Coffee Break NumPy

A Simple Road to Data Science Mastery That Fits Into Your Busy Life

MAYER, RIAZ, RIEGER

Coffee Break Numpy

A Simple Road to Data Science Mastery
That Fits Into Your Busy Life

Christian Mayer, Zohaib Riaz, and Lukas Rieger

November 2018

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

Contents

— 1 —

Introduction

In the 21st century, a new skill penetrates every area of our lives. As you will see, it is one of the most powerful skills in the world. Harvard Business Review (HBR) labeled the profession that comes with this skill as the "sexiest job of the 21st century." You can use it for good (e.g., improving health of society) or for bad (e.g., hacking democracy via massive-scale social network manipulation). It threatens, directly or indirectly, every single human profession: millions of professional drivers, factory workers, writers, medical doctors, researchers, teachers, coders, retailers, sales people, and small business owners. Those people and many more may lose their job—only because of this one skill. Why? Because it has the power to create machines that surpass human-level performance by magnitudes.

This skill is a new way of coding: data science.

The CEO of Siemens, Joe Kaeser, believes that data is the new powerful asset class of the 21st century:

> *"Data is the oil, some say the gold, of the 21st century—the raw material that our economies, societies and democracies are increasingly being built on."* [1]

If data is the new asset class of the information society, data scientists are the new investment bankers.

What is data science?

> *"Data science is an interdisciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from data in various forms, both structured and unstructured."* [2]

Data scientists use computers to gain insights from massive amounts of data. These insights have a profound influence on the products we see, the medicines we take, the education we enjoy, the movies we watch, the routes

[1] `https://www.linkedin.com/pulse/technology-society-digital-transformation-joe-kaeser/`

[2] `https://en.wikipedia.org/wiki/Data_science`

we drive, the holidays we choose, and the foods we produce and consume.

A powerful tool in the tool belts of data scientists is machine learning. Machine learning, at its core, is the discipline of teaching machines to detect patterns and perform tasks by presenting them training data. In general, the more training data, the better machines perform in various tasks such as medical analysis, financial analysis, bio technology, research, games, fraud detection—with hundreds of new applications getting published every day in diverse research disciplines such as biology, mathematics, finance, engineering, history, and law theory. More training data leads to a higher degree of automation which, in turn, generates better results much quicker and cheaper than ever before.

So data is indeed the new asset class of the 21st century. As of 2019, six of the ten world's largest companies (by market capitalization) are tech companies (Microsoft, Google, Amazon, Apple, Facebook, Alibaba). Each of those companies expends significant effort to acquire a growing chunk of this valuable asset class. For example, those data sets consist of GPS location trajectories, customer behaviors, social network activities, web surfing behavior, health indicators, search interests—just to name a few.

It's a modern-day gold rush, and this book gives you the shovel to participate. This book aims to be a stepping stone on your path to becoming a master data scientist. It helps you learn faster by making use of proven principles of good teaching. It offers you ten to twenty hours of thorough training using a fun and effective training technique, called *practice testing*. Practice testing is scientifically proven to be one of the most efficient training techniques (see Chapter 2.8). More than 60,000 online students have successfully applied this learning system at my Python online learning platform `Finxter.com`. And practice testing will work for you, too.

This book focuses on one of the most popular programming languages for data science and machine learning: Python. A recent StackOverflow article *The Incredible Growth of Python*[3] shows that Python is one of the fastest growing major programming languages. But this book is not a general Python introduction like its predecessor *Coffee Break Python*[4]. Instead, this book teaches you the ins and outs of the NumPy library, which is used for numerical computations, for data science, and—more and more—for machine learning. Let me be blunt: without understanding the concepts and ideas behind NumPy, you will not become a successful data scientist.

[3]`https://stackoverflow.blog/2017/09/06/incredible-growth-python/`

[4]`https://blog.finxter.com/coffee-break-python`

— 2 —

A Case for Puzzle-based Learning

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.

Like the other books in the *Coffee Break Python* series, this book is based on the popular concept of puzzle-based learning to code—tried and tested by tens of thousands of online students and proven by educational science to be superior to most other learning techniques.

But before diving into practical puzzle solving, let us first study 10 reasons for puzzle-based learning—and why it helps you to learn NumPy faster and keep the basics longer. **Feel free to skip this bonus chapter if you**

already know about the benefits of puzzle-based learning from previous learning material. As you will see in this chapter, 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 are mostly focused on one-way teaching: the teacher speaks and you have to listen. This book attempts to change that. In brief, the 10 reasons for puzzle-based learning are the following.

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 more than 2400 year old teaching technique 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 a tension in the learner's mind. To close this gap, the learner awaits 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 looking at the code, you first 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. Look at this riddle: "What pulls you down and never lets go?" Can you feel the tension? Opening and closing a knowledge gap is a very powerful method for effective learning.[1]

Bad teachers open a knowledge gap that is too large. The learner feels frustrated because they cannot overcome the

[1]The answer is *Gravity.*

gap. Suppose you are standing before a river that you must cross. But you have not learned to swim yet. Now, consider two rivers. The first is the Colorado River that carved out the Grand Canyon—quite a gap. The second is Rattlesnake Creek. The fact that you have never heard of this river indicates that it is not too big of an obstacle. Most likely, you will not even attempt to swim through the big Colorado River. But you could swim over the Rattlesnake if you stretch your abilities just a little bit. You will focus, pep-talk yourself, and overcome the obstacle. As a result, your swimming skills and your confidence will grow a little bit.

Puzzles are like the Rattlesnake—they are not too great a challenge. You must stretch yourself to solve them, but you can do it, if you go all-out.

Constantly feeling a small but non-trivial knowledge gap creates a healthy learning environment. Stretch your limits, overcome the knowledge gap, and become better—one puzzle at a time.

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, but this addiction makes you smarter. Solving a puzzle gives your brain instant gratification. Easy puzzles lead to small, hard puzzles, which lead to large knowledge gaps. Overcome any of them and 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 the complexity. They seek a clear path to mastery.

People tend to prioritize specific activities with clearly defined goals. If the path is not clear, we tend to drift away toward more specific paths. Most aspiring coders think they have a goal: becoming a better coder. Yet, this is not a specific goal at all. So what is a specific goal? *Watching Breaking Bad after dinner, Series 2 Episode 1* is as specific as it can be. Due to the specificity, watching Netflix is more powerful than the fuzzy path of learning to code. Hence, watching Netflix wins most of the time.

