Brain Games Python

#1 Bestselling Book Series LeanPub 2019

99 Brain Teasers to Energize Your Brain Cells and Python Logic Skills

Dr. Mayer, Rieger, Dr. Riaz

2019, Edition

Publisher: *Finxter | Coffee Break Python*
Editor: *Anna Altimira Groß*
Cover design: *Zohaib Riaz, Christian Mayer*
Newsletter: Subscribe, download your free Python cheat sheets, and become a better coder with a free puzzle a day https://blog.finxter.com/subscribe/.
Instagram: Get Your Free Daily "Espresso Break Python" by Lukas Rieger: https://www.instagram.com/finxter.com_/

If you want to rate this book, please always add a short text comment. Did you like it? What can be improved? Who would you recommend it to? Without a text comment, your star rating will be invisible on the Amazon website.

Brain Games Python

99 Brain Teasers for Beginners to Energize Your Brain Cells and Python Logic Skills

Christian Mayer, Lukas Rieger, and Zohaib Riaz

2019

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

Contents

— 1 —

Programming Your Intelligence

According to psychological research, your general intelligence divides into *fluid intelligence* and *crystallized intelligence*. [36] Fluid intelligence is the ability to quickly adapt your thinking to solve new problems. Crystallized intelligence is the ability to apply learned knowledge and experience. While you can increase crystallized intelligence by collecting new experiences and acquiring more and more knowledge [14], fluid intelligence was long considered to be a rather static factor which you cannot change. But taking a stand against the common belief of his peers, Professor Walter Perrig from the University of Bern showed in a famous 2008 article [12] that fluid intelligence can be trained by solving small working memory tasks over a relatively short period. In short: Brain puzzles make you smarter!

The purpose of this textbook is to improve both your fluid and your crystallized intelligence—the former by presenting you with focused working memory tasks and the latter by teaching you new knowledge in the area of programming and logic. It's a book full of fun brain games which you can solve daily to keep your brain function healthy and

vivid. But it's also a Python learning book that teaches you to read and understand Python source code quickly—an enormously important skill that every master coder has acquired as a result of studying myriads of code snippets.

Let's dive a bit deeper into the topic of intelligent behavior. While both fluid and crystallized intelligence are important predictors of personal success, they are not the only ways for you to access intelligence. Intelligence comes in many forms and is all around you. To maximize your results, you need to foster intelligent behavior of all systems you control: your body, your brain, your social environment, your organization, and your computational systems.

To understand what I mean with this, let's answer an important question first: What *is* intelligence anyways?

The dictionary Merriam-Webster answers this question by defining intelligence as

- *"the ability to learn or understand or to deal with new or trying situations"*,
- *"the ability to apply knowledge to manipulate one's environment"*, and
- *"the ability to perform computer functions"*.

From this definition, you can infer three types of intelligence—all of which are well-researched concepts:

- **Individual Intelligence** is about how fast you can learn, comprehend, and adapt to new environments and circumstances. [11] This type of intelligence is

best understood and known for thousands of years—even the ancient Greek philosopher Aristotle wrote about the constituents of individual *intelligence* [33]. You've already seen that it is divided into fluid and crystallized intelligence. It's the intelligence *within* yourself.

- **Collective Intelligence** is all about how you can tap into synergies, relationships, and the existing structures that are all around you to solve problems. [17] In his popular book *Programming Collective Intelligence*, Toby Segaran defined it as *"the combining of behavior, preferences, or ideas of a group of people to create novel insights."* [29] In other words, it's the intelligence that emerges *collectively* when a group of people works together towards solving a common problem.

- **Computing Intelligence**: This relatively new type of intelligence is least understood because it has existed only for a few decades. One of its early pioneers, Alan Turing, asked the thought-provoking question: *May not machines carry out something which ought to be described as thinking but which is very different from what a man does?* [35] Today, no one doubts the incredible power of computing intelligence with machines playing games, driving cars, and translating text beyond human-level performance. So you must ask yourself: How can you leverage the infinite computing power that is all around you?

A few decades ago, it would have sufficed for a person to master only the first type of intelligence and still succeed in life.

In a thought-provoking 2007 research paper, Jay Zagorsky asked the question: *Do you have to be smart to be rich?* [38] The researchers analyzed data from 1979 to 2004 and found that *"each point increase in IQ test score raises income by between* 234*and*616 *per year after holding a variety of factors constant."* In short: the higher your individual intelligence, the more money you earn.

