

Coffee Break Python

Slicing



24

Workouts to Master Slicing in
Python, Once and for All

CHRISTIAN MAYER

Coffee Break Python *Slicing*

24 Workouts to Master Slicing
in Python, Once and For All

Christian Mayer

November 2018

A puzzle a day to learn, code, and play.

Contents

Contents	ii
1 Introduction	1
2 How to Use This Book	7
2.1 What Is a Code Puzzle? . . .	8
2.2 Why Puzzle-based Learning? .	8

2.3	How to Measure Your Python Skills with This Book?	13
-----	---	----

3	A Quick Overview of the Python Language	23
----------	--	-----------

3.1	Keywords	25
3.2	Basic Data Types	29
3.3	Complex Data Types	33
3.4	Classes	38
3.5	Functions and Tricks	42

4	Introduction to Slicing	47
----------	--------------------------------	-----------

4.1	Intermezzo: Indexing Basics . .	48
4.2	Slicing	49
4.3	The Step Size in Slicing	52
4.4	Overshooting Indices in Slicing	54
4.5	Summary Basic Python Slicing	56
4.6	Frequently Asked Questions . .	59

5	Code Puzzles Slicing	87
----------	-----------------------------	-----------

6 Final Remarks

101

— 1 —

Introduction

The main driver for mastery is neither a character trait nor talent. Mastery comes from intense, structured training. The author Malcolm Gladwell formulated the famous rule of 10,000 hours based on research from psychology and neurological science.¹ The rule states

¹Malcolm Gladwell *Outliers: The Story of Success*.

that if you have average talent, you will reach mastery in any discipline by investing approximately 10,000 hours of intense training. Bill Gates, the founder of Microsoft, reached mastery at a young age as a result of coding for more than 10,000 hours. He was committed and passionate about coding and worked long nights to develop his skills.

If you are reading this book, you are an aspiring coder and you seek ways to advance your coding skills. You already have some experience in writing code, but you feel that there is a lot to be learned before you become a master coder. You want to read and understand code better. You want to challenge the status quo that some of your peers understand code faster than you. Or you are already proficient with another programming language like Java or C++ but want to learn Python to be-

come more valuable to the marketplace. Either way, you have already proven your ambition to learn and, therefore, this book is for you. To join the league of the great code masters, you only have to do one thing: stay in the game.

There is one thing that will empower *you* to invest the 10,000 hours of hard, focused work to reach mastery. It's *your ambition to learn* that will drive you through the valleys of desperation on your path to mastery: complex code, nasty bugs, and project managers pushing tight deadlines. Nurturing your ambition to learn will pay a rich stream of dividends to you and your family for as long as you live.

This book aims to be a stepping stone on your path to becoming a Python master. It focuses on one important subtopic of the Python

programming language: *slicing*. It offers you five to ten hours of thorough Python training using a scientifically proven learning technique, called *practice testing*. Investing this time will kickstart your skills to write, read, and understand Python source code.

The learning system used in this book is simple: you read the material from the first to the last page. As you go along, you solve code puzzles at various levels of difficulty. In essence, you play Python interpreter and compute the output of a code snippet in your head. Then you check whether you were right—using feedback and explanations—to adapt and improve your coding skills over time. This system has proven highly effective for teaching more than 21,000 online students on my online coding academy <https://finxter.com>.

Book Overview

Chapter 2 explains and motivates the advantages of the Finxter method of puzzle-based learning and shows how to use the chess rating method (Elo) to measure your Python skill level. Chapter 3 gives you a quick overview of the Python programming language. Chapter 4 is an in-depth tutorial about slicing in Python. Feel free to jump ahead and start reading from this chapter, if you feel confident with Python and puzzle-based learning. Chapter 5 strengthens your Python slicing skills by posing you 11 extra Python puzzles. Finally, Chapter 6 concludes the book.

— 2 —

How to Use This Book

This chapter shows you how to read this book. If you want to test your skill level to get an accurate estimate of your Python skills compared to others, read on. If you don't, go directly to the introduction to slicing in Chapter 4.

2.1 What Is a Code Puzzle?

Definition: A *code puzzle* is an educative snippet of source code that teaches a single computer science concept by activating the learner's curiosity and involving them in the learning process.

2.2 Why Puzzle-based Learning?

There are 10 good reasons why puzzle-based learning allows you to learn faster, smarter, and better. These reasons are based on published research in psychological science.

1. **Overcome the Knowledge Gap.**

If you solve a code puzzle, you want to know whether your answer is correct. You crave the correct solution and are thus preparing your mind to fully absorb it. Stretch your limits, overcome the knowledge gap, and become better—one puzzle at a time.

2. **Embrace the Eureka Moment.** To reinforce fast learning, evolution created a brilliant biological reaction. Your brain releases endorphins, the moment you close a knowledge gap. When solving a puzzle, you feel instant gratification. Learning becomes addictive—but this addiction makes you smarter.
3. **Divide and Conquer.** Code puzzles break up the huge task of learning to code into a series of smaller learning

units. The student experiences laser-focused learning. Examples of such small learning units are *recursion*, the *for loop*, and *keyword arguments*. Each puzzle is a step toward your bigger goal of mastering computer science. Keep solving puzzles and you keep improving your skills.

4. **Improve From Immediate Feedback.** To learn anything, you need feedback such that you can adapt your actions. Unconsciously, you will minimize negative and maximize positive feedback. This book offers you an environment with immediate feedback to make learning to code easy and fast.
5. **Measure Your Skills.** You can't improve what you can't measure. The main idea of this book, and the associated

learning app at <https://finxter.com>, is to transfer the method of measuring skills from chess to programming. By knowing your skills, you always know exactly where you stand on your path to mastery.

6. **Tailor Learning to You.** Individualized learning tailors the content, pace, style, and technology of teaching to the student's skills and interests. Puzzle-based learning is a perfect example of automated, individualized learning. The ideal puzzle stretches the student's abilities and is neither boring nor overwhelming.
7. **Small is Beautiful.** Puzzle-based learning breaks the bold goal, i.e., *reach a level of mastery in Python*, into tiny

actionable steps. While solving the small puzzles, you progress toward your larger goal and eventually reach the mastery level. A clear path to success.

8. **Active Beats Passive Learning.**

Puzzle-based learning is a variant of the active learning technique known as *practice testing*. Practice testing is scientifically proven to teach you more in less time.

9. **Make Source Code a First-class Citizen.**

Puzzle-based learning is code-centric. You will find yourself staring at the code for a long time until the insight strikes. Placing code to the center of the learning process creates an environment in which you will develop the powerful intuition of the expert.

10. **What You See Is All There Is.** A good code puzzle beyond the most basic level is self-contained. You can solve it purely by staring at it until your mind follows your eyes—your mind develops a solution based on rational thinking. No need to use any other material than what you already have: your brain.

2.3 How to Measure Your Python Skills with This Book?

The idea of our Python learning system is to transfer the Elo rating from chess to programming.