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. You want to become a master coder? Break the big coding skill into a list of sub-skills—understanding language features, designing algorithms, reading code—and then tackle each sub-skill one at a time.

But how can you do this if you don't know anything about the topic yet? You cannot really comprehend the important subtopics of the skill to be acquired—without a mentor who has already been there and done that, your learning speed will be slow. You don't have time to waste, do you?

Fortunately, code puzzles do this for you. They break up the huge task of learning to code into a series of smaller learning units. The student experiences laser focus on one learning task such as *matrix multiplication*, the *standard deviation*, or *slicing*. Don't worry if you do not understand these concepts yet—after working through this book, you will.

A good code puzzle delivers a single idea from the author's into the student's head. You can digest one puzzle at a time. Each puzzle is a step toward your bigger goal of mastering data science. Keep solving puzzles and you keep improving your skills.

2.4 Improve From Immediate Feedback

The right feedback is critical for your success. As a child, you learned to walk by trial and error—try, get feedback, adapt, and repeat. Unconsciously, you minimize negative and maximize positive feedback. You avoid falling because it hurts. You seek the approval of your parents. Feedback supervised your learning progress along the way.

But not only organic life benefits from the great learning technique of trial and error. In machine learning, algorithms learn by guessing an output and adapting their guesses based on their correctness. To learn anything, you need feedback such that you can adapt your actions.

However, an excellent learning environment provides you not only with feedback but with *immediate* feedback for your actions.

In contrast, poor learning environments do not provide any feedback at all or only with a large delay. Examples are activities with good short-term and bad long-term effects such as smoking, alcohol, or damaging the environment. People cannot control these activities because of the delayed feedback. If you were to slap your friend each time he lights a cigarette—a not overly drastic measure to safe his life—he would quickly stop smoking. If you want to learn fast, make sure that your environment provides immediate feedback. Your brain will find rules and patterns to maximize the reinforcement from the immediate feedback.

This book offers you an environment with immediate feedback to make learning to code NumPy easy and fast. 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. There are other environments with immediate feedback, like executing code and checking correctness, but puzzle-based learning is the most direct one: Each puzzle educates with immediate feedback.

2.5 Measure Your Skills

You need to have a definite goal to be successful. A definite goal is a powerful motivator and pushes you to stretch your skills constantly. The more definite and concrete it is, the stronger it becomes. Holding a definite goal in your mind is the first and foremost step toward its physical manifestation. Your beliefs bring your goal into reality.

Think about an experienced Python programmer you know, e.g., your nerdy colleague or class mate. 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. Not being able to measure your coding skills diverts your focus from systematic improvement. Your goal becomes less definite.

So what should be your definite goal when learning a programming language? To answer this, let us travel briefly 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 greater increase of 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 grand master titles. They become respected among chess players and in the outside world.

Every chess player dreams of being a grandmaster. The goal is as definite as it can be: reaching an Elo of 2400 and master level (see Section 3). 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 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 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. But only learners with high Elo ratings will see them. 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.

2.6 Individualized Learning

The educational system today 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. This traditional idea of classes and courses has a strong foundation in our culture and social thinking patterns. Yet, science proves again and again the value of individualized learning. Individualized learning tailors the content, pace, style, and technology of teaching to the student's skills and interests. Of course, truly individualized learning has always required a lot of teachers. But paying a high number of teachers is expensive (at least in the short term) in a non-digital environment.

In the digital era, many fundamental limitations of our society begin to crack. Compute servers and intelligent machines can provide individualized learning with ease. But with changing limitations, we must adapt our thinking as well. Machines will enable truly individualized learning very soon; yet society needs time to adapt to this trend.

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. The Finxter learning system uses a simple but effective solution to solve this problem: the Elo rating system. The student solves puzzles at their individual skill level. This book with it's web backend Finxter pushes teaching toward individualized learning.

2.7 Small is Beautiful

The 21st century has seen a rise in microcontent. Microcontent is a short and accessible piece of valuable 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 is powerful because it satisfies the desire for shallow entertainment. Microcontent has many benefits: the consumer stays engaged and interested, and it is easily digestible in a short time. Each piece of microcontent pushes your knowledge horizon a bit further. Today, millions of people are addicted to microcontent.

However, this addiction will also become a problem to these millions. 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. Contrarily, shallow work is every low-value activity that can be done by everybody (e.g., posting cat videos to social media). The demand for deep work grew with the rise of the information society; at the same time, the supply stayed constant or decreased, among other things because of 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 demanded resources rise. Because of this, surgeons, lawyers, and software developers earn $100,000 per year and more. Their work cannot easily be replaced or outsourced to unskilled workers. If you are able to do deep work, to focus your attention on a challenging problem, society pays you generously.

What if we could marry the concepts of microcontent and 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 microcontent, 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 bold goal, i.e., *reach the mastery level in Python's NumPy library*, into tiny actionable steps: solve and understand one code puzzle per day. While solving the smaller tasks, you progress toward your larger goal. You take one step at a time to eventually reach the mastery level. 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 topic, test scores of active learners improved 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.

How does active learning work? Active learning requires the student to interact with the material, 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.*

However, the study argues that students must feel safe during these tests. Therefore, the tests must be low-

---

[2] `https://en.wikipedia.org/wiki/Active_learning#Research_evidence`

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

stake, i.e., 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 10. 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. Yet, I always found this impracticable while reading on the train, on the bus, or in bed. But if these active elements drop out, learning becomes 100% passive again.

Fixing this mismatch between research and common practice drove me to write my *Coffee Break Python* book series about puzzle-based learning of Python and Python's libraries. 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 develop the solution yourself, in 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.

Let me emphasize this argument again: puzzle-based learning is a variant of the active learning technique named practice testing. Practice testing is scientifically proven to teach you more in less time.

2.9 Make Code a First-class Citizen

Each grandmaster of chess has spent tens of thousands of hours looking into myriad chess positions. Over time, they develop a powerful skill: the intuition of the expert. When presented with a new position, they are able to 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 grand masters view a whole position or a sequence of moves as a single chunk of information. The extensive training and experience has burned strong patterns into their biological neural networks. Their brain is able to hold much more information—a result of the good learning environment they have put themselves in.