Imagine you are living in the 1960s. You are a very intelligent individual who doesn't know a thing about collective and computing intelligence—the latter doesn't even exist yet for the general public (many computers of the time cost five million dollars). You only sit in your chamber, write, and think about hard research problems. Yet, your individual intelligence opens many doors and chances are that you'll become a very successful individual—because of your superior intelligence.

This would never work today: As the world becomes more and more interconnected—with ubiquitous social media, improved mobility, and high-speed communication—you also need to master the art of collaboration to reach your goals (collective intelligence). Think about your environment: Is there a person who excels in this second type of intelligence while being average in the first type? Most of us know at least one such person.

But it doesn't stop there. A new type of intelligence emerges with the power of surpassing all others. You know intuitively how to use your mind (individual intelligence) and the minds of others (collective intelligence). But if you are like most people, you have a very bad intuition regarding how to use the mind of computers.

The third type of intelligence, computing intelligence, only

existed for a few decades—a time frame too short to leave any biological traces like the other two types: Evolution couldn't adapt our brains to account for this third type of intelligence. And yet, it is widely recognized as one of the most powerful sources of influence in the 21st century: Even Merriam-Webster includes it into the official definition of intelligence.

The most successful individuals today master not one, but all three types of intelligence. Consider for example Bill Gates, Jeff Bezos, and Elon Musk. They all nourish their individual intelligence[1]—for example, by reading a massive amount of books. They all leverage collective intelligence—for example, by creating organizations with tens of thousands of smart people working together towards common goals. And they all rely on computing intelligence to a degree that has never been seen before in the history of the world.

A sure way to become successful in your own life is to tap into *all* of these three powerful sources of intelligence. If you understand how you can increase your individual, collective, and computing intelligence, your life will pick up speed and soon blast off. Soon, you'll create massive value for society, enjoy monetary success way beyond your wildest dreams, and create a healthy and thriving environment for you and your family.

And the best news is: All three types of intelligence are trainable. There are many ways of increasing your individual intelligence and, even more importantly, your intelligent behavior by leveraging the power of habits. The second type of intelligence, your social intelligence, is trainable

[1]To be more specific: their crystalline intelligence.

even to a much greater degree than the first type. After all, you are a social animal: Seek out the interaction with other people, read books about effective communication, listen, and learn to present your ideas to leverage collective intelligence. Finally, the third type is also trainable—even more so than the other two types: Today, everyone can learn to code and control computing intelligence to harvest its full potential. And it's much easier to train your IQ.

While you need to go out in the world to enhance your collective intelligence, this book will help you improve the other two types of intelligence. In other words, this book is about individual and computing intelligence. While some books cover individual intelligence (*"Improve your IQ"*) and others cover computing intelligence (*"Improve your coding skills"*), this book promises to improve *both* your individual and your computing intelligence at the same time.

1.1 Intended Audience

This book is based on the puzzle-based learning approach developed for the best-selling "Coffee Break Python" textbook series. I'll introduce puzzle-based learning in the next section.

The great response of our readers encouraged us to write this fifth book for absolute beginners based on the idea of learning Python with low-stake tests. Although this is the fifth book in the series, it's designed to be understandable for everyone—even if you have never even written a single line of code. It's for those who love brain games, quizzes, and puzzles to keep their brain fit.

The unique strength of this book is that it also teaches you

how to code basic Python programs—in a non-intrusive fun way that is accessible to everyone including non-technical readers. Learning to code is a *bonus* you'll get in addition to an improved working memory and logic skills.

1.2 Puzzle-based Learning to Code

The main part of the book consists of 1001 code puzzles. Let's have a look at the key idea of puzzle-based learning. This section answers the question: Why does solving code puzzles make you smarter?

> **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.

Here's an example of a code puzzle:

```
# Puzzle 5
a, b = True, False

out = (a and b) or (a and b and not a) or a
print(out)
# What's the output of this code snippet?
```

You don't have to understand the code puzzle, yet. We'll explain everything you need to know in a step-by-step manner later in the book. Just note that the code *"does"* something and you need to figure out what he's doing. In this book, we focus on logic-style code puzzles that you can solve

by rational thinking and which come with the look and feel of traditional "brain puzzles" to keep your thinking fresh.

Why is puzzle-based learning one of the best ways to improve your coding skills?

As you'll see, there is robust evidence in psychological science for the efficiency of puzzle-based learning. If you are already familiar with the other books of the *Coffee Break Python* series, you may skip the following sections.

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 wants to acquire the missing piece of knowledge. The learner *craves knowledge*.

