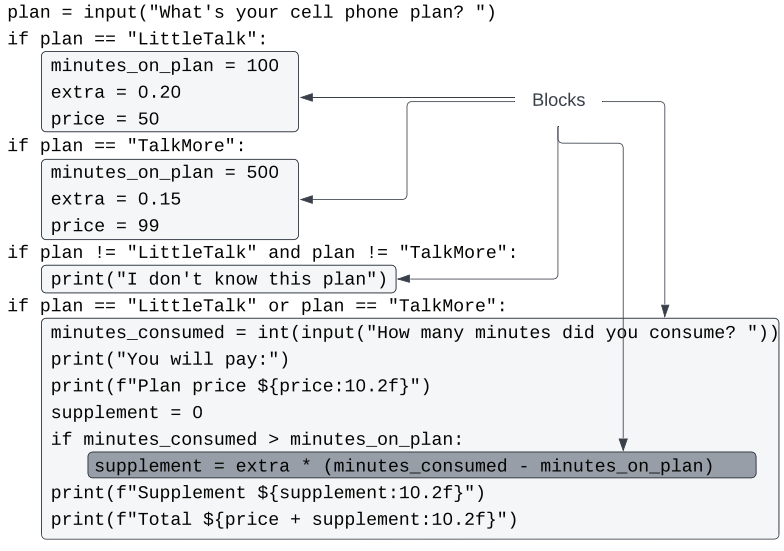

Figure 4.4: Featured blocks

Notice that the line that calculates the `supplement` is a block within another block. When one structure appears inside another, we say that they are nested. Python allows you to create blocks within blocks freely. That way, an **`if`** can contain other **`if`** structures and other statements without any problems.

Also, note that the lines outside the block are aligned with each other. The first four **`if`** all start in the same column, and all the rows in each block are also aligned with each other.

The rule is that a shift to the right creates a block (usually after an instruction that ends with a colon). A shift to the left marks the end of the block.

4.2 else

When there are problems, such as the old car message (Program 4.2), in which the second condition is simply the inverse of the first, we can use another form of **`if`** to simplify the programs. This form includes an **`else`** clause to specify what to do if the result of evaluating the condition is **`False`**, without needing a new **`if`**. Let's see what the program rewritten to use **`else`** would look like (Program 4.5):

```
# Program 4.5: New or old car, depending on its age
age = int(input("Enter the age of your car: "))
if age <= 3:
    print("Your car is new")
else: ❶
    print("Your car is old") ❷
```

Notice that, in ❶, we use "**`:`**" after **`else`**. This is necessary because **`else`** starts a block in the same way as **`if`**. It's important to note that we must write **`else`** in the same column as **`if`**, that is, with the same indent. Thus, the interpreter recognizes that **`else`** refers to a given **`if`**. You will get an error if you don't align these two structures in the same column.

The advantage of using **`else`** is making programs more straightforward since we can express what to do if the condition specified in **`if`** is **`False`**. Line ❷ is only executed if the condition `age <= 3` is **`False`**.

Let's look at another example with the larger of the two numbers. Program 4.6 but rewritten to use **`else`**:

```
# Program 4.6: Reads two values and prints which one is larger using an else
a = int(input("First value: "))
b = int(input("Second value: "))
if a > b:
    print("The first value is the largest!")
else:
    print("The second value is the largest!")
```

Although equivalent, the second program is easier to read and uses only one condition. Another advantage is that we didn't have the problem in the case where `a` is equal to `b` since we only tested one condition, and if it's **`False`**, we executed the **`else`**'s code. In Program 4.1, the condition had a flaw and didn't cover all cases where the first condition was **`False`**.

Exercise 4.5 Run Program 4.5 and try some values. Check that the results are the same as in Program 4.2.

Exercise 4.6 Write a program that asks the distance a passenger wishes to cover in kilometers. Calculate the ticket price, charging $0.50 per km for trips up to 200 km and $0.45 for longer trips.

Exercise 4.7 Analyze Program 4.3. Does using **else** in that program make sense? Explain your answer.

Exercise 4.8 Rewrite Program 4.4 and calculate the Bye operator account using **else**.

4.3 Nested structures

Our programs will not always be so simple. Often, we'll need to nest multiple **if**s to get the desired behavior from the program. To nest, in this case, is to use one **if** inside another.

Let's look at another example of calculating the bill for a cell phone from the operator Bye. Bye company plans are very interesting and offer different prices according to the number of minutes used per month. For less than 200 minutes, the company charges $0.20 per minute. Between 200 and 400 minutes, the price is $0.18. For more than 400 minutes, the price per minute is $0.15. Program 4.7 solves this problem:

```
# Program 4.7: Phone bill with three price ranges
minutes = int(input("How many minutes did you use this month:"))
if minutes < 200: ❶
    price = 0.20 ❷
else:
    if minutes < 400: ❸
        price = 0.18 ❹
    else: ❺
        price = 0.15 ❻
print(f"You will pay this month: ${minutes * price:6.2f}")
```

In ❶, we have the first condition: `minutes < 200`. If the number of minutes is less than `200`, we assign `0.20` to the price in ❷. So far, nothing new. Notice that the **if** of ❸ is inside the **else** from the previous line: it's nested inside the **else**. The condition of ❸, `minutes < 400`, decides whether we are going to execute line ❹ or line ❻. Notice that **else** from ❺ is aligned with **if** from ❸. At the end, we calculate and print the price on the screen. Remember that text alignment is very important in Python.