What are some principles of good learning? Let us dive into another example of a great learning environment—this time for machines. Google's artificial intelligence AlphaZero has proven to be the best chess playing entity in the world. AlphaZero uses artificial neural networks. An artificial neural network is the digital twin of the human brain with artificial neurons and synapses. It learns by example much like a grandmaster of chess. It presents itself a position, predicts a move, and adapts its prediction to the extent the prediction was incorrect.

Chess and machine learning exemplify 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 reading mathematical papers with theorems and proofs a habit. 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. 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 isolated stimuli again and again until they have mastered them. Then they move on to more complex stimuli.

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

2.10 What You See is All There is

My professor of theoretical computer science used to tell us that if we only stare long enough at a proof, the meaning will transfer into 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 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.

My gold standard was to design each puzzle such that it is mostly self-contained. However, to deliver on the book's promise of training your understanding of the Python basics, puzzles must introduce syntactical language elements as well. But even if the syntax in a puzzle challenges you, you should still 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—and NumPy

Pick any sport you always loved to do. 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.

Let us give a small example of how the Elo rating works in chess. Alice is a strong player with an Elo rating of 2000 while Bob is an intermediate player with Elo 1500. Say Alice and Bob play a chess game against each other. Who will win the game? As Alice is the stronger player, she should win the game. The Elo rating system rewards players for good and punishes for bad results: the better the result, the higher the reward. For Bob, a win, or even a draw, would be a very good outcome of the game.

For Alice, the only satisfying result is a win. Winning against a weaker player is less rewarding than winning against a stronger player. Thus, the Elo rating system rewards Alice with only +3 Elo points for a win. A loss costs her -37 Elo points, and even a draw costs her -17 points. Playing against a weaker player is risky for her because she has much to lose but little to win.

The idea of Finxter is to view your learning as a series of games between two players: you and the Python puzzle. Both players have an Elo rating. Your rating measures your current skills and the puzzle's rating reflects its difficulty. On our website `finxter.com`, a puzzle plays against hundreds of Finxter users. Over time, the puzzle's Elo rating converges to its true difficulty level—while your Elo rating converges to your true skill level. A compelling idea, isn't it?

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 describe how you can use this book to test your Python skills.

3.1 How to Use This Book

This book provides a series of 48 code puzzles plus explanations to test and train your NumPy skills. The puzzles start from an intermediate level and become gradu-

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.

ally harder to reach advanced level. This book is perfect for users who have already reached intermediate Python coding level. Yet, even expert users can improve their speed of code understanding. No matter your current skill level, you will benefit from puzzle-based learning. It will deepen and accelerate your understanding of basic coding patterns. But even more importantly, you will take the first step towards your thorough data science education.

3.2 The Ideal Code Puzzle

The ideal code puzzle possesses each of the following six properties. The puzzle

1. has a surprising result;
2. provides new information;
3. is relevant and practical;
4. delivers one main idea;
5. can be solved by thinking alone; and
6. is challenging but not overwhelming.

This was the gold standard for all the puzzles created in this book. I did my best to adhere to this standard.

3.3 How to Exploit the Power of Habits?

You are what you repeatedly do. Your habits determine your success in life and in any specific area such as coding. Creating a powerful learning habit can take you a long way on your journey to becoming a code master. Charles Duhigg, a leading expert in the psychology of habits, shows that each habit follows a simple process called the *habit loop*. This process consists of three steps: trigger, routine, and reward.[1] First, the trigger starts the process. A trigger can be anything such as drinking your morning coffee. Second, the routine is an action you take when presented with the trigger. An example routine is to solve a code puzzle. Each routine is in anticipation of a reward. Third, the reward is anything that makes you feel good. When you overcome a knowledge gap, your brain releases endorphins—a powerful reward. Over time, your habit becomes stronger—you seek the reward.

Habits with strong manifestations in these three steps are life-changing. Invest 10% of your paycheck every month and you will be rich one day. Get used to the habit of solving one Python (or NumPy) puzzle a day as you drink your morning coffee—and enjoy the endorphin dose in your brain. Implementing this *Finxter loop* in your day

[1] Charles Duhigg, *The Power of Habit: Why We Do What We Do in Life and Business.*

sets up an automatic progress toward you becoming a better and better coder. As soon as you have established the Finxter loop as a strong habit, it will cost you neither a lot of time, nor energy. This is self-engineering at its finest level.

3.4 How to Test and Train Your Skills?

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

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

1. Track your individual Elo rating as you read the book and solve the code puzzles. Simply write your current Elo rating into the book. Start with an initial rating of 1500 if you are a Python intermediate who is just starting out with NumPy. Otherwise, adapt this initial rating towards your estimated skill level in Python. Of course, if you already have an online rating on `finxter.com`, starting with this

rating would be the most precise option. Figure 3.4 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 initial rating and how well you perform (the latter being more important).

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

Solve the puzzles in a sequential manner because they build upon each other. Advanced readers can also solve puzzles in the sequence they wish—the Elo rating will still work. The Elo rating will become more accurate as you solve more and more puzzles. 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 surprisingly accurate.

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

1. Select a daily trigger after which you solve code puzzles for 10 minutes. For example, decide on your *Coffee Break NumPy*, or even solve code puzzles as you brush your teeth or sit on the train to work, university, or school.

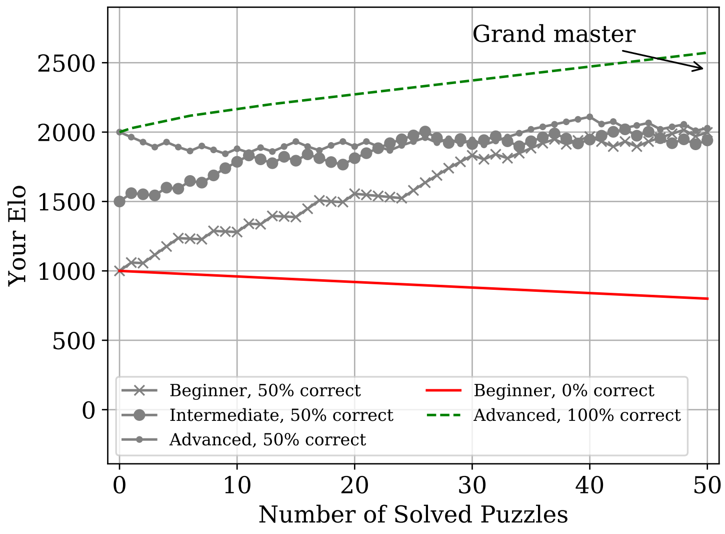


Figure 3.1: This plot exemplifies how your Elo rating may change while you work through the 50 code puzzles. No matter how you select your initial Elo, it will converge on your true skill level as you solve more puzzles. Note that you will lose Elo points faster when you have a higher Elo number. Your final Elo will be anywhere between 900 and 2500 after working through this book.