Code puzzles open an immediate knowledge gap. When looking at the code, you first do not understand the meaning of the puzzle. The puzzle's semantics are hidden. However, if you study a puzzle long enough, you'll often find yourself experiencing a eureka effect. Your brain releases endorphins the moment you close a knowledge gap. The instant gratification from puzzle-solving is highly addictive, but this addiction makes you smarter.

Still, learning to code is a complex task. You must learn a myriad of new concepts and language features. Many aspiring coders are overwhelmed by complexity. They seek a clear path to mastery. As any productivity expert will tell you: Break a big task or 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, pushing you one step closer toward mastery.

Code puzzles do this for you by breaking up the huge task of

learning to code into a series of smaller learning units. You can digest one puzzle at a time. Each puzzle is a step toward your bigger goal of mastering computer science. Keep solving puzzles and you keep improving your skills.

To learn anything, you need feedback so that you can adapt your actions. However, an excellent learning environment provides you not only with feedback but with *immediate* feedback for your actions. If you were to slap your friend each time he lights a cigarette—a not overly drastic measure to save his life—he would quickly stop smoking (or, more likely, stop meeting you).

Puzzle-based learning offers you an environment with immediate feedback to speed up the process of learning to code.

Over time, your brain will absorb the meaning of a code snippet quicker and with higher precision this way. Learning this skill pushes you toward the top 10% of all coders.

Robust scientific evidence shows that active learning—that is, students actively participate in the learning process—doubles students' learning performance. In a study on that matter, test scores of active learners improve by more than one grade compared to their passive learning fellow students.[2] Not using active learning techniques wastes your time and hinders you in reaching your full potential in any area of life. Switching to active learning is a simple tweak that will instantly improve your performance when learning any subject.

Active learning requires the student to interact with the ma-

[2]Freeman, Scott, et al. "Active learning increases student performance in science, engineering, and mathematics." Proceedings of the National Academy of Sciences 111.23 (2014): 8410-8415.

terial, rather than simply consuming it. It is student- 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*.[3] In this learning technique, you test your knowledge even if you have not learned everything yet. 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-stake so that students have little to lose. After the test, students get feedback about the correctness of the tests. The study shows that practice testing boosts long-term retention of the material by almost a factor of ten. As it turns out, 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. But how can you try code if you're reading on the train, on the bus, or in bed? And if these active elements drop out, learning becomes 100% passive again.

Fixing this mismatch between research and common practice drove us to write this book based on puzzle-based learning. In contrast to other books, this book makes active learning a first-class citizen.

But there's also another angle to it: For example, each grandmaster of chess has spent tens of thousands of hours

[3] http://journals.sagepub.com/doi/abs/10.1177/1529100612453266

looking into 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. For normal people, the position of a single chess piece is one chunk of information so they can only memorize the position of a few chess pieces. But chess grandmasters view a whole position or a sequence of moves as a single chunk of information. They operate on a higher level than lesser chess players. The extensive training and experience have burned strong patterns into their biological neural networks. Their brains 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 to learn into a stimulus that you present to yourself over and over again. In chess, study as many chess positions as you can. In math, make it a habit to read mathematical papers with 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 much better than no feedback at all. Third, take your time to learn and understand thoroughly.

In the world of coding, some people recommend learning by coding practical projects and doing nothing more. Chess grandmasters do not follow this advice. They learn by practicing isolated stimuli again and again until they have mastered them. Then they move on to more complex stimuli—such as more complex chess positions.

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 whole learning process creates an environment in which you will develop the powerful intuition of the expert. Therefore: *Maximize the learning time you spend looking at code rather than at other stimuli.*

My professor of theoretical computer science used to argue that if you only stare long enough at a proof, the meaning will transfer into your brain 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 purely by staring at it until your mind follows your eyes—your mind develops a solution based on 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 can still find the solution—it is just more challenging. The gold standard in this book was to design each puzzle to be self-contained.

1.3 How to Read This Book

Think about an experienced Python programmer you know, e.g., your nerdy colleague or classmate. How good are their Python skills compared to yours? On a scale from your grandmother to Bill Gates, where is your colleague 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 severe problem for your learning progress: the concept of being a good programmer becomes fuzzy and diluted. What you can't measure, you can't improve.

So how can you possibly measure your learning progress? To answer this, let us once more travel to the world of chess, which happens to provide 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 ambitious chess player dreams of being a grandmaster. The goal is as measurable as it can be: reaching an Elo of 2500. Thus, chess is a great learning environment—every player is always aware of their skill level. A player can measure how decisions and habits impact their Elo rating. How does sleep influence their game performance? How does training opening variants? How does solving chess puzzles? What you can measure, you can improve.

