

Coffee Break Python WORKBOOK



127 Python Puzzles To Push You From
Zero To Hero In Your Coffee Breaks

MAYER, RIEGER, RIAZ

Coffee Break Python Workbook

127 Python Puzzles to Push You from
Zero to Hero in Your Coffee Breaks

Christian Mayer, Lukas Rieger, and Zohaib Riaz

January 23, 2020

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

Contents

Contents	ii
1 Introduction	1
2 A Case for Puzzle-based Learning	4
2.1 Overcome the Knowledge Gap	5
2.2 Embrace the Eureka Moment	6
2.3 Divide and Conquer	7
2.4 Improve From Immediate Feedback	7
2.5 Measure Your Skills	8
2.6 Individualized Learning	10
2.7 Small is Beautiful	11
2.8 Active Beats Passive Learning	12
2.9 Make Code a First-class Citizen	14
2.10 What You See is All There is	16
3 Elo	17
3.1 How to Use This Book	17

3.2	How to Test and Train Your Skills?	19
3.3	What Can This Book Do For You?	22
4	A Quick Overview of the Python Language	27
4.1	Keywords	28
4.2	Basic Data Types	31
4.3	Complex Data Types	34
4.4	Classes	38
4.5	Functions and Tricks	41
5	Puzzles: Basic to Scholar	45
5.1	Printing values	45
5.2	Basics of variables	46
5.3	Getting started with strings	47
5.4	Types of variables I	48
5.5	Types of Variables II	49
5.6	Minimum	50
5.7	String Concatenation	51
5.8	Line Breaks I	52
5.9	Line Breaks II	53
5.10	List Length	54
5.11	Comparison Operators I	55
5.12	Comparison Operators II	56
5.13	Multiple Initializations	57
6	Puzzles: Scholar to Intermediate	59
6.1	Maximum	59
6.2	Memory addresses	60
6.3	Swapping Values	61
6.4	The Boolean Operator AND	62

6.5	The Boolean Operator OR	64
6.6	Boolean Operators	66
6.7	Arithmetic Expressions	67
6.8	Integer Division and Modulo	69
6.9	Building Strings	71
6.10	The <code>len()</code> Function	72
6.11	String Indices	73
6.12	The <code>upper()</code> Function	74
6.13	The <code>lower()</code> Function	75
6.14	Somebody Is Shouting	76
6.15	Counting Characters	77
6.16	String Lengths	78
6.17	Finding Characters in Strings	79
6.18	Not Finding Characters in Strings	80
6.19	Counting Letters	81
6.20	<code>Min()</code> and <code>Max()</code> of a String	82
6.21	Reversed Strings	83
6.22	String Equality	84
6.23	Slicing I	86
6.24	Slicing II	87
6.25	Slicing III	88
6.26	Slicing IV	89
6.27	Slicing V	90
6.28	Memory Addresses and Slicing	91
6.29	Accessing List Items I	92
6.30	Accessing List Items II	93
6.31	List as Stack	94
6.32	More String Operations	95
6.33	Checking for Substrings	96

6.34 Stripping String Boundaries	97
6.35 Strings: Stripping vs. Replacement	98
6.36 Gluing Strings Together	99
6.37 The Copy Operation	100
6.38 Growing List Contents I	101
6.39 Growing List Contents II	102
6.40 List Operations I	104
6.41 List Operations II	105
6.42 List Operations III	106
6.43 List Operations IV	107
6.44 List Operations V	107
6.45 List Operations VI	108
6.46 List Operations VII	109
6.47 List Operations VIII	110
6.48 List Operations IX	111
6.49 List Operations X	112
6.50 Lists and the Range Function I	113
6.51 Lists and the Range Function II	114
6.52 Lists and the Range Function III	115
6.53 Python's Multiple Assignment I	116
6.54 Slice Assignments	117
6.55 Strings and Lists II	118
6.56 String Comparisons	119
6.57 From Booleans to Strings	120
6.58 Boolean Trickery I	121
6.59 Boolean Trickery II	122
6.60 Boolean Trickery III	123
6.61 Looping over Ranges	124
6.62 Reversed Loops	126