Let us give a small example of how the Elo rating works in chess. Alice is a strong player

with an Elo rating of 2000 while Bob is an intermediate player with Elo 1500. Say Alice and Bob play a chess game against each other. As Alice is the stronger player, she is expected to win. The Elo rating system rewards players for good and punishes for bad results: the better the result, the higher the reward. For Bob, a win, or even a draw, would be a very good outcome of the game. For Alice, the only satisfying result is a win. Winning against a weaker player is less rewarding than winning against a stronger player. Thus, the Elo rating system rewards Alice with only +3 Elo points for a win. A loss costs her -37 Elo points, and even a draw costs her -17 points. Playing against a weaker player is risky for her because she has much to lose but little to win.

The idea of Finxter is to view your learning as a series of games between two players: you

and the Python puzzle. Both players have an Elo rating. Your rating measures your current skills and the puzzle's rating reflects the difficulty. On our website finxter.com, a puzzle plays against hundreds of Finxter users. Over time, the puzzle's Elo rating converges to its true difficulty level.

Table 2.1 shows the ranks for each Elo rating level. The table is an opportunity for you to estimate your Python skill level. In the following, I describe how you can use this book to test your Python skills.

This book provides a series of 28 code puzzles plus explanations to test and train your Python slicing skills.

I recommend solving at least one or two code puzzles at a casual moment every day, e.g., while you drink your morning coffee. Then you can spend the rest of your learning time on

Elo rating	Rank
2500	World Class
2400-2500	Grandmaster
2300-2400	International Master
2200-2300	Master
2100-2200	National Master
2000-2100	Master Candidate
1900-2000	Authority
1800-1900	Professional
1700-1800	Expert
1600-1700	Experienced Intermediate
1500-1600	Intermediate
1400-1500	Experienced Learner
1300-1400	Learner
1200-1300	Scholar
1100-1200	Autodidact
1000-1100	Beginner
0-1000	Basic Knowledge

Table 2.1: Elo ratings and skill levels.

real projects that matter to you. The puzzles guarantee that your skills improve over time, while the real project will bring you results.

If you want to test your Python skills, use the following simple method.

1. Select a daily trigger after which you solve code puzzles for 10 minutes. For example, decide on your *Coffee Break Python*, or even solve code puzzles as you brush your teeth or sit on the train to work, university, or school.
2. Scan over the puzzle in a first quick pass and ask yourself: what is the unique idea of this puzzle?
3. Dive deeply into the code. Try to understand the purpose of each symbol, even if it seems trivial at first. Avoid being

shallow and lazy. Instead, solve each puzzle thoroughly and take your time. It's counterintuitive: To learn faster in less time, you must stay calm and take your time and allow yourself to dig deep. There are no shortcuts.

4. Make sure you carry a pen with you and write your solution into the book. This ensures that you stay objective—we all have the tendency to fake ourselves. Active learning is a central idea of this book.
5. Look up the solution and read the explanation with care. Do you understand every aspect of the code? Write open questions down and look them up later, or send them to me (info@finxter.com). I will do everything I can to come up

with a good explanation.

6. Only if your solution was 100% correct—including whitespaces, data types, and formatting of the output—award yourself with Elo points for this puzzle. Otherwise, you should count it as an incorrect solution and swallow the negative Elo points. The reason for this strict rule is that this is the best way to train yourself to solve the puzzles thoroughly.
7. Track your individual Elo rating as you read the book and solve the code puzzles. Simply write your current Elo rating into the book. Start with an initial rating of 1000 if you are a beginner, 1500 if you are an intermediate, and 2000 if you are an advanced Python programmer. Of course, if you already have

an online rating on <https://finxter.com>, starting with this rating is the most precise option.

8. If your solution is correct, add the Elo points according to the table given with the puzzle. Otherwise, subtract the given Elo points from your current Elo number.

As you follow this simple training plan, your skill to see through source code will improve. Over the long haul, this will have a huge impact on your career, income, and work satisfaction. You do not have to invest much time because the training plan requires only 10–20 minutes per day. But you must be persistent in your training efforts. If you get off track, get right back on track the next day. When you run out of code puzzles, feel free to

check out <https://finxter.com> with more than 300 hand-crafted code puzzles. I regularly publish new code puzzles on the website as well.

— 3 —

A Quick Overview of the Python Language

Before diving into the puzzles, work through the following five cheat sheets. The cheat sheets are highly compressed. I always refer to them as the 80/20 principle applied to learning to code—learn 80% of the Python language features in 20% of the learning time

(it would take you to acquire these features browsing through the web). So they are definitely worth your time investment. This introductory chapter is taken from my book¹.

Learn them thoroughly. Try to understand every single line of code. And catapult your skills to the next level. Most Python coders neglect to invest enough time in a thorough and comprehensive understanding of the basics such as language features, data types, and language tricks. Be different and absorb the examples in each of the cheat sheets. Open up your path to become a master coder and join the top ten percent of coders.

You can download all five cheat sheets as concise PDFs and post them on your wall until

¹*Coffee Break Python: 50 Workouts to Kickstart Your Rapid Code Understanding in Python*. If you don't feel comfortable reading and understanding Python code fast, get the book at <http://bit.ly/coffee-break-python>.

you know them by heart (<https://bit.ly/free-python-email-course>).

3.1 Keywords

Each programming language reserves a special meaning to a fixed set of words. These words are called *keywords*. With keywords, the programmer can issue commands to the compiler or interpreter. They let you tell the computer what to do. Without keywords, the computer could not make sense from the seemingly random text in your code file. Note that as keywords are reserved words, you cannot use them as variable names.

The most important Python keywords are the following:

False	True	and	or
not	break	continue	class
def	if	elif	else
for	while	in	is
None	lambda	return	

The next cheat sheet explains these keywords in detail. In each row, you can find the keyword itself, a short description, and an example of its usage.