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. Make sure you carry a pen with you and write your solution into the book. This ensures that you stay objective—we all have the tendency to fake ourselves. Active learning is a central idea of this book.

5. Look up the solution and read the explanation with care. Do you understand every aspect of the code? Write open questions down and look them up later, or send them to me (`info@finxter.com`). I will do everything I can to come up with a good explanation.

6. Only if your solution was 100% correct—including whitespaces, data types, and formatting of the output—do 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 understand 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 `Finxter.com`, which has more than 300 handcrafted code puzzles. I regularly publish new code puzzles on the website as well.

3.5 What Can This Book Do For You?

Before we dive into puzzle solving, let me anticipate and address possible misconceptions about this book.

The puzzles are too easy/too hard. This book is for you if you already have some experience in coding. Your skill level in the Python programming language ranges from intermediate to expert. Even so, if you are already an advanced coder, this book is for you as well—if you read it in a different way. Measure the time you need to solve the puzzles and limit your solution time to only 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.

Learning to code is best done via coding on projects. This is only part of the truth. Yes, you can improve your skills to a certain level by diving into practical projects. But as in every other discipline, your skills will quickly hit your personal ceiling. Your ceiling is the maximum skill level you are able to reach, given your current limitations. These limitations come from a lack of thorough understanding of basic knowledge. You cannot understand higher-level knowledge properly without 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 axe."* 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 axe 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 enjoyed an academic education. They know that solving hundreds or thousands of toy examples in their studies built a strong and thorough foundation.

How am I supposed to solve this puzzle if I do not know the meaning of this specific NumPy function? Guess it! Python is an intuitive language, and NumPy has very intuitive naming of its functions. 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.

Why should I buy the book when puzzles are available for free at `Finxter.com`? My goal is to remove barriers to learning Python. Thus, all puzzles are available for free online. This book is based on the 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 with valuable information. Over time, they have a huge advantage over their peers.

They get the opportunities to develop themselves even further. They enjoy their jobs and have much higher work satisfaction and life quality. Belonging to the top ten percent in your field yields hundreds of thousands of dollars during your career. However, there is a price you have to pay to unlock the gates to this world: you have to invest in books and your own 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. Several finxters find the rating helpful, fair, and accurate in comparison to others. It provides a good indication of where one stands in the field of 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 have a chance to improve this book and the Finxter back-end.

— 4 —

A Quick Data Science Tutorial: The NumPy Library

This tutorial gives you a simple introduction, with many practical examples, to Python's NumPy library. You don't need any prerequisites to follow the tutorial. The idea of the tutorial is to give you everything you need to know to successfully solve the puzzles in the later parts of the book.

By working through the tutorial, you will gain a basic understanding of the most important NumPy functionality. Moreover, it will give you references to further reading as well as "next steps." Reading this tutorial takes 20–30 minutes and will be a fertile investment in your education and your coding efficiency. It's our belief that the purpose of any good learning material is to ultimately save your time.

So without further introduction, let's dive into the NumPy library in Python.

4.1 What is NumPy?

NumPy is a Python library that allows you to perform numerical calculations. Think about linear algebra in school or university—NumPy is the Python library for it. It's about matrices and vectors—and performing mathematical operations on them.

At the heart of NumPy is a basic data type, called the *NumPy array*. A NumPy array may have a number of dimensions, thus allowing it to represent quantities such as vectors (1D), matrices (2D), or higher dimensional arrays such as tensors. A NumPy array allows only one data type for all its elements. In this sense, NumPy arrays are different from Python lists that allow arbitrary data types. Therefore we say that NumPy requires homogeneous data values, so a NumPy array contains either integer or float values, but not both at the same time. These data type restrictions allow NumPy to specialize in providing efficient linear algebra operations.

Among those operations are maximum, minimum, average, standard deviation, variance, dot product, matrix product, and many more. NumPy implements these operations efficiently and in an easy-to-use manner. By learning NumPy, you equip yourself with a powerful tool for data analysis on numerical multi-dimensional data. But you may ask (and rightly so):

4.2 What can NumPy do for me?

Fear of missing out in machine learning and data science? Learning NumPy now is a great first step into the field of machine learning and data science. In machine learning, crucial algorithms and data structures rely on matrix computations, which are efficiently handled by NumPy. As said earlier, matrices in NumPy are nothing but arrays with homogenous data, for example, float values.

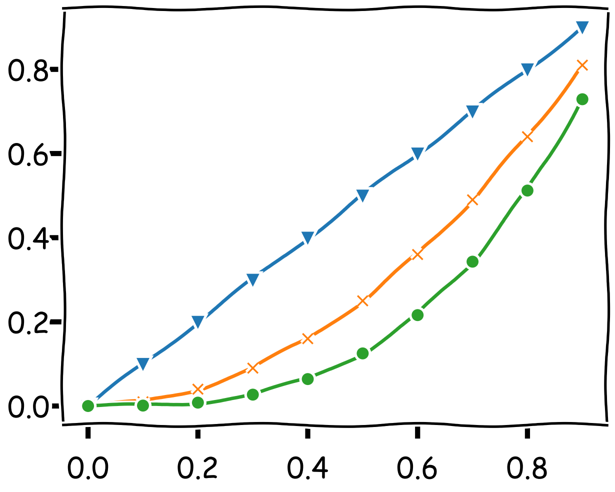
NumPy is among the most popular libraries in Python. Most machine learning experts agree that Python is the top programming language for machine learning. Within Python, NumPy is one of the most important libraries for data science and machine learning. For instance, searching for the keyword 'NumPy machine learning' reveals more than 3 million pages. Compare this to Python's scikit-learn library that directly addresses machine learning and results in approximately 3 million pages as well. So NumPy is as popular for machine learning as scikit-learn in Python! As you can see, NumPy produces as many results—even though it is not directly addressing machine learning (unlike scikit-learn).