The main idea of this book series and the associated learning app `https://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 number that measures your coding skills. Every Python puzzle has a rating number as well, according to its difficulty level. You 'play' against a puzzle at

your difficulty level: The puzzle and you will have more or less the same Elo rating so that you can enjoy personalized learning. 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.

This self-organizing system ensures that you are always challenged but not overwhelmed, while you constantly receive feedback about how good your skills are in comparison with others. You always know exactly where you stand on your path to mastery.

This book is an extension of the `https://Finxter.com` website. It provides you with 1001 (!) brain puzzles specifically in the math and logic area. Each puzzle has an associated Elo rating. Initially, solving each puzzle will take time. But as you go through the puzzles, you'll become faster and faster—feel free to measure your time! Ultimately, you'll look at a piece of simple code and the meaning will immediately "pop into your head".

Table 1.1 shows the ranks for each Elo rating level. The table is an opportunity for you to estimate your logic skill level.

1.4 How to Test and Train Your Skills?

I recommend solving at least one or two brain puzzles every day, e.g., as you drink your morning coffee. If you want to track your logic skills, use the following simple method.

1. Track your Elo rating as you read the book and solve

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 1.1: Elo ratings and skill levels.

the 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 would be the most precise option.

2. 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.

Solve the puzzles in a sequential manner because they gradually become harder. Advanced readers can also solve puzzles in the sequence they wish—the Elo rating will work as well.

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

1. Select a daily trigger after which you solve 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 is no shortcut.

4. Stay objective when evaluating your solution—we all tend to fake ourselves.

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 (`admin@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—you get Elo points for this puzzle. Otherwise, you should count it as a wrong 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.

As you follow this simple training plan, your skill to see through source code quickly 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 effort. If you get off track, get right back on track the next day.

When you run out of code puzzles, feel free to checkout `https://Finxter.com` with more than 300 hand-crafted code puzzles. I regularly publish new code puzzles on the website as well.

— 2 —

How to Boost Your Intelligence? 10 Tips From Science

The introductory chapter shows that intelligent behavior is a prerequisite of your success. There are many different ways of tapping into intelligence. This chapter provides you with tips to help you become a more intelligent human being. Ignore them at your own risk!

2.1 Sleep More

Sleep deprivation reduces your intelligence [15]. Say, you decide to sleep only six instead of eight hours from now on. Congratulations, you've gained 8% more time which you could theoretically use to be more productive. However, scientific research papers prove that if you sleep less than seven hours, you'll become dumber and less healthy: you'll develop diabetes, heart disease, stroke, depression, and die earlier [1]. Reducing your daily sleep intake will reduce your productivity. So sleeping for eight hours is one of those low-hanging fruits which you'd be crazy to ignore.

2.2 Don't Do Drugs

While it's hard to increase your intelligence, it's very easy to decrease it. Medical science shows that alcohol consumption negatively correlates with IQ measurements [20]. Chronic excessive drinkers are more likely to develop cognitive deficits [26]. An extensive meta-study based on 21 research papers [8] shows that young people who consume alcohol regularly have smaller volumes of grey and white matter in specific brain areas. The volume of white matter is known to be associated with reduced IQ values [21].

Those studies indicate that alcohol consumption correlates with reduced IQ values. Similar results can be observed for other types of drugs. Consuming five joints of Marijuana per week reduces your IQ by 5.1 points as discovered in a study on seventy young adults in their teens and twenties [9]. But there's good news: the negative effect on IQ seems to be only temporary—so if you'd stop consuming Marijuana now, you've got a good chance of recovering from your reduced IQ level.

It's, therefore, a no-brainer (literally) to remove all types of drugs from your life. You'll quickly observe a positive effect on your daily intelligence and clarity of thinking.

2.3 Eat Brain Food

A 2017 study shows that high levels of Omega-3 improve blood flow in your brain [2]. Thus, eating Omega-3 rich foods such as fish and flax seeds may boost your brain function. A large body of research confirms that antioxidants reduce your chance of developing Alzheimer's and prevent a reduction of your cognitive function with old age [25].

Consuming foods (fruits and vegetables) that are rich with antioxidants has numerous other health benefits such as reducing the degenerative impact of free radicals in your body and, particularly, your brain [13]. A very robust research finding is that lifelong coffee consumption reduces the negative effects of aging and prevents brain diseases such as Alzheimer's and Parkinson's [22]. Roughly speaking, eat more fruits, berries, vegetables, nuts, cocoa, coffee, and soy products and lead a more intelligent life.