6.63	Boolean Trickery IV	127
6.64	Lists and Memory Addresses	128
6.65	List Objects	129
6.66	Boolean Tricks	130
6.67	Complex Numbers	131
6.68	Tuples	132
6.69	Multiple Assignments	133
6.70	Boolean Integer Conversion	134
6.71	The <code>any()</code> Function	135
6.72	The <code>sum()</code> Function	137
6.73	Accessing Complex Numbers	139
6.74	Tuple Confusion	140
6.75	Understanding While ... Else (1/3)	141
6.76	Understanding While ... Else (2/3)	143
6.77	Understanding While ... Else (3/3)	145
6.78	Basic Arithmetic	147
6.79	Dictionary	148
6.80	Dictionary of Dictionaries	150
6.81	Reverse Dictionary Index	152
6.82	Default Arguments	154
7	Puzzles: Intermediate to Professional	156
7.1	Building Strings II	156
7.2	String: Slicing and Indexing	158
7.3	Built-in Python Operations	159
7.4	Strings and Lists I	160
7.5	Formatting Printouts	161
7.6	Floating Point Comparisons	163
7.7	Python's Multiple Assignment II	165
7.8	The Not-So-Obvious Case	167

7.9	Rounding Values	168
7.10	Initializing Integers	170
7.11	Basic Typing	171
7.12	Short Circuiting	172
7.13	While Arithmetic	174
7.14	The Lambda Function	176
7.15	Zip	177
7.16	Basic Filtering	178
7.17	List Comprehension	181
7.18	Encryption by Obfuscation	182
7.19	String Dictionary	184
7.20	Functions are Objects	186
7.21	Dictionary of Dictionaries	188
7.22	Sorting Dictionary Keys	190
7.23	Pythonic Loop Iteration	192
7.24	Filtering with List Comprehension	194
7.25	Aggregating with List Comprehension	196
7.26	Maximum of Tuples	198
7.27	The Key Argument	200
7.28	Puzzle 123	202
7.29	Set Operations (1/2)	204
7.30	Set Operations (2/2)	205
7.31	Recursive Algorithm	207
7.32	Fibonacci	210
8	Final Remarks	214
	Your skill level	214
	Where to go from here?	215
9	50 Bonus Workouts	222

9.1	Arithmetic	222
9.2	Whitespace	223
9.3	Modulo	224
9.4	Tuple	225
9.5	Dictionary	226
9.6	Asterisk	227
9.7	Slicing 1	228
9.8	Slicing 2	229
9.9	Nested Loop	230
9.10	List Arithmetic	231
9.11	Exception	232
9.12	Insert	233
9.13	Sorted Dictionary	234
9.14	Default	235
9.15	Keyword Argument	236
9.16	Global	237
9.17	Flow 1	238
9.18	Flow 2	239
9.19	Enumerate	240
9.20	Reverse	241
9.21	Hierarchical Functions	242
9.22	Sorting++	243
9.23	Indexing	244
9.24	Count	245
9.25	Power	246
9.26	Lambda	247
9.27	Recursion	248
9.28	Kwargs	249
9.29	Dictionary Magic	250

9.30	Sort Key	251
9.31	Print	252
9.32	Logic	253
9.33	Argument Confusion	254
9.34	Pass	255
9.35	List Magic	256
9.36	Zipzip	257
9.37	Comprehension	258
9.38	Slice Extend	259
9.39	Max	260
9.40	Zip	261
9.41	Unpack	262
9.42	Minimax	263
9.43	Sort	264
9.44	Tuple List	265
9.45	While	266
9.46	String Logic	267
9.47	Unorthodox Dict	268
9.48	Count	269
9.49	Cut	270
9.50	End	271

— 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 after collecting research from various fields such as psychology and neurological science.¹ The rule states that if you have average talent, you will reach mastery in any discipline after 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. He was anything but an overnight success.

If you are reading this book, you are an aspiring coder and seek ways to advance your coding skills. Nurturing your ambition to learn will pay dividends to you and

¹Malcolm Gladwell *Outliers: The Story of Success*

your family for as long as you live. It will make you a respectable member of society. Plus you provide value to different parts of society such as information technology, automation, and digitization. Ultimately, it will give you confidence. So, keeping your ambition to learn is the one thing you must place above all else.

The *Coffee Break Python Workbook* is the fourth book in the series.² In a way, it's an extension of its predecessor *Coffee Break Python*³—but with 127 brand-new code puzzles teaching new Python concepts, it stands on its own.

This book aims to be a stepping stone on your path to becoming a Python master. It helps you to learn faster by using the principles of good teaching. It contains 15-25 hours of Python training using one of the most efficient learning techniques: *practice testing*. This technique is guaranteed to improve your ability to read, write, and understand Python source code.