Keyword	Description	Code example
False , True	Data values from the data type Boolean	<code>False == (1 > 2)</code> <code>True == (2 > 1)</code>
and , or , not	Logical operators: (x and y) → both x and y must be True (x or y) → either x or y must be True (not x) → x must be False	<code>x, y = True, False</code> <code>(x or y) == True</code> <code># True</code> <code>(x and y) == False</code> <code># True</code> <code>(not y) == True</code> <code># True</code>
break	Ends loop prematurely	<code>while(True):</code> <code>break # no infinite loop</code> <code>print("hello world")</code>
continue	Finishes current loop iteration	<code>while(True):</code> <code>continue</code> <code>print("43") # dead code</code>
class def	Defines a new class → a real-world concept (object oriented programming) Defines a new function or class method. For latter, first parameter self points to the class object. When calling a class method, the first parameter is implicit.	<code>class Beer:</code> <code>def __init__(self):</code> <code>self.content = 1.0</code> <code>def drink(self):</code> <code>self.content = 0.0</code> <code># constructor creates class</code> <code>becks = Beer()</code> <code># empty beer bottle</code> <code>becks.drink()</code>
if , elif ,	Conditional program execution: program starts	<code>x = int(input("your val: "))</code> <code>if x > 3: print("Big")</code>

<code>else</code>	with “if” branch, tries “elif” branches, and finishes with “else” branch (until one evaluates to True).	<code>elif x == 3: print("Medium")</code> <code>else: print("Small")</code>
<code>for,</code> <code>while</code>	<code># For loop declaration</code> <code>for i in [0,1,2]:</code> <code> print(i)</code>	<code># While loop - same semantics</code> <code>j = 0</code> <code>while j < 3:</code> <code> print(j)</code> <code> j = j + 1</code>
<code>in</code>	Checks whether element is in sequence	<code>42 in [2, 39, 42] # True</code>
<code>is</code>	Checks whether both elements point to the same object	<code>y = x = 3</code> <code>x is y # True</code> <code>[3] is [3] # False</code>
<code>None</code>	Empty value constant	<code>def f():</code> <code> x = 2</code> <code>f() is None # True</code>
<code>lambda</code>	Function with no name (anonymous)	<code>(lambda x: x + 3)(3) #</code> <code>returns 6</code>
<code>return</code>	Terminates function execution and passes the execution flow to the caller. An optional value after the return keyword specifies the result.	<code>def incrementor(x):</code> <code> return x + 1</code> <code>incrementor(4) # returns 5</code>

3.2 Basic Data Types

Many programmers know basic data types as *primitive data types*. They provide the primitives on which higher-level concepts are built. A house is built from bricks. Likewise, a complex data type is built from basic data types. I introduce basic data types in the next cheat sheet and complex data types in Section 3.3.

Specifically, the next cheat sheet explains the three most important (classes of) basic data types in Python. First, the *Boolean* data type encodes truth values. For example, the expression $42 > 3$ evaluates to **True** and $1 \in \{2, 4, 6\}$ evaluates to **False**. Second, the numerical types *integer*, *float*, and *complex numbers* encode integer values, floating point values, and complex values, respectively. For example, 41 is an integer value, 41.99 is

a float value, and $41.999 + 0.1 * i$ is a complex value (the first part of the equation being the real number and the second the imaginary number). Third, the *string* datatype encodes textual data. An example of a string value is the Shakespeare quote 'Give every man thy ear, but few thy voice'.

Data Type + Description	Example
<p>Boolean The Boolean data type is a truth value, either True or False.</p> <p>These are important Boolean operators ordered by priority (from highest to lowest):</p> <p>not <i>x</i> → “if <i>x</i> is False, then <i>x</i>, else <i>y</i>”</p> <p>x and y → “if <i>x</i> is False, then <i>x</i>, else <i>y</i>”</p> <p>x or y → “if <i>x</i> is False, then <i>y</i>, else <i>x</i>”</p>	<pre>x, y = True, False print(x and not y) # True print(not x and y or x) # True ## All of those evaluate to False if (None or 0 or 0.0 or '' or [] or {} or set()): print("Dead code") ## All of those evaluate to True if (1 < 2 and 3 > 2 and 2 >=2 and 1 == 1 and 1 != 0): print("True")</pre>
<p>Integer An integer is a positive or negative number without floating point (e.g. 3).</p> <p>Float A float is a positive or negative number with floating point precision (e.g. 3.14159265359).</p> <p>The ‘//’ operator performs integer division. The result is an integer value that is rounded towards the smaller integer number (e.g. 3 // 2 == 1).</p>	<pre>## Arithmetic Operations x, y = 3, 2 print(x + y) # = 5 print(x - y) # = 1 print(x * y) # = 6 print(x / y) # = 1.5 print(x // y) # = 1 print(x % y) # = 1s print(-x) # = -3 print(abs(-x)) # = 3 print(int(3.9)) # = 3 print(float(3)) # = 3.0 print(x ** y) # = 9</pre>

String

Python Strings are sequences of characters. They are immutable which means that you can not alter the characters without creating a new string.

The four main ways to create strings are the following.

1. Single quotes

```
'Yes'
```

2. Double quotes

```
"Yes"
```

3. Triple quotes (multi-line)

```
"""Yes
```

```
We Can"""
```

4. String method

```
str(5) == '5' # True
```

5. Concatenation

```
"Ma" + "hatma" #
```

```
'Mahatma'
```

These are whitespace characters in strings.

- Newline \n
- Space \s
- Tab \t

Indexing & Slicing

```
s = "The youngest pope was 11 years old"
```

```
print(s[0]) # 'T'
```

```
print(s[1:3]) # 'he'
```

```
print(s[-3:-1]) # 'ol'
```

```
print(s[-3:]) # 'old'
```

```
x = s.split() # string array
```

```
print(x[-3] + " " + x[-1] + " " +
```

```
x[2] + "s") # '11 old popes'
```

Key String Methods

```
y = " This is lazy\t\n"
```

```
print(y.strip()) # 'This is lazy'
```

```
print("DrDre".lower()) # 'drdre'
```

```
print("stop".upper()) # 'STOP'
```

```
s = "smartphone"
```

```
print(s.startswith("smart")) # True
```

```
print(s.endswith("phone")) # True
```

```
print("another".find("other")) # 2
```

```
print("cheat".replace("ch", "m"))
```

```
# 'meat'
```

```
print(', '.join(["F", "B", "I"]))
```

```
# 'F,B,I'
```

```
print(len("Rumpelstiltskin")) # 15
```

```
print("ear" in "earth") # True
```

3.3 Complex Data Types

In the previous section, you learned about basic data types. These are the building blocks for *complex data types*. Think of complex data types as containers—each holding a multitude of (potentially different) data types.

Specifically, the complex data types in this cheat sheet are *lists*, *sets*, and *dictionaries*. A list is an ordered sequence of data values (that can be either basic or complex data types). An example for such an ordered sequence is the list of all US presidents:

```
['Washington',  
'Adams',  
'Jefferson', ...,  
'Obama',  
'Trump'].
```

In contrast, a set is an *unordered* sequence of data values:

```
{ 'Trump',  
  'Washington',  
  'Jefferson', ...,  
  'Obama' }.
```

Expressing the US presidents as a set loses all ordering information—it's not a sequence anymore. But sets do have an advantage over lists. Retrieving information about particular data values in the set is much faster. For instance, checking whether the string **'Obama'** is in the set of US presidents is blazingly fast even for large sets. I provide the most important methods and ideas in the following cheat sheet.