2.4 Meditate

There are many positive effects associated with meditation. For example, meditation decreases your stress level and increases your IQ [32]. A study performed on 351 full-time working adults showed that there's a positive correlation (causal or not) between emotional intelligence and meditation [5]. Another study shows that a short meditation course reduces depression, anxiety, and stress [28]. Yet another study shows that meditation can reduce the decline of fluid intelligence when aging [10]. Last but not least, meditation is scientifically proven to improve your sleep quality and has, therefore, positive effects on your cognitive abilities in multiple dimensions [3]. So if you want to increase your intelligent behavior while having a calmer and less stressful life, why not meditate for twenty minutes every day?

2.5 Exercise

Physical exercise has a (modest) positive correlation with IQ [16]. But is it a causal relation or do intelligent people

simply exercise more? In fact, a 2012 study discovers a causal relationship: people who exercise more can improve their intelligence and reduce the cognitive decline with old age [23]. With countless additional (health) benefits such as increased life expectancy, health, and overall quality of life [24], injecting a dose of physical exercise into your day is another powerful ingredient towards a more intelligent form of living.

2.6 Read More

People who read more books tap into the knowledge of the world. For almost any problem you encounter in life, another human being has already spent decades researching it—and writing a book about how to solve it.

Imagine you'd start reading one book per week in your field (say, you are a programmer). You don't have to read scientific studies to know that your work success will skyrocket. Because of this one habit, you'll quickly become one of the most knowledgeable workers in your field. Joining the top 1% in your field will be highly profitable—it doesn't matter which field you are in. If you assume that your pay is roughly proportional to your productivity (otherwise, change your job or become self-employed), your income will start to grow rapidly. Just by implementing this one reading habit.

Successful leaders read multiple books per month: Warren Buffett, Bill Gates, Mark Zuckerberg, Jeff Bezos. Reading books for success, achievement, and self-development may easily be one of the main differentiating factors between the rich and the poor [6]. But even pleasure reading has been

shown to have a positive impact on academic performance [37].

You may argue that you don't have a lot of time reading books. But this is unlikely to be true: today's youth spends 2-3 hours per day watching television and playing games on the computer [4]. So why not replace 30-40 minutes of your screen time with reading books in your field (or listen to audiobooks while being in commute)? This simple habit will bring great success in your life.

2.7 Set Goals

Here's another factor for success: set goals. A wildly popular psychological research paper [18] investigated the performance of tens of thousands of people with different ages, nationalities, and varying demographic factors. The result: setting goals is a robust predictor of many success measures—especially monetary and production-related goals.

It doesn't matter how you approach goal-setting. Simply decide now for one goal-setting approach and keep doing it for the rest of your life. For example, every morning when you start with your work, write down your life goals, 5-year goals, 1-year goals, 1-month goals, and daily goals. A powerful strategy for success and a great way of fostering intelligent behavior in your everyday life!

2.8 Focus

No matter how smart you are, if you spread your focus, you'll reduce your intelligent behavior and problem-solving skills. Say your goal is to become a great coder. But you

also want to learn the piano, play football and chess, study four different languages, become a master cook, have five kids, and travel the world. At the same time, your friend Alice focuses on coding only. She will outperform you while you won't reach mastery level in any of your endeavors: in every single field, there will be people who focus on this field and this field only. Those people will crush it while you'll become average at best. Alice spends ten hours every day studying code, and you can invest only one or two hours per day given your diverse focus. Time is your scarcest resource. There's a famous Russian saying: *if you chase two rabbits, you will catch neither one.*

So what to do? The answer is simple: focus on one thing. Any great master focuses their time and energy on one field. Focus on one career, one marriage, one life. Studies confirm that increased focus leads to a higher level of creative performance [30]. Focus is a powerful predictor of success for individuals who push forward companies, open-source projects, and social media presence [34]. The intelligent professional focuses on one thing: becoming a more intelligent human-being by implementing these tips.

2.9 Play Your Strengths

There is more to be gained by improving your strengths than by mitigating your weaknesses. Why? Because you've got a lot more weaknesses than strengths. If you try to overcome all weaknesses, you'll become average in some of them. Does this sound like a strategy for success and intelligent behavior? Follow the advice of management genius Peter Drucker: perform from your strengths [7]. Strength-based training approaches are usually employed when train-

ing mental toughness of high-performance athletes [31].

Playing strengths is good advice for organizations, too. An organization where people can play their strengths develops a higher level of collective intelligence than organizations where each person must compensate for their own weaknesses. In the former organization, there's always another person who compensates your weaknesses with their strengths—this is the source of synergy. There's nothing as powerful as a group of people compensating for each other's weaknesses with each person's strengths. This way, one plus one can truly become three.