No matter which library is more popular, NumPy is the 600-pound Gorilla in the machine learning and data science space. If you are serious about your career as a data scientist, you have to conquer NumPy now!

But NumPy is not only important for machine learning and data science. NumPy's diverse functionality and its computational efficiency leads to its use in various fields such as mathematics, electrical engineering, high-performance computing, and simulations.

Also, if you need to visualize data, you are very reliant on the NumPy library. Here is an example from the official documentation of Python's plotting library matplotlib. You can see a small script that plots a linear, quadratic, and cubic function. It uses only two libraries: matplotlib and, obviously, NumPy!

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

# evenly distributed data between 0 and 1
x = np.arange(0., 1., 0.1)

# xkcd-styled plot
plt.xkcd()

# linear, quadratic, and cubic plots
plt.plot(x, x, 'v-', x, x**2, 'x-', x, x**3, 'o-')
plt.savefig("functions.pdf")
plt.show()
```



Wherever you go in Python, NumPy is already there!

4.3 What are the Limitations of NumPy?

The focus of NumPy is working with numerical data. It's both powerful and low-level, thereby providing basic functionality for high-level algorithms. If you enter the machine learning and data science space, you want to master NumPy first. But eventually, you will use other libraries that operate on a higher level such as TensorFlow, Pandas, and scikit-learn. Those libraries contain out-of-the-box machine learning functions such as training and inference algorithms. Have a look at them after reading this tutorial.

Nevertheless, NumPy's handy functionality will definitely help you to use any off-the-shelf machine learning algorithms effectively. For example, it will typically allow you to pre-process your datasets and to post-process the predictions made by your trained machine-learning algorithms. If you do not understand the last sentence, don't worry. Simply stated, you will gradually find out that NumPy is often used alongside popular machine-learning and data-science libraries such as scikit-learn, TensorFlow, and Keras.

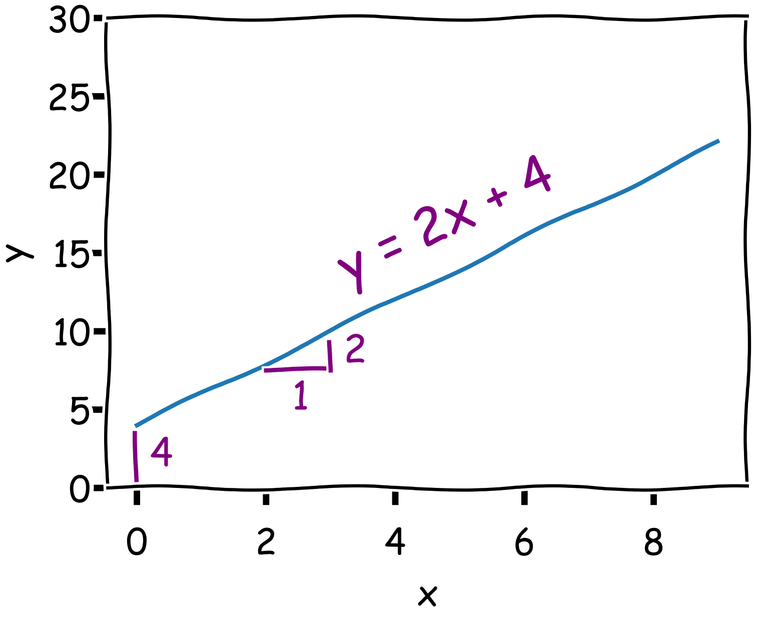
4.4 What are the Linear Algebra Basics You Need to Know?

NumPy is all about manipulating arrays. By learning NumPy, you will also learn and refresh your basic linear algebra skills from school. We will also repeat many concepts of linear algebra in this book. It's always better to learn the concepts first and the tools later. NumPy is only a specific tool that implements these concepts of linear algebra.

At the center of linear algebra stands the solution of linear equations. Here is one of those equations:

$$y = 2x + 4$$

If you plot this equation, you get the following output:



As you can see, the equation $y = 2x+4$ leads to a straight line on the space. This line defines a relationship between the values on the x-axis and the y-axis. Particularly, it allows the value of y to be determined for any given value of x.

Let me repeat this: You can determine the value of y for any given value of the input x.

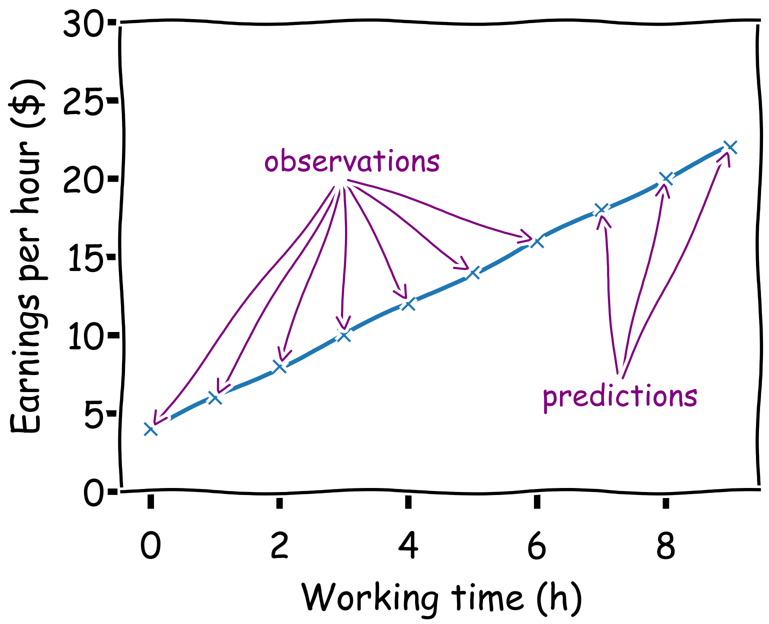
As it turns out, this is the very goal of any machine learning technique. From a bunch of data values belonging to certain variables (e.g, x and y), you want to find a function that describes the relationship between these variables. In machine learning this is called the learning phase. Subsequently you can use the learned function to "predict" the output value for any new input value. It works, even if you have never seen this input before. This second phase is called the inference phase.

Linear algebra helps you solve equations to do exactly that.

Here is an example with some fake data. Say, you have learned the relationship between the work ethics in number of hours worked per day and hourly wage in US dollars. Your learned relationship, also called a "model," is the equation $y = 2x + 4$. The input x is the number of hours worked per day and the output y is the hourly wage.

With this model, you can predict how much your boss earns based on the number of hours he or she invests in work. It works just like a machine: you put in x and get out y. This is what machine learning is all about:

learning a model representing the relationship between different variables using data from past observations and, later, using this model for making precise predictions.



Here is the script that does this plot for us. For simplicity, we do not include the code that labels the data points as observations and predictions.

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

x = np.arange(0., 10., 1)
```

```
# [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]

y = 2 * x + 4
# [ 4.  6.  8. 10. 12. 14. 16. 18. 20. 22.]

print(x)
print(y)

# xkcd-styled plot
plt.xkcd()

plt.plot(x, y, 'x-')
plt.xlabel("Working time (h)")
plt.ylabel("Earnings per hour ($)")
plt.ylim((0,30))

plt.tight_layout()
plt.savefig("simple_linear_equation_example.pdf")
plt.show()
```