The idea is that you solve code puzzles. They start simple and become more and more complex as you read the book. In essence, you play the Python interpreter and compute the output of each code snippet in your head. Then you check whether you were right. Using the accompanying feedback and explanations, you will adapt and improve your coding skills over time. To make this idea a real-

²The third book is more advanced. It's about NumPy, Python's library for data science and numerical computation: <https://blog.finxter.com/coffee-break-numpy/>

³<https://blog.finxter.com/coffee-break-python/>

ity, I developed the online coding academy **Finxter.com**. The next section explains the advantages of the Finxter method of puzzle-based learning. If you already know about the benefits of puzzle-based learning from previous books, and you want to dive right into the puzzles, you can skip the following chapter and start at Chapter 5.

A Case for Puzzle-based Learning

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

Before diving into practical puzzle-solving, let’s first study 10 reasons why puzzle-based learning accelerates your learning speed and improves retention of the learned material. There is robust evidence in psychological science for each of these reasons. Yet, none of the existing coding books lift code puzzles to being first-class citizens. Instead, they mostly focus on one-directional teaching. This book attempts to change that. In brief, the 10 reasons for puzzle-based learning are:

1. Overcome the Knowledge Gap (Section 2.1)

2. Embrace the Eureka Moment (Section 2.2)
3. Divide and Conquer (Section 2.3)
4. Improve From Immediate Feedback (Section 2.4)
5. Measure Your Skills (Section 2.5)
6. Individualized Learning (Section 2.6)
7. Small is Beautiful (Section 2.7)
8. Active Beats Passive Learning (Section 2.8)
9. Make Source Code a First-class Citizen (Section 2.9)
10. What You See is All There is (Section 2.10)

2.1 Overcome the Knowledge Gap

The great teacher Socrates delivered complex knowledge by asking a sequence of questions. Each question built on answers to previous questions provided by the student. This teaching is more than 2400 years old and is still in widespread use today. A good teacher opens a gap between their knowledge and the learner's. The knowledge gap makes the learner realize that they do not know the answer to a burning question. This creates tension in the learner's mind. To close this gap, the learner waits for the missing piece of knowledge from the teacher. Better yet, the learner starts developing their own answers. The learner *craves knowledge*.

Code puzzles open an immediate knowledge gap. When you first look at the code, you do not understand the meaning of the puzzle. The puzzle's semantics are hidden. But only you can transform the unsolved puzzle into a solved one.

The problem of many teachers is that they open a knowledge gap that is too large. The learner feels frustrated because they cannot cross this gap. *Rated* code puzzles solve this problem because, by design, they are not too great a challenge. You must stretch yourself to solve them, but you can do it if you go all-out.

2.2 Embrace the Eureka Moment

Humans are unique because of their ability to learn. Fast and thorough learning has always increased our chances of survival. Thus, evolution created a brilliant biological reaction to reinforce learning in your body. Your brain is wired to seek new information; it is wired to always process data, to always learn.

Did you ever feel the sudden burst of happiness after experiencing a eureka moment? Your brain releases endorphins, the moment you close a knowledge gap. The instant gratification from learning is highly addictive and this addiction makes you smarter. Solving puzzles gives your brain instant gratification. Easy puzzles lead to harder puzzles, which open large knowledge gaps. Each one you solve, shortens the knowledge gap and you learn

in the process.

2.3 Divide and Conquer

Learning to code is a complex task. You must learn a myriad of new concepts and language features. Many aspiring coders are overwhelmed by this complexity. So they seek a clear path to mastery.

As any productivity expert will tell you: break a big goal into a series of smaller steps. Finishing each tiny step brings you one step closer to your big goal. *Divide and conquer* makes you feel in control and takes you one step closer to mastery.

Code puzzles do this for you. They break up the huge task of learning to code into a series of smaller steps. You experience laser focus on one learning task at a time such as *recursion*, the *for loop*, or *keyword arguments*. Each puzzle is a step toward your bigger goal of mastering computer science. Keep solving puzzles and you keep improving your skills.

2.4 Improve From Immediate Feedback

As a child, you learned to walk by trial and error—try, receive feedback, adapt, and repeat. Unconsciously, you will minimize negative and maximize positive feedback.

You avoid falling because it hurts, and you seek the approval of your parents. To learn anything, you need feedback so that you can adapt your actions.

However, an excellent learning environment provides you not just with feedback but with *immediate* feedback for your actions. If you were to slap your friend each time he lit a cigarette—a not overly drastic measure to save his life—he would quickly stop smoking. Puzzle-based learning with this book offers you an environment with immediate feedback. This makes learning to code easy and fast. Over time, your brain will absorb the meaning of a code snippet quicker and with higher precision. Learning this skill will take you to the top 10% of all coders.