Complex Data Type + Description	Example
<p>List</p> <p>A container data type that stores a sequence of elements. Unlike strings, lists are mutable: modification possible.</p>	<pre>l = [1, 2, 2] print(len(l)) # 3</pre>
<p>Adding elements</p> <p>to a list with append, insert, or list concatenation. The append operation is fastest.</p>	<pre>[1, 2, 2].append(4) # [1, 2, 2, 4] [1, 2, 4].insert(2,2) # [1, 2, 2, 4] [1, 2, 2] + [4] # [1, 2, 2, 4]</pre>
<p>Removing elements</p> <p>is slower (find it first).</p>	<pre>[1, 2, 2, 4].remove(1) # [2, 2, 4]</pre>
<p>Reversing</p> <p>the order of elements.</p>	<pre>[1, 2, 3].reverse() # [3, 2, 1]</pre>
<p>Sorting a list</p> <p>Slow for large lists: $O(n \log n)$, n list elements.</p>	<pre>[2, 4, 2].sort() # [2, 2, 4]</pre>
<p>Indexing</p> <p>Finds index of the first occurrence of an element in the list. Is slow when traversing the whole list.</p>	<pre>[2, 2, 4].index(2) # index of element 4 is "0" [2, 2, 4].index(2,1) # index of el. 2 after pos 1 is "1"</pre>
<p>Stack</p> <p>Python lists can be used intuitively as stack via the two list operations append() and pop().</p>	<pre>stack = [3] stack.append(42) # [3, 42] stack.pop() # 42 (stack: [3]) stack.pop() # 3 (stack: [])</pre>
<p>Set</p>	<pre>basket = {'apple', 'eggs', 'banana', 'orange'}</pre>

<p>Unordered collection of unique elements (<i>at-most-once</i>).</p>	<pre>same = set(['apple', 'eggs', 'banana', 'orange']) print(basket == same) # True</pre>
<p>Dictionary A useful data structure for storing (key, value) pairs.</p>	<pre>calories = {'apple' : 52, 'banana' : 89, 'choco' : 546}</pre>
<p>Reading and writing Read and write elements by specifying the key within the brackets. Use the keys() and values() functions to access all keys and values of the dictionary.</p>	<pre>c = calories print(c['apple'] < c['choco']) # True c['cappu'] = 74 print(c['banana'] < c['cappu']) # False print('apple' in c.keys()) # True print(52 in c.values()) # True</pre>
<p>Dictionary Looping You can access the (key, value) pairs of a dictionary with the items() method.</p>	<pre>for k, v in calories.items(): print(k) if v > 500 else None # 'chocolate'</pre>
<p>Membership operator Check with the keyword in whether the set, list, or dictionary contains an element. Set containment is faster than list containment.</p>	<pre>basket = {'apple', 'eggs', 'banana', 'orange'} print('eggs' in basket) # True print('mushroom' in basket) # False</pre>
<p>List and Set Comprehension List comprehension is the concise Python way to create lists. Use brackets plus an expression, followed by a for clause. Close with</p>	<pre>## List comprehension [('Hi ' + x) for x in ['Alice', 'Bob', 'Pete']] # ['Hi Alice', 'Hi Bob', 'Hi Pete'] [x * y for x in range(3) for y in range(3) if x>y] # [0, 0, 2]</pre>

zero or more for or if clauses.

Set comprehension is similar to list comprehension.

```
## Set comprehension
```

```
squares = { x**2 for x in [0,2,4] if x  
< 4 } # {0, 4}
```

3.4 Classes

Object-oriented programming is an influential, powerful, and expressive programming abstraction. The programmer thinks in terms of classes and objects. A class is a blueprint for an object. An object contains specific data and provides the functionality specified in the class.

Suppose you are programming a game to let you build, simulate, and grow cities. In object-oriented programming, you represent all things (buildings, persons, or cars) as objects. For example, each building object stores data such as name, size, and price tag. Additionally, each building provides a defined functionality such as `get_monthly_earnings()`. This simplifies reading and understanding your code for other programmers. Even more im-

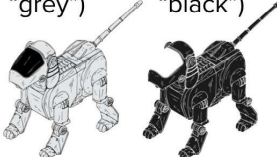
portant, you can now easily divide responsibilities. You code the buildings and your colleague codes the moving cars.

In short, object-oriented programming helps you to write readable code. By learning object orientation, your skill of collaborating with others on complex problems improves. The next cheat sheet introduces the most basic concepts.

Description

Classes

A class encapsulates data and functionality: data as attributes, and functionality as methods. It is a blueprint for creating concrete instances in memory.

Class: Dog Attributes name, state, color Methods command(x), bark(freq)	Instances (“Alice”, “sleeping”, “grey”) (“Bello”, “wag tail”, “black”) 
--	--

Instance

You are an instance of the class human. An instance is a concrete implementation of a class: all attributes of an instance have a fixed value. Your hair is blond, brown, or black---but never unspecified.

Each instance has its own attributes independent of other instances. Yet, class variables are different. These are data values associated with the class, not the instances. Hence, all instance share the same class variable `species` in the example.

Self

The first argument when defining any method is always the `self` argument. This argument specifies the instance

Example

```
class Dog:
    """ Blueprint of a dog """

    # class variable
    # for all instances
    species = ["canis lupus"]

    def __init__(self, n, c):
        self.name = n
        self.state = "sleeping"
        self.color = c

    def command(self, x):
        if x == self.name:
            self.bark(2)
        elif x == "sit":
            self.state = "sit"
        else:
            self.state = "wag tail"

    def bark(self, freq):
        for i in range(freq):
            print(self.name + ": Woof!")

bello = Dog("bello", "black")
alice = Dog("alice", "white")

print(bello.color) # black
print(alice.color) # white
```

on which you call the method.

`self` gives the Python interpreter the information about the concrete instance. To *define* a method, you use `self` to modify the instance attributes. But to *call* an instance method, you do not need to specify `self`.

```
class Employee():
    pass
employee = Employee()
employee.salary = 122000
employee.firstname = "alice"
employee.lastname =
"wonderland"

print(employee.firstname +
" " + employee.lastname +
" $" + str(employee.salary))
# alice wonderland $122000
```

```
bello.bark(1) # bello: Woof!

alice.command("sit")
print("alice: " +
alice.state)
# alice: sit

bello.command("no")
print("bello: " +
bello.state)
# bello: wag tail

alice.command("alice")
# alice: Woof!
# alice: Woof!

bello.species += ["wulf"]
print(len(bello.species)
== len(alice.species))
# True (!)
```

3.5 Functions and Tricks

Python is full of extra tricks and special functionality. Learning these tricks makes you more efficient and productive. But more importantly, these tricks make programming easy and fun. In the next cheat sheet, I give you the most important ones.