2.10 Play Brain Games

Brain games can improve fluid intelligence as discovered in a highly popular 2008 article [12]. The article has had a lasting impact on the science of brain games— thousands of follow-up articles discussed the extent of the concrete benefits of brain games. While there hasn't been a conclusive study about *how much* brain games improve fluid intelligence, it is undisputed that brain games improve performance on tasks that are similar to the ones that are trained [19].

Millions of people today solve brain games to keep their brain fit while having fun in the process. A large part of the performance improvement addresses cognitive performance in specific tasks (these improvements are *non-transferable*). However, these specific tasks are often not important for practical matters. After all, how does it help you in practice to be able to solve Sudokus? This book channels the non-transferable part of improvement towards a specific task

with high-value: programming. By tapping into computational intelligence in combination with improving your individual intelligence, you'll experience meaningful benefits in your life.

— 3 —

Absolute Computer Science Basics for Non-Coders

3.1 What is Program Code?

Program code is a list of instructions stored in files and executed by a computer. You need to write these instructions in a language that computers understand. So, which language do computers speak? It's called binary: every word is nothing but a sequence of 0 and 1. For example, the binary representation of word `'Python'` is 01010000 01111001 01110100 01101000 01101111 01101110. Because the word `'Python'` consists of six letters, there are six blocks of 0 and 1, each encoding one letter. For example, the binary sequence 01010000 encodes the character `'P'`. We call this binary code ASCII.

Rather than writing program code in binary, we use high-level programming languages. The *compiler* translates your code, written in a programming language, into executable binary code. Over the years many programming languages have been developed, each one with its special purpose. Yet,

Python has come to be the most popular one on the planet. In the last years, it has seen enormous growth due to its simplicity and wide range of applications such as machine learning.

3.2 How to Read Program Code?

Reading program code is easy and since you are reading a book, you already know the basics. Just like text, you read program code from top left to bottom right, line by line. Each line contains instructions that use certain keywords defined by the programming language. For example, some Python keywords are: `if`, `else`, `int`, `while`, `def`, `class`, `float`. To get you on track as fast as possible, we focus only on the keywords used in the puzzles in this book. But first, we have to look at three broader concepts.

3.3 Variables

Imagine a variable as a box in which you can put things. In programming these things are values like 1, 2, 3 or `'word'`. Of course, the computer doesn't store these values in boxes but in the memory. We say: *A variable points to a memory (RAM) address that stores a value.* Now, let's write all this as Python code:

```
my_variable = 1
my_other_var = 3
```

What exactly is happening here? The first variable named `my_variable` contains the value 1. The second variable

named `my_other_var` contains the value 3. To fill the variable boxes with values, we use the `=` operator and write the variable's name on the left and the value on the right. Programmers say: *we assign a value to a variable*. Congratulations, you just read your first two lines of code!

Variable can have different types, depending on the values stored in them. For example, the variables from the previous example are both integer variables since they hold integer values. Python defines the type of a variable as soon as you assign a value to it. In the following code snippet, we define three variables with different types.

```
float_var = 3.1415
string_var = 'this is text'
bool_var = True
```

We call the type of `float_var` *float* as indicated by the decimal point, the dot, in the number. We call the type of `string_var` *string* as it contains strings of characters. A string of characters can be a single character like `'a'`, `'b'`, `'!'`, `'?'`, entire words or even a whole text. We call type of `bool_var` *Boolean*. Such a variable contains either `True` or `False`. For this book, this is the most important type. Therefore, we will go more into depth about Boolean variables later.

You can also assign values to several variables in a single line as shown in the following code snippet:

```
a, b, c = True, False, True
```

This is a short version of:

```
a = True
b = False
c = True
```

We use this single line assignment of multiple variables throughout the book so make sure to get it.

3.4 Control Flow

Sometimes you need to repeat an instruction several times. Other times you need to execute an instruction only under certain conditions. For this purpose programming languages have *control flow* instructions which you will get to know next.