2.5 Measure Your Skills

Think about an experienced Python programmer you know. How good are their Python skills compared to yours? On a scale from your grandmother to Bill Gates, where is that person and where are you? These questions are difficult to answer because there is no simple way to measure the skill level of a programmer. This creates a problem for your learning progress. The concept of being a good programmer has become fuzzy and diluted. What you can't measure, you can't improve.

So what should be your measurable goal when learning to program? To answer this, let's travel briefly to the

world of chess. This sport provides an excellent learning environment for aspiring players. Every player has an Elo rating number that measures their skill level. You get an Elo rating when playing against other players—if you win, your Elo rating increases. Victories against stronger players lead to a higher increase in the Elo rating. Every ambitious chess player simply focuses on one thing: increasing their Elo rating. The ones that manage to push their Elo rating very high, earn grandmaster titles. They become respected among chess players and in the outside world.

Every chess player dreams of being a grandmaster. The goal is as measurable as it can be: reaching an Elo of 2400 and master level (see Section 3). Thus, chess is a great learning environment as every player is always aware of their skill level. A player can measure how their decisions and habits impact their Elo number. Do they improve when sleeping enough before important games? When training opening variants? When solving chess puzzles? What you can measure, you can improve.

The main idea of this book and the associated learning app **Finxter.com** is to transfer this method of measuring skills from the chess world to programming. Suppose you want to learn Python. The Finxter website assigns you a rating that reflects your coding skills. Every Python puzzle has a rating number according to its difficulty level. You ‘play’ against a puzzle at your difficulty level. The puzzle and you will have a similar Elo rating so that your learning is personalized. If you solve the puzzle, your Elo

increases and the puzzle's Elo decreases. Otherwise, your Elo decreases and the puzzle's Elo increases. Hence, the Elo ratings of the difficult puzzles increase over time. But only learners with high Elo ratings will see them. This self-organizing system ensures that you are always challenged but not overwhelmed. Plus you constantly receive feedback about how good your skills are in comparison to others. You always know exactly where you stand on your path to mastery.

2.6 Individualized Learning

Today, the education system is built around the idea of classes and courses. In these environments, all students consume the same learning material from the same teacher applying the same teaching methods.

In the digital era, however, computer servers and intelligent machines provide individualized learning with ease. 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. Finding the perfect learning material for each learner is an important and challenging problem. Finxter uses a simple but effective solution to solve this problem: the Elo rating system. The student solves puzzles at their skill level. This book and the book's web backend Finxter push teaching towards individualized learning.

2.7 Small is Beautiful

The 21st century has seen the rise of microcontent. Microcontent is a short and accessible piece of information such as the weather forecast, a news headline, or a cat video. Social media giants like Facebook and Twitter offer a stream of never-ending microcontent. Microcontent has many benefits: the consumer stays engaged and it is easily digestible in a short amount of time. Each piece of microcontent pushes your knowledge horizon a bit further. Today, millions of people are addicted to microcontent.

However, this addiction will become a problem. The computer science professor, Cal Newport, shows in his book *Deep Work* that modern society values deep work more than shallow work. Deep work is a high-value activity that needs intense focus and skill. Examples of deep work are programming, writing, or researching. On the other hand, shallow work is a low-value activity that can be done by anybody e.g., posting cat videos to social media. The demand for deep work has grown with the rise of the information society. At the same time, the supply has stayed constant or decreased. One reason for this is the addictiveness of shallow social media. People that see and understand this trend can benefit tremendously. In a free market, the prices of scarce and high-demand resources rise. Because of this, surgeons, lawyers, and software developers earn \$100,000+ per year. Their work cannot easily be replaced or outsourced to unskilled work-

ers. If you can do deep work and focus your attention on a challenging problem, society pays you generously.

What if we could marry the concept of microcontent with deep work? This is the promise of puzzle-based learning. Finxter offers a stream of self-contained microcontent in the form of hundreds of small code puzzles. But instead of just being unrelated nonsense, each puzzle is a tiny stimulus that teaches a coding concept or language feature. Hence, each puzzle pushes your knowledge *in the same direction*.

Puzzle-based learning breaks the big goal of *reach mastery level in Python* into tiny actionable steps: solve and understand one code puzzle per day. A clear path to success.

2.8 Active Beats Passive Learning

Robust scientific evidence shows that active learning doubles students' learning performance. In a study on this matter, test scores of active learners improved by more than a grade compared to their passive learning counterparts.¹ Not using active learning techniques wastes your time and hinders you if you want to reach your full potential. Switching to active learning is a simple tweak that instantly improves your performance.