ADVANCED FUNCTIONS

`map(func, iter)`

Executes the function on all elements of the iterable. Example:

```
list(map(lambda x: x[0], ['red', 'green', 'blue']))  
# Result: ['r', 'g', 'b']
```

`map(func, i1, ..., ik)`

Executes the function on all k elements of the k iterables. Example:

```
list(map(lambda x, y: str(x) + ' ' + y + 's', [0, 2, 2],  
['apple', 'orange', 'banana']))  
# Result: ['0 apples', '2 oranges', '2 bananas']
```

`string.join(iter)`

Concatenates iterable elements separated by string. Example:

```
' marries '.join(list(['Alice', 'Bob']))  
# Result: 'Alice marries Bob'
```

`filter(func, iterable)`

Filters out elements in iterable for which function returns False (or 0). Example:

```
list(filter(lambda x: True if x>17 else False, [1, 15, 17,  
18])) # Result: [18]
```

`string.strip()`

Removes leading and trailing whitespaces of string. Example:

```
print("\n \t 42 \t ").strip() # Result: 42
```

`sorted(iter)`

Sorts iterable in ascending order. Example:

```
sorted([8, 3, 2, 42, 5]) # Result: [2, 3, 5, 8, 42]
```

`sorted(iter, key=key)`

Sorts according to the key function in ascending order. Example:

```
sorted([8, 3, 2, 42, 5], key=lambda x: 0 if x==42 else x)  
# [42, 2, 3, 5, 8]
```

`help(func)`

Returns documentation of func. Example:

```
help(str.upper()) # Result: '... to uppercase.'
```

```
zip(i1, i2, ...)
```

Groups the i-th elements of iterators i1, i2, ... together. Example:

```
list(zip(['Alice', 'Anna'], ['Bob', 'Jon', 'Frank']))  
# Result: [('Alice', 'Bob'), ('Anna', 'Jon')]
```

```
Unzip
```

Equal to: 1) unpack the zipped list, 2) zip the result. Example:

```
list(zip(*(['Alice', 'Bob'], ('Anna', 'Jon')))  
# Result: [('Alice', 'Anna'), ('Bob', 'Jon')]
```

```
enumerate(iter)
```

Assigns a counter value to each element of the iterable. Example:

```
list(enumerate(['Alice', 'Bob', 'Jon']))  
# Result: [(0, 'Alice'), (1, 'Bob'), (2, 'Jon')]
```

TRICKS

```
python -m http.server <P>
```

Want to share files between your PC and your phone? Run this command in your PC's shell. <P> is any port number between 0–65535. Type < IP address of PC>:<P> in the phone's browser. Now, you can browse the files in the PC's directory.

```
Read comic
```

```
import antigavity
```

Opens the comic series xkcd in your web browser

```
Zen of Python
```

```
import this
```

```
'...Beautiful is better than ugly. Explicit is ...'
```

```
Swapping variables
```

This is a breeze in Python. No offense, Java! Example:

```
a, b = 'Jane', 'Alice'
```

```
a, b = b, a
```

```
# Result: a = 'Alice', b = 'Jane'
```

Unpacking arguments

Use a sequence as function arguments via asterisk operator *. Use a dictionary (key, value) via double asterisk operator **. Example:

```
def f(x, y, z):  
    return x + y * z  
f(*[1, 3, 4]) # 13  
f(**{'z' : 4, 'x' : 1, 'y' : 3}) # 13
```

Extended Unpacking

Use unpacking for multiple assignment feature in Python. Example:

```
a, *b = [1, 2, 3, 4, 5]  
# Result: a = 1, b = [2, 3, 4, 5]
```

Merge two dictionaries

Use unpacking to merge two dictionaries into a single one. Example:

```
x={'Alice' : 18}  
y={'Bob' : 27, 'Ann' : 22}  
z = {**x,**y}  
# Result: z = {'Alice': 18, 'Bob': 27, 'Ann': 22}
```


— 4 —

Introduction to Slicing in Python

Slicing is a Python-specific concept for carving out a range of values from sequence types such as lists or strings.

It is one of the most popular Python features. To master Python, you must master slicing first. Any non-trivial Python code base

relies on slicing. In other words, the time you invest now in mastering slicing will be repaid a hundredfold during your career.

4.1 Intermezzo: Indexing Basics

To bring everybody on the same page, let me quickly introduce indexing in Python. Each Python sequence object (e.g. strings, lists, tuples) consists of a number of elements. You access each element in the sequence via its *index*. Each position in the sequence has a unique index. The first element of the sequence has index 0. The second element of the sequence has index 1. And the i -th element of the sequence has index $i-1$. The indices of any sequence `s` start at position 0 and end in position `len(s)-1`.

Let's dive into an example. Suppose, you have a string **universe**. What are the indices in this example? The indices are simply the positions of the characters of this string.

Index	0	1	2	3	4	5	6	7
Character	u	n	i	v	e	r	s	e

The first character has index 0, the second character has index 1, and the *i*-th character has index *i*-1.

4.2 Slicing

The idea of slicing is simple. You carve out a subsequence (called a *slice*) from a sequence by defining the start and end indices. While indexing retrieves only a single element, slicing retrieves a whole subsequence within an index range.

Use the bracket notation for slicing with the start and end position identifiers. For example, `word[i:j]` returns the substring starting from index `i` (included) and ending in index `j` (excluded). But be cautious! Forgetting that the end index is excluded is a common source of bugs.

Based on this information, can you solve the first code puzzle?

```
#####  
## Puzzle 1  
## Level: Easy  
## Elo +10 / -10  
#####
```

```
x = 'universe'  
print(x[2:4])
```

Puzzle 1: What is the output of this code puzzle?

The following table gives the resulting slice in bold. It starts with the start index 2 (inclusive) and ends with the end index 4 (exclusive). This means that the character with index 4, i.e., character **e**, is not part of the resulting slice. The solution of this puzzle is the substring **iv**.

If you solved this puzzle correctly, you get ten experience points (+10 Elo), otherwise, you lose ten experience points (-10 Elo). In the latter case, your current experience level would be -10 Elo. Track your experience level as you proceed with the book.

Index	0	1	2	3	4	5	6	7
Character	u	n	i	v	e	r	s	e

4.3 The Step Size in Slicing

For the sake of completeness, let me explain the advanced slicing notation consisting of three arguments `[start:end:step]`. The only difference to the previous notation is that it allows you to specify the step size as well. For example, the command `'python'[:5:2]` returns every second character up to the fourth character, i.e., the string `'pto'`.

With this information, you should now be able to solve the following puzzle.

```
#####  
## Puzzle 2  
## Level: Easy  
## Elo +10 / -10  
#####
```

```
x = 'universe'  
print(x[2::2])
```

Puzzle 2: What is the output of this code puzzle?

The following table shows the selected slice indices in bold. The slice begins with the start index 2. Then, it adds every second index in the slice. Thus, the solution of this puzzle is the substring **ies**.

Index	0	1	2	3	4	5	6	7
Character	u	n	i	v	e	r	s	e

4.4 Overshooting Indices in Slicing

Slicing is robust even if the end index shoots over the maximal sequence index. Just remember that nothing unexpected happens if slicing overshoots sequence indices. Here is a code puzzle that exemplifies this behavior. Can you solve it?

```
#####  
## Puzzle 3  
## Level: Easy  
## Elo +10 / -10  
#####
```

```
word = "galaxy"  
print(word[4:50])
```

Puzzle 3: What is the output of this code puzzle?