Take, for example, a situation in which five categories are needed. Let's make a program that reads a product's category and determines the price using Table 4.1.

Table 4.1: Product and price categories

Category	Price
1	10.00
2	18.00
3	23.00
4	26.00
5	31.00

The program line numbers, numbered from 1 to 19, are in the leftmost column of Program 4.8. These numbers only serve to help you understand the following explanation — remember not to type them.

```
# Program 4.8: Category x price
1 category = int(input("Enter the product category:"))
2 if category == 1:
3      price = 10
4  else:
5      if category == 2:
6          price = 18
7      else:
8          if category == 3:
9              price = 23
10         else:
11             if category = 4:
12                 price = 26
13             else:
14                 if category == 5:
15                     price = 31
16                 else:
17                      print("Invalid category, enter a value between 1 and 5!")
18                     price = 0
19 print(f"Product price is: ${price:6.2f}")
```

Notice that alignment became a big problem since we had to move each **else** to the right.

In Program 4.8, we introduced the concept of input validation. This time, if the user enters an invalid value, they will receive an error message on the screen. Nothing very practical or beautiful, but for now, it's enough for us to know that we've entered an invalid value.

Let's see the execution of the lines depending on the category entered in Table 4.2.

Table 4.2: Rows executed

Category	Lines executed
1	1, 2, 3, 19
2	1, 2, 4, 5, 6, 19
3	1, 2, 4, 5, 7, 8, 9, 19
4	1, 2, 4, 5, 7, 8, 10, 11, 12, 19
5	1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 15, 19
others	1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 18, 19

When we read a program with nested structures, we must pay close attention to view the blocks correctly. Notice how important alignment is.

Exercise 4.9 Trace Program 4.8. Compare your result to that shown in Table 4.2.

4.4 elif

Python presents a very interesting solution to the problem of multiple nested **if**. The **elif** clause replaces an **else if** pair, but without creating another level of structure, avoiding unnecessary displacement to the right problems.

Let's revisit Program 4.8 — this time using **elif**. See the result in Program 4.9:

```
# Program 4.9: Category x price, using elif
category = int(input("Enter the product category:"))
if category == 1:
    price = 10
elif category == 2:
    price = 18
elif category == 3:
    price = 23
elif category == 4:
    price = 26
elif category == 5:
    price = 31
else:
    print("Invalid category, enter a value between 1 and 5!")
    price = 0
print(f"Product price is: ${price:6.2f}")
```

Let's look at Program 4.10, which calculated the account price on the LittleTalk and TalkMore plans, rewritten to use **elif**:

```
# Program 4.10: Bye plans with elif
valid = True
plan = input("What's your cell phone plan? ")
if plan == "LittleTalk":
    minutes_on_plan = 100
    extra = 0.20
    price = 50
elif plan == "TalkMore":
    minutes_on_plan = 500
    extra = 0.15
    price = 99
else:
    valid = False
if not valid:
    print(f"Error: I don't know this plan {plan}")
else:
    minutes_consumed = int(input("How many minutes did you consume? "))
    print("You will pay:")
    print(f"Plan price ${price:10.2f}")
    supplement = 0
    if minutes_consumed > minutes_on_plan:
        supplement = extra * (minutes_consumed - minutes_on_plan)
    print(f"Supplement ${supplement:10.2f}")
    print(f"Total      ${price + supplement:10.2f}")
```

In Program 4.10, we added a `valid` variable to indicate whether the plan is known or not. This avoids repeating the condition in the second **`if`** and makes the program more prepared if the company launches other plans. We used an **`else`** at the end of the first **`if`** — **`elif`** to say that the plan is invalid or `valid == False`. In the second **`if`**, if the plan is unknown, we display an error message and we don't calculate the price. Other than that, the program does the same thing as Program 4.4 but uses **`if`**-**`elif`**-**`else`**. An **`if`** can have multiple **`elif`** but only one **`else`**.

Exercise 4.10 Write a program that reads two numbers and asks what operation you want to perform. You must be able to calculate sum (+), subtraction (-), multiplication (*), and division (/). Display the result of the requested operation.

Exercise 4.11 Write a program to approve a bank loan for the purchase of a home. The program must ask the price of the house to buy, the salary, and the number of years to pay. The amount of the monthly installment cannot exceed 30% of the salary. Calculate the installment as the amount of the house to be purchased divided by the number of months to pay.

Exercise 4.12 Write a program that calculates the price to pay for electricity. Ask the amount of kWh consumed and the type of installation: R for residential, I for industrial, and C for commercial. Calculate the price to pay according to the following table.