¹ https://en.wikipedia.org/wiki/Active_learning#Research_evidence

How does active learning work? Active learning requires the student to interact with the material, rather than simply consume it. It is student-centric rather than teacher-centric. Great active learning techniques are asking and answering questions, self-testing, teaching, and summarizing. A popular study shows that one of the best learning techniques is *practice testing*.² In this technique, you test your knowledge before you have learned everything. Rather than *learning by doing*, it's *learning by testing*.

The study argues that students must feel safe during these tests. Therefore, the tests must be low-stakes, i.e., students have little to lose. After the test, students get feedback on how well they did. The study shows that practice testing boosts long-term retention by almost a factor of 10. So, solving a daily code puzzle is not just another learning technique—it is one of the best.

Although active learning is twice as effective, most books focus on passive learning. The author delivers information; the student passively consumes the information. Some programming books include active learning elements by adding tests or by asking the reader to try out the code examples. Yet, I've always found this impractical when reading on the train, bus or in bed. If these active elements drop out, learning becomes 100% passive again.

Fixing this mismatch between research and common practice drove me to write this book about puzzle-based learn-

² <http://journals.sagepub.com/doi/abs/10.1177/1529100612453266>

ing. In contrast to other books, this book makes active learning a first-class citizen. Solving code puzzles is an inherent active learning technique. You must figure out the solution yourself for every single puzzle. The teacher is as much in the background as possible—they only explain the correct solution if you couldn't work it out yourself. But before telling you the correct solution, your knowledge gap is already ripped wide open. Thus, you are mentally ready to digest new material.

2.9 Make Code a First-class Citizen

Each chess grandmaster has spent tens of thousands of hours looking at a nearly infinite number of chess positions. Over time, they develop a powerful skill: the intuition of the expert. When presented with a new position, they can name a small number of strong candidate moves within seconds. They operate on a higher level than normal people. For normal people, the position of a single chess piece is one chunk of information. Hence they can only memorize the position of about six chess pieces. But chess grandmasters view a whole position or a sequence of moves as a single chunk of information. This extensive training and experience has burned strong patterns into their biological neural networks. Their brain can hold much more information—a result of the good learning environment they have put themselves in.

Chess exemplifies principles of good learning that are valid in any field you want to master.

First, transform the object you want to learn into a stimulus and then look at that it over and over. In chess, study as many chess positions as you can. In math, read mathematical papers containing theorems and proofs. In coding, expose yourself to lots of code.

Second, seek feedback. Immediate feedback is better than delayed feedback. However, delayed feedback is still better than no feedback at all.

Third, take your time to learn and understand thoroughly. Although it is possible to learn on-the-go, you will cut corners. The person who prepares beforehand always has an edge. In the world of coding, some people recommend learning by coding practical projects and doing nothing more. Chess grandmasters, sports stars, and intelligent machines do not follow this advice. They learn by practicing small parts again and again until they have mastered them. Then they move on to more complex bits.

Puzzle-based learning is code-centric. You will find yourself staring at the code for a long time until the insight strikes. This creates new synapses in your brain that help you understand, write, and read code fast. Placing code in the center of the learning process means you will develop an intuition as powerful as the experts. *Maximize the learning time you spend looking at code rather than other things.*

2.10 What You See is All There is

My professor of theoretical computer science used to tell us that if we stare long enough at a proof, the meaning will transfer to our brains by osmosis. This fosters deep thinking, a state of mind where learning is more productive. In my experience, his staring method works—but only if the proof contains everything you need to know to solve it. It must be self-contained.

A good code puzzle beyond the most basic level is self-contained. You can solve it by staring at it until your mind follows your eyes, i.e., your mind develops a solution from rational thinking. There is no need to look things up. If you are a great programmer, you will find the solution quickly. If not, it will take more time but you will still find the solution—it is just more challenging.

My gold standard was to design each puzzle such that they are (mostly) self-contained. However, to ensure you learn new concepts, puzzles must introduce new syntactical language elements as well. Even if the syntax in a puzzle challenges you, develop your own solutions based on your imperfect knowledge. This probabilistic thinking opens the knowledge gap and prepares your brain to receive and digest the explained solution. After all, your goal is long-term retention of the material.

— 3 —

The Elo Rating for Python

Pick any sport you’ve always loved to play. How good are you compared to others? The Elo rating answers this question with surprising accuracy. It assigns a number to each player that represents their skill in the sport. The higher the Elo number, the better the player.