As you can see, before doing anything else in the script, we have to import the NumPy library. Use the statement `import NumPy as np` to do so. Each time you want to call a NumPy function, you have to use the short-name 'np' (e.g., `np.average(x)`). In theory, you can specify any other short-name, but it is better not to do this. The name 'np' has crystallized as a convention for the

NumPy library, so every experienced coder will expect adherence to this convention.

After the initial import, we create a series of floating point values between 0 and 9. These values serve as the x values that we want to map to their respective function values $y = f(x)$. The variable x holds a NumPy array of those floating point values.

The variable y holds a NumPy array of the same size. It's our output—one for each observed x value. Do you see the basic arithmetic of how to get the y values?

The equation $y = 2x + 4$ seems to do the same thing as discussed in the previous equation. But as it turns out, the meaning is very different: x is not a numerical value, it is a NumPy array!

When calculating $y = 2x + 4$, we are basically multiplying the NumPy array by 2 and adding the constant 4 to it. These are basic mathematical operations on multi-dimensional NumPy arrays, not just single numerical values.

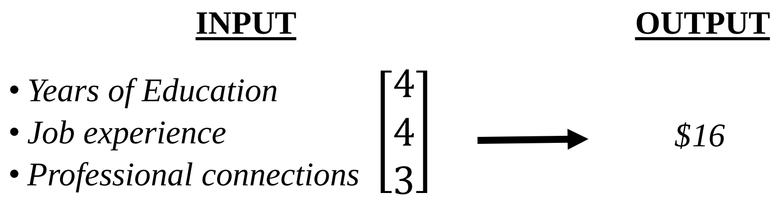
Investigating these kinds of operations lies at the core of linear algebra. The NumPy array in the example is called a one-dimensional matrix or a vector of scalar values. The matrix x consists of 10 floating point values between 0 and 9 (inclusive): [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]. How do we know that the values in the NumPy array are of type float? We indicate this by writing a dot "." after each array value. It's simply a short form of [0.0 1.0 2.0 ... 9.0].

The linear algebra magic of NumPy calculates the respective array of y values. Finally, we plot the result using the library matplotlib.

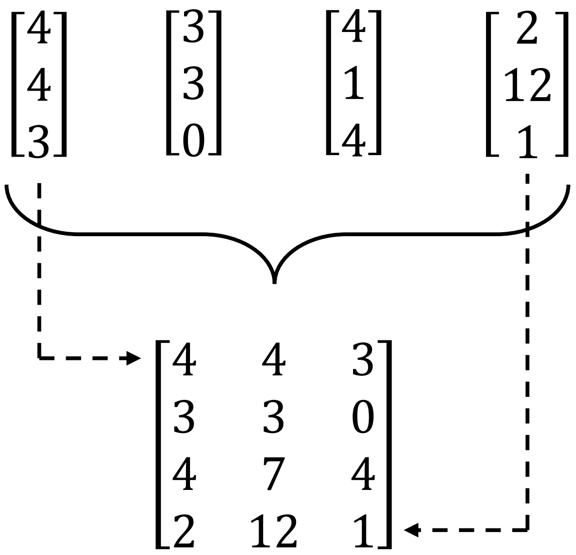
In the two-dimensional space shown in the plot, we work with one-dimensional NumPy arrays (vectors). Each numerical input value leads to an output value. One observation leads to one prediction. For example "worked 4 hours per day" therefore the model predicts "earned $12 per hour." But real problems are far more complex than that.

Think about it: we have to consider a multitude of other factors to accurately predict the hourly wage of a person. For example, their education measured in number of years studied, their experience in number of years worked in the job, how many professional connections they have, and so on.

In this case, each observation (input) is not a single factor as in the last plot but a collection of factors. We express a single input value as a vector (one-dimensional array) to account for the multiple relevant observations. Together, this vector of observations leads to a single output. Here is an example:


INPUT
OUTPUT
• Years of Education
• Job experience
• Professional connections
$\begin{bmatrix}4\\4\\3\end{bmatrix}$
$16

In the first plotting script, we assembled up all the scalar observations into a one-dimensional matrix. In a similar fashion, we can now assemble all the one-dimensional observation-vectors into a two-dimensional matrix. The following graphic shows how to do this.

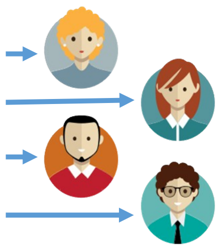

$\begin{bmatrix}4\\4\\3\end{bmatrix}$
$\begin{bmatrix}3\\3\\0\end{bmatrix}$
$\begin{bmatrix}4\\1\\4\end{bmatrix}$
$\begin{bmatrix}2\\12\\1\end{bmatrix}$
$\begin{bmatrix}4 & 4 & 3\\3 & 3 & 0\\4 & 7 & 4\\2 & 12 & 1\end{bmatrix}$

There are four observations in this graphic, each having three factors per observation: [4, 4, 3], [3, 3, 0], [4, 1, 4], [2, 12, 1]—each being a one-dimensional matrix. We collect those observations in a two-dimensional observation matrix. Every single row of this matrix consists of one observation. Each column consists of all observations for a single factor. For example, the first row [4, 4, 3] stands for the first observation: [experience = 4, education = 4, family = 3]. The first column [4, 3, 4, 2] stands for all the observed values of the factor "experience."