Again, you can find the selected slice indices in the following table. Note that the end index is 50, which is much larger than the maximal index 5 of the string 'galaxy'. Thus, all *impossible* indices are simply ignored by the interpreter. The correct solution is the substring `xy`.

Index	0	1	2	3	4	5
Character	g	a	l	a	x	y

4.5 Summary Basic Python Slicing

Short recap – the slice notation `s[start:end:step]` carves out a substring from `s`. The substring consists of all characters between the two characters at the start index (inclusive) and the end index (exclusive). An optional step size

indicates how many characters are left out from the original sequence. Here is an example:

```
#####  
## Puzzle 4  
## Level: Easy  
## XP +10 /- 10  
#####
```

```
s = 'sunshine'  
print(s[1:5:2] + s[1:5:1])
```

Puzzle 4: What is the output of this code puzzle?

In the following table, you can find both slices and how they emerged from the original string '**sunshine**'. The first slice consists of every other character between the start index 1 and end index 5 (excluded). The second slice consists of all characters between those start and end indices. The resulting string is a simple *concatenation* of these two substrings as indicated by the context-sensitive **+** operator. Therefore, the solution is the substring **usunsh**.

0	1	2	3	4	5	6	7		0	1	2	3	4	5	6	7
s	u	n	s	h	i	n	e		s	u	n	s	h	i	n	e

4.6 Frequently Asked Questions

Let's dive deeper into slicing to make sure that you fully understand it.

I have searched Quora to find all the little problems new Python coders are facing with slicing. I will answer six common questions next.

What does it mean to skip slicing indices (e.g. `s[::2]`)?

The Python interpreter assumes certain default values for `s[start:stop:step]`. You can tell the interpreter to use the default values by simply skipping the value(s) before (and after) the colon `:`. At the first glance, such code may look a bit awkward. Don't worry,

you will get used to this more dense notation.

So what are the default values assumed by the interpreter? For the most simple cases, the default values are **start=0**, **stop=len(s)**, and **step=1**. You would write those same default values in the slice notation as follows: `s[::]==s[0:len(s):1]`).

The next puzzle contains the three basic cases of skipping indices while assuming (default) step size 1.

```
#####  
## Puzzle 5  
## Level: Easy  
## Elo +10 / -10  
#####
```

```
word = "galaxy"  
print(word[:-2] + word[4:])  
print(word[:])
```

Puzzle 5: What is the output of

this code puzzle?

In this example, we obtain three slices from the original string `galaxy`. The first slice carves out a substring starting from the default index 0 and ending in the second last character with index -2 (exclusive). (A negative index means that you start counting from the right, so that the index -1 refers to the last element.) The second slice carves out a substring starting from index 4 and ending in the default end index `len('galaxy') == 6`. The third string assumes both default indices 0 and 6 — so it basically retrieves a copy of the original string. As the first and second slices are concatenated, the solution of this puzzle is two times `galaxy` in the first and the second line of the output.

When to use the single colon notation (e.g. `s[:]`) and when double colon notation (e.g. `s[::2]`)?

A single colon (e.g. `s[1:2]`) allows for two arguments, the start and the end index. A double colon (e.g. `s[1:2:2]`) allows for three arguments, the start index, the end index, and the step size. Only if the step size is set to the default value 1, should you use the single colon notation for brevity. In all other cases, you have to use the double colon notation in order to specify the step size.

```
#####  
## Puzzle 6  
## Level: Easy  
## Elo +10 / -10  
#####
```

```
word = "O brother where art thou?"  
print(word[::])  
print(word[::1])  
print(word[:])
```

Puzzle 6: Two out of three slice versions are considered as bad style. Which of them?

The puzzle contains three different slices. Each produces the same result—the original string `0 brother where art though?` But the first two slices are considered bad style. Why? Because they contain redundant information. The first version has a redundant colon. The second version is even worse as it contains not only a redundant colon, but also a redundant default value. As a rule of thumb, you should always avoid redundant code.

What does a negative step size mean (e.g. `s[5:1:-1]`)?

This is an interesting feature in Python. A negative step size indicates that we are not slicing from left to right, but from right to left. Hence, the start index should be larger than or equal to the end index (otherwise, the

resulting sequence is empty). This is handy if you want to reverse the order of sequences.

```
#####  
## Puzzle 7  
## Level: Intermediate  
## Elo +10 / -10  
#####
```

```
word = "0 brother where art thou?"  
print(word[9:1:-1][::-1])
```

Puzzle 7: What's the output of this code snippet?

In this puzzle, we apply the slicing operator with negative step size -1 twice. First, we cut out a slice starting from index 9 (inclusive) and ending in index 1 (exclusive). Note that this creates a new string with reversed order of characters with respect to the original string. The result of the first slice operations is, therefore, the string '**rehtorb**'. Second, we reverse the order of characters of this resulting substring again while assuming the default indices. Hence, the final result of the puzzle is '**brother**'. Congratulations, if you managed to solve this puzzle correctly!

What are the default indices when using a negative step size (e.g. `s[::-1]`)?

If you followed the above explanations very carefully, you might have realized that the default handling of indices has to be different when using a negative step size.

To recap, these are the default values for the slice `[start:end:step]`:

- `start = 0`,
- `end = len(s)`, and
- `step = 1`.

However, if we assume negative step size, these default values do not make a lot of sense because the default behavior should be to slice from the **last** to the *first* index.

Thus, the default indices in this case are not `start = 0` and `end = len(s)` but the other way round:

- `start = len(s) - 1` and
- `end = -len(word) - 1`.

Why does the end index have a value of `-len(word)-1`? The reason is that the start index is still included and the end index still excluded from the slice. Only if the end index is smaller than the value `-len(word)` do we actually slice all the way to the first sequence element (inclusive).

```
#####  
## Puzzle 8  
## Level: Hard  
## Elo +40 / -10  
#####
```

```
word = "you shall not pass!"  
print(word[len(word):0:-1] == word[::-1])  
print(word[len(word)-1:-len(word)-1:-1] ==  
↪ word[::-1])
```

Puzzle 8: What's the output of this code snippet?

This puzzle shows you the default indexing for negative slices.

Seemingly, the first slice `word[len(word):0:]` produces the same slice as when assuming the default indices. This is not the case. The reason is simple: the end index 0 is *exclusive*: it's not included in the final result. Hence, the result of `word[len(word):0:-1]` is the string `'!ssap ton llahs uo'` missing the first character `'y'` from the original string.

In contrast to that does the second slice produce the same string as the default operation `word[::-1]`. It starts from the last index of the string `len(word)-1` (inclusive), and slices `len(word)+1` characters to the left (because of the equation `-len(word)-1 == -(len(word)+1)`).

If you solved this advanced puzzle, consider yourself a master slicer!