Table 3.1 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’ll describe how you can use this book to test your Python skills.

3.1 How to Use This Book

This book contains 127 code puzzles and explanations to test and train your Python skills. The puzzles start from beginner-level and become gradually harder. At the end, you will be an intermediate-level coder. The Elo ranges

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 3.1: Elo ratings and skill levels.

3.2. HOW TO TEST AND TRAIN YOUR SKILLS? 19

from 987 to 1899 points (between *beginner* and *professional* level in the table). Follow-up books cover more advanced levels. This book is perfect for you if you are between the beginner and intermediate levels. Yet, even experts will improve their speed of code understanding if they follow the outlined strategy.

3.2 How to Test and Train Your Skills?

I recommend solving at least one code puzzle every day, e.g. as you drink your morning coffee. Then spend the rest of your learning time on real projects that matter to you. The puzzles guarantee that your skills will improve over time and the real project brings you results.

To test your Python skills, do the following:

1. Track your Elo rating as you read the book and solve the code puzzles. Write your current Elo rating in 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 `finxter.com`, start with that. Figure 3.2 shows five different examples of how your Elo will change while working through the book. Two factors impact the final rating: how you select your

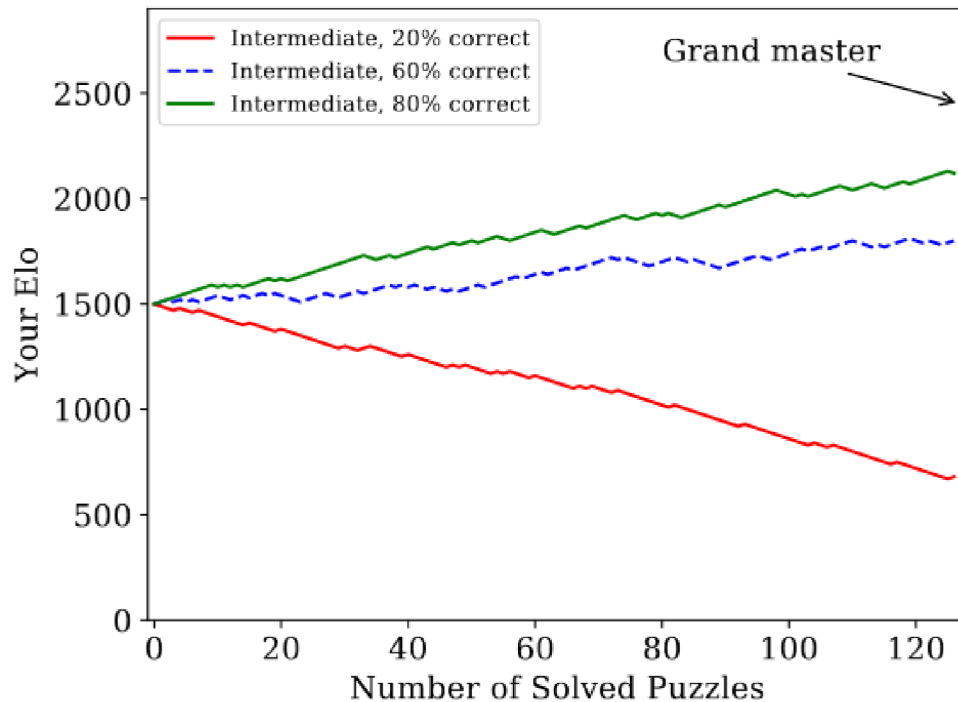


Figure 3.1: This is an example of how your Elo rating could change while working through the 127 puzzles. Your final Elo rating depends on where you started and (to a larger extent) on the percentage of correctly solved puzzles.

initial rating and how good you perform (the latter being more important).

2. If your solution is correct, add the Elo points given with the puzzle. Otherwise, subtract the points from your current Elo number.

3.2. HOW TO TEST AND TRAIN YOUR SKILLS? 21

Solve the puzzles sequentially because they build upon each other. Advanced readers can solve them in the sequence they wish—the Elo rating will work just as well.

Use the following training plan to develop a strong learning habit with puzzle-based learning.

1. Choose or create a daily trigger after which you'll solve code puzzles for 10 minutes. For example, decide on your *Coffee Break Python*. Or solve code puzzles as you brush your teeth or sit on the train to work, university, or school.
2. Scan over the puzzle 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 may seem counter intuitive at first but to learn faster, you must take your time and allow yourself to dig deep. There is no shortcut.
4. Stay objective when evaluating your solution—we all tend to lie to ourselves.
5. Look up the solution and read the explanation with care. Do you understand every aspect of the code? Write any questions down that you have and look up the answers later. Or send them to me (admin@

`finxter.com`). I will do everything I can to come up with a good explanation.