Now recall our goal: we want to calculate the y value (=hourly wage) based on the observed factors "x1 = experience," "x2 = education," and "x3 = responsibility." So let's assume that a machine learning algorithm tells us that the hourly wage can be calculated by summing up those factors: $y = x1 + x2 + x3$. For example, the first observation leads to $y = x1 + x2 + x3 = 4 + 4 + 3 = 11$. In other words: if you have four years of experience, four years of education, and are responsible for three coworkers, you will earn $11 per hour.

Now, instead of using numerical values, we can also use the factor vectors, i.e., the columns, as variables $x1$, $x2$, and $x3$—and the equation still works. So instead of setting $x1 = 4$, $x2 = 4$, and $x3 = 3$, you can set $x1 = [4, 3, 4, 2]$, $x2 = [4, 3, 1, 12]$, and $x3 = [3, 0, 4, 1]$. Why should you do that? Because it allows you to calculate the predictions of ALL observations in a single step.

$$\begin{bmatrix}4\\3\\4\\2\end{bmatrix}+\begin{bmatrix}4\\3\\1\\12\end{bmatrix}+\begin{bmatrix}3\\0\\4\\1\end{bmatrix}=\begin{bmatrix}11\\6\\9\\15\end{bmatrix}$$



In each row, we calculate the prediction of one person. Each of the sum operands is a one-dimensional array (vector). As we are calculating the sum of the vectors (rather than the sum of the numerical values), we get a resulting vector [11, 6, 9, 15] that holds the predicted hourly wages of each of the four persons.

At this point, you have already learned how and why to add vectors. It allows the computer to crunch large amounts of data (and predict hourly wages for a large number of persons).

4.5 What are Arrays and Matrices in NumPy?

Are you confused about the terms matrices, arrays, vectors? Don't despair. In NumPy, there is only one data structure: NumPy arrays. A NumPy array can be one-dimensional, two-dimensional, or 1000-dimensional. It's one concept to rule them all.

The NumPy array is the core object of the whole NumPy library. You have to know it by heart before you can go on and understand the operations provided by the NumPy library. So what is the NumPy array?

It's a data structure that stores a bunch of numerical values (well, you could store strings as well). In particular, these are the most important numerical data types of a NumPy array:

Type	**Description**	**Bytes**
bool	Default boolean data type	1
int	Default integer data type	4 or 8
float	Default float data type	8
complex	Default complex data type	16
np.int8	Integer data type	1
np.int16	Integer data type	2
np.int32	Integer data type	4
np.int64	Integer data type	8
np.float16	Float data type	2
np.float32	Float data type	4
np.float64	Float data type	8

Here is an example that shows how to create NumPy arrays of different data types.

```
import numpy as np

a = np.array([1, 2, 3], dtype=np.int16)
print(a) # [1 2 3]
print(a.dtype) # int16

b = np.array([1, 2, 3], dtype=np.float64)
print(b) # [1. 2. 3.]
print(b.dtype) # float64
```

In this example, we created two NumPy arrays.

The first array a is of data type np.int16. If we print the array, we can already see that the numbers are of type integer since there is no "dot" after the number. Specifically, when printing out the dtype property of the array a, we get the result int16.

The second array b is of data type float64. So even if we pass a list of integers as a function argument, NumPy will convert the type to np.float64.

You should remember two things from this example: 1) NumPy gives you control over the data, and 2) the data in a NumPy array is homogeneous (= of the same type).

4.6 What are Axes and the Shape of an Array?

The second restriction of NumPy arrays is the following. NumPy does not simply store a bunch of arbitrarily-typed data values—for that purpose you can use lists. Instead, NumPy imposes a strict homogeneous data type. Additionally, NumPy, in contrast to lists, organizes data in fix-sized axes. Axes represent the different dimensions of NumPy arrays. You may or may not represent your high dimensional data, say n-dimensional, with n axes. For example, it is possible to represent a 3-dimensional using a one-dimensional or single-axis array, e.g., [1, 2, 3]. If you want to know the number of axes of a NumPy array, count the number of opening brackets "[" until reaching the first numerical value. Here is an example:

```
import numpy as np

a = np.array([1, 2, 3])
print(a.ndim)
# 1

b = np.array([[1, 2], [2, 3], [3, 4]])
print(b.ndim)
# 2
```

```
c = np.array([[[1, 2], [2, 3], [3, 4]],
[[1, 2], [2, 3], [3, 4]]])
print(c.ndim)
# 3
```

We create three NumPy arrays a, b, and c and print the number of axes. How do we know this? NumPy stores the number of axes in the array property ndim. As you can see, manually counting the number of square brackets gives you the correct number of axes of the NumPy array.

But there is another important piece of information you will often need to know about a NumPy array: the shape. The shape returns not only the number of axes but also the number of elements in each axis, that is the size.

Here is a code example:

```
import numpy as np

a = np.array([1, 2, 3])
print(a.shape)
# (3, )

b = np.array([[1, 2], [2, 3], [3, 4]])
print(b.shape)
# (3, 2)

c = np.array([[[1, 2], [2, 3], [3, 4]],
[[1, 2], [2, 3], [3, 4]]])
print(c.shape)
# (2, 3, 2)
```

Study this example carefully. The shape property returns three types of information about each array.

First, it shows the number of axes per array—that is, the length of the shape tuple. NumPy array a has one axis, NumPy array b has two axes, and NumPy array c has three axes.

Second, it indicates the length of each axis as a numerical value. For example, array a has one axis with three elements. Hence, the shape of the array is (3,). Don't get confused by this weird tuple notation. The reason why the NumPy shape operation does not return a tuple with a single element (3) is: Python converts it to a numerical value 3, which is not iterable anymore. This has the following benefit: if you access the first element of your shape object `a.shape[0]`, the interpreter does not throw an exception.