We have seen many examples for string slicing. How does list slicing work?

Slicing works the same for all sequence types (e.g., lists, strings, tuples). For lists, consider the following example:

```
#####  
## Puzzle 9  
## Level: Easy  
## Elo +10 / -10  
#####
```

```
l = [1, 2, 3, 4]  
print(l[2:])  
print(l[::-2])
```

Puzzle 9: What is the output of this code puzzle?

The first line of the puzzle generates a list starting from index 2 and slicing all the way to the end of the list. Hence, the result is the sublist `[3, 4]`.

The second line of the puzzle generates a list with reversed order of the sequence elements. However, as the step size is set to the value `-2`, only every second element is taken from the string. Hence, the slice operation generates the list `[4, 2]`.

Why is the last index excluded from the slice?

This question goes well beyond the topic of slicing with Python. In Python, as in many other programming languages when defining an interval `[start:end]`, the end index is not included in the subsequence.

The last index is excluded from the slice because of language consistency, e.g. the range function also does not include the end index.

It makes sense to exclude the main index because of clarity. Try not to think about the following question. Answer it immediately to see what your intuition tells you. How many sequence elements are in the sequence interval $[x, x+k]$? Exactly, k elements!

Now suppose, we included the end index here. In this case, the total length of the sequence interval would be $k + 1$ characters. This would be very counter-intuitive.

```
#####  
## Puzzle 10  
## Level: Easy  
## Elo +10 /- 10  
#####
```

```
customer_name = 'Hubert'
```



```
k = 3 # maximal size of database entry
x = 1 # offset
db_name = customer_name[x:x+k]
print(db_name)
```

Puzzle 10: What's the output of this puzzle?

This puzzle exemplifies the above explanations. Suppose a database key is allowed to have only three characters, i.e., $k + 1$. We create a new variable with name `db_name`. By specifying the offset `x` and the maximal size of the database entry `k`, we can easily obtain a correctly sized subsequence via the slice operation `customer_name[x:x+k]`. The result is the string literal `'ube'`.

Is the slice substring an alias of the original substring (i.e., does the resulting object refer to the same object in memory)?

In general, this is not the case. Strings are immutable objects in Python—they cannot be changed. Thus, if you apply the slice opera-

tion to a string, the interpreter will create a new string.

The same holds for lists: slicing creates a copy of the list. So if you modify a sliced copy, you do not have to worry about accidentally modifying the original sequence. With this information, you should be able to solve the following puzzle.

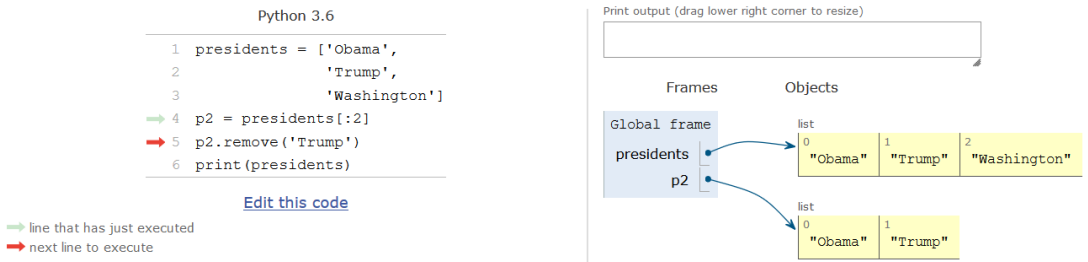
```
#####  
## Puzzle 11  
## Level: Intermediate  
## Elo +20 / -10  
#####
```

```
presidents = ['Obama',  
              'Trump',  
              'Washington']  
p2 = presidents[:2]  
p2.remove('Trump')  
print(presidents)
```

Puzzle 11: What's the output?

The following visualization shows which objects are created by the Python interpreter when executing the code in the puzzle (generated with the excellent tool <http://www.pythontutor.com/> from Philip Guo).

The first graphic shows the situation after executing line 4 in the code. The two names `presidents` and `p2` do refer to different list objects. The list `p2` is shorter because of the selection of the slice indices in line 4.



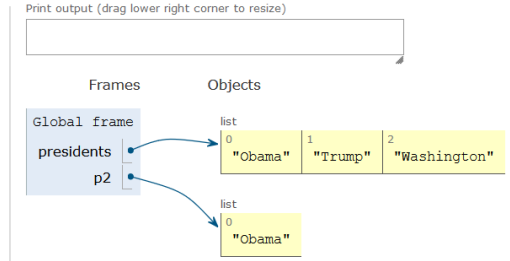
The second graphic shows the situation after executing line 5 of the code. Still, both names refer to different objects. Thus, removing the string `'Trump'` from the list object referred by the name `p2` has no effect on the

list object referred by the name **presidents**.

```
Python 3.6
1 presidents = ['Obama',
2               'Trump',
3               'Washington']
4 p2 = presidents[:2]
5 p2.remove('Trump')
6 print(presidents)
```

[Edit this code](#)

→ line that has just executed
→ next line to execute



How does slice assignment work?

We have seen many examples of the following type: carving out a substring of an original string by using the slice operation:

`s[start:stop:end]`.

However, there is also another interesting application of slicing in Python that arises quite frequently in practical code bases: *slice assignments*. Do not confuse slicing and slice assignments as they are two different operations!

Slicing in the form of `y = x[start:stop]` creates a new string from the original string `x` and assigns it to the name `y`.

In contrast to that, slice assignments have the form `x[start:stop] = y`. It replaces parts of the original string `x` by the string `y`. In other words, the string literal `y` conceptually overwrites the selected slice in `x`. List slicing works in a similar way.

Here is a puzzle testing your understanding of slice assignments.

```
#####  
## Puzzle 12  
## Level: Intermediate  
## Elo +20 / -10  
#####
```

```
l = [1, 2, 3, 4, 5]  
l[:3] = [42, 41]  
print(l)
```

Puzzle 12: What's the output of this puzzle?

The first line of the puzzle creates a new list with values between 1 and 5 (inclusive). Note that we could have also used the expression `list(range(6))` to achieve the same result.

In the second line, we use slice assignment to replace the first three list elements with the new list `[42, 41]`. In general, we select an arbitrary combination of elements via slicing on the *left-hand side* of the equation. Then, we create an arbitrary sequence *right-hand side* of the equation by which we replace the selected items. Note that in this basic case, the sequence on the left-hand side can even be smaller (or larger) than the selected right-hand side. Hence, the result of this puzzle is the list `[42, 41, 4, 5]`.

To dive a bit deeper into slice assignments for lists, let's have a look at another example.

In the last paragraph, I have indicated that the left-hand side and the right-hand side of the equation can have a different number of elements. This only makes sense if we use sub-sequence list elements that can be replaced. If we use extended slicing with a non-standard step size (e.g., `s[::2]`), the Python interpreter cannot decide anymore where to put the remaining elements (in case the right-hand side has more elements). Therefore, for extended slicing, Python assumes the same number of elements for the left-hand side and the right-hand side. If the number of elements is different, the Python interpreter will throw an exception such as *ValueError: attempt to assign sequence of size 2 to extended slice of size 3*.