6. If your solution was 100% correct—including whitespaces, data types, and formatting of the output—you get the Elo points for this puzzle. Otherwise, your solution was wrong and you should subtract Elo points. This rule is strict because code is either right or wrong. If you miss some whitespace in the 'real world', you may get an error. So make sure you don't forget it in training.

As you follow this simple training plan, your ability to quickly understand source code will improve. In the long term, 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 with your effort. If you get off track, get right back on the next day. When you run out of code puzzles, feel free to checkout `Finxter.com`. It has more than 300 hand-crafted code puzzles and I regularly publish new ones.

3.3 What Can This Book Do For You?

Before we dive into puzzle-solving, let me address some common misconceptions about this book.

The puzzles are too easy/too hard. This book is for you if you already have some coding experience. Your Python skill level ranges from beginner to intermediate. If you are already an advanced coder, this book is for you too—if you read it differently. Simply measure the time you need to solve the puzzles and limit your solution time to 10–20 seconds. This introduces an additional challenge for solving the puzzles: time pressure. Solving puzzles under time pressure sharpens your rapid code understanding skills even more. Eventually, you will feel that your coding intuition has improved. If the puzzles are too hard, great. Your knowledge gap must be open before you can effectively absorb information. Just take your time to thoroughly understand every bit of new information. Study the cheat sheets in Chapter 4 properly.

Why is this book not conventionally structured by topic? The puzzles are sorted by Elo and not structured by topic. Puzzles with a small Elo rating are easier and more fundamental. Puzzles with a higher Elo rating are harder. To solve them, you need to combine the fundamental lessons from the easier puzzles. Ordering puzzles by difficulty has many advantages one being that you can solve puzzles at your skill level. As you get better, the puzzles get harder. Finally, ordering by complexity allows us to combine many topics in a single puzzle. For example, a Python one-liner may use two topics: list comprehension and lambda functions.

Learning to code is best done via coding projects. I agree but it's only part of the truth. Yes, you can improve

your skills by diving into practical projects. But, as in every other discipline, your skills will quickly hit an upper limit. These limits come from a lack of understanding of the basic concepts. You cannot understand high level knowledge without first understanding the basic building blocks. Have you ever used machine learning techniques in your work? Without theoretical foundations, you are doomed. Theory pushes your ceiling upwards and gets rid of the limitations that hold you back.

Abraham Lincoln said: *“Give me six hours to chop down a tree and I will spend the first four sharpening the ax.”* Do not fool yourself into the belief that *just doing it* is the most effective road to reach any goal. You must constantly sharpen the saw to be successful in any discipline. Learning to code is best done via practical coding *and* investing time into your personal growth. Millions of computer scientists have an academic background. They know that solving hundreds or thousands of toy examples in their studies built a strong foundation.

How am I supposed to solve this puzzle if I do not know the meaning of this specific Python language feature? Guess it! Python is an intuitive language. Think about potential meanings. Solve the puzzle for each of them—a good exercise for your brain. The more you work on the puzzle, even with imperfect knowledge, the better you prepare your brain to absorb the puzzle’s explanation.

*How does this book interplay with the puzzles at **Finxter.com**?* My goal is to remove barriers to learning Python.

Thus, many puzzles are available for free online. This book is based on some puzzles available at Finxter, but it extends them with more detailed and structured information. Nevertheless, if you don't like reading books, feel free to check out the website.

Anyway, why do some people thrive in their fields and become valued experts while others stagnate? They read books in their field. They increase their value to the marketplace by feeding themselves valuable information. Over time, they have a huge advantage over their peers. They get opportunities to develop themselves even further. They enjoy much higher job satisfaction and life quality. Belonging to the top ten percent of your field yields hundreds of thousands of dollars throughout your career. However, there is a price to pay to unlock the gates of this world: invest in books and personal development. The more time and money you spend on books, the more valuable you become to the marketplace!

The Elo-based rating is not accurate.

The Elo rating will get more accurate the more puzzles you solve. Although only an estimate, your Elo rating is an objective measure to compare your skills with the skills of others. Several Finxter users have reported that the rating is fair and surprisingly accurate. It provides a good indication of where one stands in comparison to other Python coders. If you feel the rating is not accurate, ask yourself whether you are objective. If you think you are, please let me know so that I can improve this book

and the Finxter back-end.