We use the keyword `if` to execute code only under certain circumstances. The Python syntax is: `if <condition>:`. In the following lines, you write the code you want to execute if the condition is `True`, for example, `if 5 > 1:`

Note that we indent the code by four white spaces: the whole indented block executes only under the given condition. To make this more tangible, imagine that you are working in a company 'SaferStreets Inc.'. Your goal is to develop a street sign which shows different colors based on the speed of passing cars. As long as the speed doesn't surpass 80 the sign shows green, only if the car drives faster than 80, the sign turns red. So you would write the following code:

```
speed = 50
color = 'green'
```

```
if speed > 80:
    color = 'red'
```

There are two variables `speed` and `color`. The former contains an integer and the latter a string. In case the integer value is larger than 80 we execute the code in the `if` statement which assigns the value `'red'` to variable `color`. Let's consider a more advanced example:

```
speed = 90
color = 'unknown'

if speed < 30:
    color = 'green'
elif speed < 80:
    color = 'orange'
else:
    color = 'red'
```

You just developed a new version of the digital street sign with an extra color! If `speed` is below 30, it shows `green`. If `speed` between 30 and 79, it shows `orange`. And only if `speed` is above 80, it turns `red`. That's great! And here comes your boss and asks you to remove the `orange` state from the digital street sign. So, you remove the `elif` part from the code. Now it looks like this:

```
    speed = 90
    color = 'unknown'

    if speed < 30:
        color = 'green'
```

```
else:
    color = 'red'
```

You can even simplify this further to:

```
speed = 90
color = 'red'

if speed < 30:
    color = 'green'
```

Can you see that the last two snippets do the same thing? To sum it all up, let's put it more abstractly:

```
if condition_1:
    # execute if condition 1 is true
elif condition_2:
    # execute if condition 2 is true
else
    # execute if conditions are false

# execute in any case
```

You may ask: what's the meaning of the hashtags? A hashtag in Python marks a comment line. It's not part of the code and the computer won't execute it. Still, comments are useful to explain your code to others (and to your future self reading the code in a year or so).

You can also nest `if` statements, like this:

```
speed = 50
age = 24
```

```
color = 'unknown'

if speed > 100:
    if age < 21:
        # too young
        color = 'red'
    elif age > 85:
        # too old
        color = 'red'
    else:
        color = 'orange'
else:
    # 100 and less is ok for everybody
    color = 'green'

# prints the value of color
print(color)
```

After the first couple of puzzles, this will become easy for you, almost like speaking your mother tongue.

To find out, what the last code snippet would print, proceed like this:

- Replace the variable names with their values (speed -> 50).
- Replace all comparisons with their truth values (`True`, `False`).
- Follow the `True` values through the code.

Do you know the output of the code snippet above?

Solution: `'green'`

3.5 Functions

Every programming language has *functions*. You can reuse those code snippets wherever you want in your code. For example, to write the text `'my text'` to the computer's screen, you call the function `print('my text')`. The values you pass to a function between the parenthesis are the function's arguments. There are many other such functions in Python. For this book, you only need to know the `print()` function and the functions `any()` and `all()`.

So what do functions `any()` and `all()` do? Function `all()` checks if all Boolean values passed to it are `True` and `any()` checks if at least one value is `True`. Both functions expect a list argument. In Python you create a list with the notation `[...]`. For example, the expression `[True, False, False]` creates a list of three Boolean values. You won't need this for the puzzles in this book. But if you dive deeper into Python programming, you'll see this all the time. If you read `all([a, b, c])` in the puzzles, think of it as: Are all the variables `a, b, c` `True`?

Let's look at an example:

```
a, b, c = True, True, False

d = all([a, b, c])
# d is False

e = any([a, b, c])
# e is True
```

Variable `d` contains the value `False` because not all values passed to the `all()` function are `True`. On the other hand,

variable `e` contains the value `True` because at least one value passed to the `any()` function is `True`. Now, check if you got this. Look at the puzzle below and try to find out which values variables `d` and `e` contain at the end.

```
# Puzzle
a, b, c = True, True, False

d = all([a, b])
e = any([c])
```

Solution: Variable `d` is `True` and variable `e` `False`.

3.6 Boolean Operators

We use *Boolean operators* to connect conditions or Boolean variables. Before you freak out, don't worry, you use Boolean operators all the time in everyday language. For example, when you say: If it rains and it's cold, I won't come to the party. There we have two Boolean variables, A - it rains - and B - it's cold. These two variables are connected with the Boolean operator `and`. So, if it rains but the weather is still warm, this gives us A = `True` and B = `False`. Let's consider 'I won't come to the party' as another variable, say C. Now, we can define the variable C as A `AND` B. Since A is `True` and B is `False`, C is also `False`. So, you have to do what? You have to come to the party! The following tables show the truth values for the operators `AND`, `OR` and `NOT`. Keep these truth tables in mind throughout the whole book!

3.7 Truth Tables

The connection of two Boolean variables with the `AND` operator returns `True` only if both inputs are `True`. In any other case this returns `False`. In formal logic, it is very common to represent such things as truth tables. A truth table has at least two columns, one for the input and one for the output. But in general, there are two or more inputs and one output. Since a truth table has to show the output for any combination of the inputs, for n inputs it has 2^n rows. Look at the following tables. The `NOT` table has one input value which can be either `True` or `False` and, therefore, it has two rows. If there are two different input values, such as in the case of `AND` and `OR`, the table has four rows. How many rows are in a table with three input values?