Price per type and consumption range		
Type	Range (kWh)	Price
Residential	Up to 500	$ 0.40
	Over 500	$ 0.65
Commercial	Up to 1000	$ 0.55
	Over 1000	$ 0.60
Industrial	Up to 5000	$ 0.55
	Over 5000	$ 0.60

4.5 Inversion of conditions

We have discussed inverse conditions several times, but to make it clearer, let's build a table.

Table 4.3: Inverting conditions

Condition	Inverse
A	**`not`** A
A > B	A <= B
A < B	A >= B
A == B	A != B
A **`and`** B	**`not`** (A **`and`** B) **`not`** A **`or`** **`not`** B
A **`or`** B	**`not`** (A **`or`** B) **`not`** A **`and`** **`not`** B

While you don't have to invert conditions in most cases, knowing the inverse of a condition helps you understand problems and errors in your programs. The negation (**not**) operator can be used in all cases where it is necessary to invert a logical result. Remember that the **not** operator has a higher priority, and if your condition has several **and**, **or**, it will probably need to be placed in parentheses to be negated correctly. In some programs, it may be interesting to write the **else** code instead of the **if** code block. This is possible by inverting the condition.

Exercise 4.13 In the following program, invert the **if** and **else** lines, negating the condition. Add the necessary lines to make it work in Python.

```
if a > b:
    print("a is greater than b")
else:
    print("b is greater than a")
```

Exercise 4.14 Rewrite the following program with **if-elif-else**. Add the necessary lines to make it work in Python.

```
if a < 10:
    print("a is less than 10")
if a >= 10 and a < 20:
    print("a is greater than 10 and less than 20")
if a >= 20:
    print("a is greater than 20")
```

Exercise 4.15 Rewrite the following program with **if-elif-else**.

```
time = int(input("Enter the current time:"))
if time < 12:
    print("Good morning!")
if time >=12 and time <=18:
   print("Good afternoon!")
if time >= 18:
    print("Good evening!")
```

Exercise 4.16 Correct the following program:

```
score = input("Enter your exam scores:")
if score < 4:
    print("Unfortunately you didn't pass")
if score < 7:
    print("You need to resit the exam")
if score > 7:
    print("You passed the year")
```

4.6 Note of caution when comparing values

When we compare variables of different types, we may have some surprises. For example:

```
>>> "1" == 1
False
```

The first "1" is a string and the second is an integer. The result will always be different.

```
>>> 1.0 == 1
True
```

Although the first 1.0 is of the **`float`** type and the second is of the integer type, Python automatically does the type conversion for you, and the result is **`True`**.

```
>>> "A" == "a"
False
```

Uppercase and lowercase letters are always different.

```
>>> " A " == "A"
False
```

Blank spaces also alter the comparison. Notice that the first " A " has blank spaces before and after the letter, and the second has no spaces.

```
>>> 10 / 3 == 3.33
False
```

Although similar, the result of 10 / 3 is a different number with several 3s after the comma.

```
>>> 10 / 3 == 3.333333
False
```

Even with more digits added, it's still different.

Let's see what 10 / 3 looks like when represented as a floating-point number:

```
>>> 10 / 3
3.3333333333333335
>>> 10 / 3 == 3.3333333333333335
True
```

Who would have guessed that, after so many 3 digits, we would have a 5? That's a surprise when using floating-point numbers. You will gradually get used to these minor quirks and imperfections, but don't be alarmed if the result is slightly different than expected.

```
>>> 10 // 3 == 3
True
```

A `//` does the integer division, so the result is 3.

```
>>> 10 / 2 == 5
True
>>> 10 / 2
5.0
```

As for the case of `10 / 2`, since it is an exact division, even if the result is of the **`float`** type, Python can make the comparison with an integer.

TRIVIA

Python has a function called `isclose` in the `math` module that allows the approximate comparison of floating-point numbers.

```
>>> import math
>>> math.isclose(10 / 3, 3.3, rel_tol=0.1)
True
>>> math.isclose(10 / 3, 3.1, rel_tol=0.1)
True
>>> math.isclose(10 / 3, 3.2, rel_tol=0.1)
True
>>> math.isclose(10 / 3, 3.0, rel_tol=0.1)
False
>>> math.isclose(10 / 3, 3.3, rel_tol=0.01)
False
>>> math.isclose(10 / 3, 3.33, rel_tol=0.01)
True
>>> math.isclose(10 / 3, 3.4, rel_tol=0.1)
True
```

The optional `rel_tol` parameter is the accepted relative difference between the two numbers. In this case, 0.1 represents 10%, 0.01, 1%, etc. The default value is 10-9 (1e-09 in scientific notation). Don't worry now about **`import`** or what an optional parameter is; it will be explained later in Chapter 8.

In Chapter 7, we will see more functions for cleaning and treating data entered by the user, such as removing blank spaces and converting everything to lowercase letters. Until then, try to input data in your programs to avoid generating unnecessary errors. Remember, we are just getting started. When an error occurs, try to understand its cause. It will help you create the validation code in the following chapters.