— 4 —

A Quick Overview of the Python Language

Before diving into the puzzles, work through the following five cheat sheets. By reading them, you'll learn 80% of Python's language features in 20% of the time it takes most people. They are definitely worth your time investment.

Read them thoroughly. If you try to understand every single line of code, you will catapult your skills to the next level. Most Python coders don't invest enough time into learning the basics such as the core language features, data types, and language tricks. Be different and absorb the examples in each cheat sheet. 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. Post them to your wall until you know them by heart:

<https://blog.finxter.com/subscribe/>.

4.1 Keywords

All programming languages reserve certain words to have a special meaning. These words are called *keywords*. With keywords, you can issue commands to the compiler or interpreter. They let you tell the computer what to do. Without keywords, the computer would not be able to understand 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:

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

The next cheat sheet introduces the most important keywords in Python. In each row, you'll find the keyword, a short description, and an example of its usage.

Keyword	Description	Code example
False, True	Data values from the data type Boolean	False == (1 > 2) True == (2 > 1)
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	x, y = True, False (x or y) == True # True (x and y) == False # True (not y) == True # True
break	Ends loop prematurely	while(True): break # no infinite loop print("hello world")
continue	Finishes current loop iteration	while(True): continue print("43") # dead 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 class method, first parameter is implicit.	class Beer: def __init__(self): self.content = 1.0 def drink(self): self.content = 0.0 # constructor creates class becks = Beer() # empty beer bottle becks.drink()
if, elif,	Conditional program execution: program starts	x = int(input("your val: ")) if x > 3: print("Big")

<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</code> <code>declaration</code> <code>for i in [0,1,2]:</code> <code>print(i)</code>	<code># While loop - same</code> <code>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>

4.2 Basic Data Types

Many programmers call basic data types *primitive data types*. They provide the primitives on which the 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 4.3.

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 ∈ {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 being the real number and the second the imaginary number). Third, the *string* data type 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
Boolean The Boolean data type is a truth value, either <code>True</code> or <code>False</code>. These are important Boolean operators ordered by priority (from highest to lowest): <code>not x</code> → <i>“if x is False, then x, else y”</i> <code>x and y</code> → <i>“if x is False, then x, else y”</i> <code>x or y</code> → <i>“if x is False, then y, else x”</i>	<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>
Integer An integer is a positive or negative number without floating point (e.g. <code>3</code>). Float A float is a positive or negative number with floating point precision (e.g. <code>3.14159265359</code>). The ‘//’ operator performs integer division. The result is an integer value that is rounded towards the smaller integer number (e.g. <code>3 // 2 == 1</code>).	<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 <code>'Yes'</code> 2. Double quotes <code>"Yes"</code> 3. Triple quotes (multi-line) <code>"""Yes We Can"""</code> 4. String method <code>str(5) == '5' # True</code> 5. Concatenation <code>"Ma" + "hatma" # 'Mahatma'</code> These are whitespace characters in strings. • Newline <code>\n</code> • Space <code>\s</code> • Tab <code>\t</code>	<pre>## 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</pre>
--	---

4.3 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 holds 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 for complex data types in the following cheat sheet.

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

CHAPTER 4. A QUICK OVERVIEW OF THE PYTHON LANGUAGE

36

Unordered collection of unique elements (<i>at-most-once</i>).	<pre>same = set(['apple', 'eggs', 'banana', 'orange']) print(basket == same) # True</pre>
Dictionary A useful data structure for storing (key, value) pairs.	<pre>calories = {'apple' : 52, 'banana' : 89, 'choco' : 546}</pre>
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.	<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>
Dictionary Looping You can access the (key, value) pairs of a dictionary with the items() method.	<pre>for k, v in calories.items(): print(k) if v > 500 else None # 'chocolate'</pre>
Membership operator Check with the keyword in whether the set, list, or dictionary contains an element. Set containment is faster than list containment.	<pre>basket = {'apple', 'eggs', 'banana', 'orange'} print('eggs' in basket) # True print('mushroom' in basket) # False</pre>
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	<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.	<pre>## Set comprehension squares = { x**2 for x in [0,2,4] if x < 4 } # {0, 4}</pre>
---	--

4.4 Classes

Object-oriented programming (OOP) is an influential, powerful, and expressive programming paradigm. 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.

Say, you are programming a game to build, simulate, and grow cities. In object-oriented programming, you would 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 `calculate_monthly_earnings`. This simplifies the reading and understanding of your code for other programmers. Even more important, you can now easily divide responsibilities between programmers. For example, you code the buildings and your colleague codes the moving cars.

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