Now you should be able to solve the following puzzle.

```
#####  
## Puzzle 13  
## Level: Intermediate  
## Elo +20 / -10  
#####
```

```
l = [1, 2, 3, 4, 5]  
l[::-2] = [42, 41]  
print(l)
```

Puzzle 13: What's the output of this puzzle?

Well done, if you saw that this leads to a `ValueError` as discussed above. The extended slice selects three elements but we try to replace it with only two elements.

Can you use slice assignments for strings?

No. In contrast to lists, strings are immutable data types. In other words, strings cannot be changed after they are created. But the semantics of slice assignments is to replace a slice of the original sequence. Due to the immutability, Python cannot replace any part of the original string. Therefore, it throws an error when you attempt to do slice assignments for strings.

— 5 —

Code Puzzles Slicing

In the previous chapters, we have introduced slicing in depth and revisited the Python basics.

Now take your pen, fill your cup of coffee, and let's dive into the 11 code puzzles in the book. The puzzles are very basic in the beginning but become harder as you proceed.

Again, take your time and try to understand each line before you move on to the next puzzle.

```
#####  
## Puzzle 14  
## Level: Intermediate  
## XP +20 / -10  
#####
```

```
# Shakespeare:  
s = "All that glitters is not gold"  
print(s[9:-9])  
print(s[::10])  
print(s[:-4:-1])
```

Puzzle 14: What's the output of this puzzle?

The result of the first print statement is 'glitters is'. The result of the second slice operation is the new string 'Al'. The result of the third slice operation is the new string 'dlo'.

```
#####  
## Puzzle 15  
## Level: Easy  
## Elo +10 / -10  
#####
```

```
x = list('universe')  
print(x)  
print(x[2:4])
```

Puzzle 15: What is the output of this code puzzle?

The solution of this puzzle is the list ['i', 'v'].

```
#####  
## Puzzle 16  
## Level: Easy  
## Elo +10 / -10  
#####
```

```
l = ['u', 'n', 'i', 'v', 'e', 'r', 's', 'e']  
print(l[2::2])
```

Puzzle 16: What is the output of this code puzzle?

The solution of this puzzle is the new list
['i', 'e', 's'].

```
#####  
## Puzzle 17  
## Level: Easy  
## Elo +10 / -10  
#####
```

```
word = list("galaxy")  
print(word[4:50])
```

Puzzle 17: What is the output of this code puzzle?

The correct solution is the new list `['x', 'y']`.

```
#####  
## Puzzle 18  
## Level: Easy  
## Elo +10 / -10  
#####
```

```
lst = list(range(10))  
print(lst[1:5:2] + lst[1:5:1])
```

Puzzle 18: What is the output of this code puzzle?

The solution is the list [1, 3, 1, 2, 3, 4].

```
#####  
## Puzzle 19  
## Level: Easy  
## Elo +10 / -10  
#####
```

```
lst = [2, 4, 6, 8, 10, 12]  
print(lst[:-2] + lst[4:])  
print(lst[:])
```

Puzzle 19: What is the output of this code puzzle?

The solution of this puzzle is two times [2, 4, 6, 8, 10, 12] in the first and the second line of the output.

```
#####  
## Puzzle 20  
## Level: Easy  
## Elo +10 / -10  
#####
```

```
lst = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]  
print(lst[:])  
print(lst[::1])  
print(lst[:])
```

Puzzle 20: Two out of three slicing statements in the puzzle are considered bad style. Which of them?

The result is [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] for all three versions. But the first two versions are bad programming style due to the code redundancy.

```
#####  
## Puzzle 21  
## Level: Intermediate  
## Elo +10 / -10  
#####
```

```
word = list(range(10))  
print(word[9:1:-1][::-1])
```

Puzzle 21: What's the output of this code snippet?

The final result of the puzzle is [2, 3, 4, 5, 6, 7, 8, 9].

```
#####  
## Puzzle 22  
## Level: Hard  
## Elo +40 / -10  
#####
```

```
word = [0, 3, 7, 10, 13]  
print(word[len(word):0:-1] == word[::-1])  
print(word[len(word)-1:-len(word)-1:-1] ==  
↪ word[::-1])
```

Puzzle 22: What's the output of this code snippet?

The result of this puzzle is **False** in the first line and **True** in the second line of the output.

```
#####  
## Puzzle 23  
## Level: Easy  
## XP +10 /- 10  
#####
```

```
lst = [1, 2, 3, 4, 5]  
k = 3  
x = 1  
print(lst[x:x+k])
```

Puzzle 23: What's the output of this puzzle?

The result is the new list [2, 3, 4].

```
#####  
## Puzzle 24  
## Level: Intermediate  
## Elo +20 / -10  
#####
```

```
l = [1, 2, 3, 4, 5]  
l[2:3] = [42, 41]  
print(l)
```

Puzzle 24: What's the output of this puzzle?

The result of this puzzle is the new list [1, 2, 42, 41, 4, 5].

— 6 —

Final Remarks

Congratulations, you made it through this slicing book. As a result, you have improved your skills in understanding slicing in Python—an integral part of your journey to becoming a Python master coder.

By now, you should have a fair estimate of your skill level in comparison to others. Be

sure to check out Table 2.1 again to get the respective rank for your Elo rating.

Consistent effort and persistence is the key to success. Do you feel that solving code puzzles has advanced your skills? Make it a daily habit to solve a Python puzzle and watch the related video on the Finxter web app. This habit alone will push your coding skills through the roof—and provide a comfortable living for you and your family in a highly profitable profession. Build this habit into your life—e.g., use your morning coffee break routine—and you will soon become one of the best programmers in your environment.

Where to go from here? I am publishing a fresh code puzzle every couple of days on our website <https://finxter.com>. All puzzles are available for free. My goal with

Finxter is to make learning to code easy, individualized, and accessible.

- For any feedback, questions, or problems where you struggle or need help, please send an email to `info@finxter.com`.
- To grow your Python skills on autopilot, register for the free Python email course at the following url: `http://bit.ly/free-python-course`.
- This is the second book in the *Coffee Break Python* series which is all about pushing you—in your daily coffee break—to the intermediate level in Python. Please find the first book at `http://bit.ly/coffee-break-python`.

Having read this book, you will feel confident using slicing in your everyday life. Please rate the book on Amazon to help others finding it.

Finally, I would like to express my deep gratitude that you have spent your time solving code puzzles and reading this book. Above everything else, I value your time. The ultimate goal of any good textbook should be to *save your time*. By working through this textbook, you have gained insights about your coding skill level and I hope that you have experienced a positive return on invested time and money. Now, please keep investing in yourself and stay active within the Finxter community.