Now, it's easy to see that the functions `any()` and `all()` are shorthand for connecting all given parameters with `OR` or `AND`. For example, `all([True, True, False])` is the same as writing `True AND True AND False`. Both expressions return `False`.

The Negation Operator NOT

Truth table for the `NOT` operator.

A	NOT A
True	False
False	True

Table 3.1: The NOT operator

The AND Operator

Below you find the truth table for the `AND` operator.

A	B	A AND B
`True`	`True`	`True`
`True`	`False`	`False`
`False`	`True`	`False`
`False`	`False`	`False`

Table 3.2: The AND operator

Columns A and B show all the possible combinations of inputs. When you connect both of the inputs with the `and` operator you get the result displayed in the last column.

The OR Operator

The following table shows the truth values for the `OR` operator.

A	B	A OR B
`True`	`True`	`True`
`True`	`False`	`True`
`False`	`True`	`True`
`False`	`False`	`False`

Table 3.3: The OR operator

Again, we give all possible combinations of input values in columns A and B. The last column shows the result of the `or` operator with the given inputs. Note that `or` is also `True` if both input values are `True`. When you say 'or'

in everyday language, in general you mean either of both. So, in this case, the `or` operator is different from everyday language.

3.8 Advanced Boolean Operators

There are more than the three basic Boolean operators you saw in the previous section. Yet, we can emulate all operators not discussed here by combining the three basic operators. The advanced operators are not used in the puzzles in this book. However, we want to give you a short overview.

The Exclusive-Or (xor) Operator

First, there is the `xor`. `xor` represents: either A or B. In every day language this is mostly what you mean when you say 'or'. As you can see in the truth tables, `or` is is also `True` when both inputs are `True`, not so the `xor`. `xor` returns `True` only if exactly one input is `True`. We can translate this into: (A `and not` B) `or` (`not` A `and` B). Try to draw the truth tables and compare them.

Here you can see the truth table for the `XOR` operator. This is the way you probably use 'or' in everyday language.

A	B	A XOR B
True	True	False
True	False	True
False	True	True
False	False	False

Table 3.4: The XOR operator

Other advanced Boolean operators are the implication, written as `=>`, and equivalency, written as `<=>`. The equivalency is an implication in both directions. Or in a more formal way: `A <=> B` is the same as `A => B and B => A`. Using only basic operators, the implication can be written as `not` A `or` B.

The Implication

In this truth table, you see how the implication works.

A	**B**	**A => B**
`True`	`True`	`True`
`True`	`False`	`False`
`False`	`True`	`True`
`False`	`False`	`True`

Table 3.5: The Implication

3.9 Boolean Operators Precedence

As you know, multiplication and division precede addition and subtraction in calculus. Similarly, there are precedence rules for Boolean operators. The operator `not` always goes first, then comes `and` and `or` comes last. Let's take an example to make this clearer:
A `or` B `and not` C
is the same as
(A `or` (B `and` (`not` C))).
If you want to change the order or precedence you have to use parenthesis.

3.10 Happy Puzzle Solving!

After this brief introduction to the basics of computer science, you are good to go! Whenever you have doubts, you can use this chapter as a reference and come back. Happy puzzle solving!

— 4 —

Kindergarten Logic

When my two-year-old daughter started to form her first sentences, she already knew how to leverage basic if-then-else logic to get what she wanted. While I am not too fond of buying ice cream for my kids—because it contains unhealthy amounts of sugar—my daughter used to present her conditional logic so convincingly (*if I don't get ice cream, I'll cry*) that she regularly got what she wanted.

In this chapter, we start slowly with some basic if-then-else logic puzzles. I want to highlight that those puzzles are not hard to solve, but even advanced coders often need a lot of time to solve them. If you're slow in reading and understanding basic code snippets, this may easily be the one thing that holds you back the most. So, remove this barrier now!

For readability, we present you the solution for each puzzle on the subsequent page to prevent you from accidentally looking up the solution without thinking for yourself first.

```
# PUZZLE
a, b, c = False, True, False

if not b and a:
    print('python')
else:
    print('42')
```

```
# RESULT
'''
42
'''
# Your new Elo = Your old Elo +/- 12
```

```
# PUZZLE
a, b, c, d = True, False, True, True

if d or b and not d:
    print('love')
else:
    print('yes')
```