Third, it shows the ordering of the axes. Consider NumPy array c. It has three tuple values (2, 3, 2). Which tuple value is for which axis?

- The first tuple value is the number of elements in the first level of nested arrays. In other words: how many elements are in the outermost array? The outermost array for c is [X1, X2] where X1 and X2 are nested arrays themselves. Hence, the first axis consists of two elements.

- But what is the number of elements for the second axis? Let's check the axis X1. It has the shape X1 = [Y1, Y2, Y3] where Y1, Y2, and Y3 are arrays themselves. As there are three such elements, the result is 3 for the second tuple value.
- Finally check the innermost axis Y1. It consists of two elements [1, 2], so there are two elements for the third axis.

In summary, the axes are ordered from the outermost to the innermost nesting level. The number of axes is stored in the ndim property. The shape property represents the number of elements in each axis.

4.7 How to Create and Initialize NumPy Arrays?

There are many ways to create and initialize NumPy arrays. You have already seen some of them in the previous examples. But the easiest way to create a NumPy array is via the function `np.array(s)`. Simply put in a sequence `s` of homogeneous numerical values and voilà—there is your NumPy array.

In the last code example, we created three arrays a, b, and c. The sequence argument for array a is a list of integer values. But we could also use a tuple to initialize the same NumPy array, e.g., `np.array((1, 2, 3))`. Both produce the same NumPy array of integer values. The sequence argument for array c is a list of floats. As you can see, the result is a NumPy array of float values.

Puzzle: What is the output of this code snippet?

```
import numpy as np


# 2D numpy array
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a.shape)

# 3D numpy array
b = np.array([[[1, 2], [3, 4], [5, 6]],
[[1, 2], [3, 4], [5, 6]]])
print(b.shape)
```

Answer: The puzzle prints two shape objects. The shape of array a is (2, 3) because the first axis has two elements and the second axis has three elements. The shape of array b is (2, 3, 2) because the first axis has two elements (sequences of sequences), the second axis has three elements (sequences), and the third axis has two elements (integers).

Note: Having at least one floating type element, the whole NumPy array is converted to a floating type array. Recall that NumPy arrays have homogeneously typed data. Here is an example of such a situation:

```
import numpy as np


a = np.array([[1, 2, 3.0], [4, 5, 6]])
print(a)
# [[1. 2. 3.]
#  [4. 5. 6.]]
```

Now, let's move on to more automated ways to create NumPy arrays. For the examples given above, you can simply type in the whole array. But what if you want to create huge arrays with thousands of values?

For this purpose, use NumPy's array creation routines called `ones(shape)` and `zeros(shape)`. All you have to do is specify the shape tuple we discussed in the last paragraphs. Suppose you want a 5-dimensional array with 1000 values per dimension, initialized with 0.0 values. Using these routines, you would simply call: `np.zeros((1000, 1000, 1000, 1000, 1000))`.

As it turns out, this simple array creation routine overwhelms your computer's memory capacity. The Python interpreter throws an error when you try to create a NumPy array of this size. Why? Because you told it to create 1000 * 1000 * 1000 * 1000 * 1000 = 10**15 or 1000 trillion (!) integer numbers.

Anyway, here are examples of how to create NumPy arrays using the functions ones() and zeros().

```
a = np.zeros((10, 10, 10, 10, 10))
print(a.shape)
# (10, 10, 10, 10, 10)

b = np.zeros((2,3))
print(b)
# [[0. 0. 0.]
#  [0. 0. 0.]]

c = np.ones((3, 2, 2))
print(c)
# [[[1. 1.]
#   [1. 1.]]
#
# [[1. 1.]
#  [1. 1.]]
#
# [[1. 1.]
#  [1. 1.]]]

print(c.dtype)
# float64
```

Note that the data types are implicitly converted to floats. Floating point numbers are the default NumPy array data type (on my computer: the np.float64 type). But what if you want to create a NumPy array of integer values? Just specify the data type of the NumPy array as a second argument to the `ones()` or `zeros()` functions. Here is an example:

```
import numpy as np


a = np.zeros((2,3), dtype=np.int16)
print(a)
# [[0 0 0]
#  [0 0 0]]

print(a.dtype)
# int16
```

Finally, there is another way to create NumPy arrays that is also very common: the NumPy arange function.

Here is a quick summary of NumPy arange: The NumPy function `np.arange(start[, stop[, step])` creates a new NumPy array with evenly spaced numbers between start (inclusive) and stop (exclusive) with the given step size. For example, `np.arange(1, 6, 2)` creates the NumPy array [1, 3, 5]. The following detailed example shows you how to do this:

```
import numpy as np


a = np.arange(2, 10)
print(a)
# [2 3 4 5 6 7 8 9]

b = np.arange(2, 10, 2)
print(b)
# [2 4 6 8]

c = np.arange(2, 10, 2, dtype=np.float64)
print(c)
# [2. 4. 6. 8.]
```

It is also possible to specify the dtype argument as for any other array creation routine in NumPy.

But keep in mind the following. If you want to create an evenly spaced sequence of float values in a specific interval, don't use the NumPy arange function. The documentation discourages this because it's improper handling of boundaries. Instead, the official NumPy tutorial recommends using the NumPy `linspace()` function instead.

The NumPy linspace function works like the NumPy arange function. But there is one important difference: instead of defining the step size, you define the number of elements in the interval between the start and stop values. Check the following example:

```
import numpy as np


a = np.linspace(0.5, 9.5, 10)
print(a)
# [0.5 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5]

b = np.linspace(0.5, 9.5, 5)
print(b)
# [0.5  2.75 5.   7.25 9.5 ]
```

Now, you know everything you need to know about array creation to get started with NumPy. Play around with the Python interpreter to get acquainted with the code.

If you feel that you have mastered the array creation routines, go on to the next important topic in Python's

NumPy library.

How does indexing and slicing work in Python? Indexing and slicing in NumPy are very similar to indexing and slicing in Python. If you have mastered slicing in Python, understanding slicing in NumPy is easy. In the next paragraphs, you will get a short introduction into indexing in Python. After this, we will briefly explain slicing in Python. Having understood indexing and slicing in Python, you will then learn about indexing and slicing in NumPy.

Let's start with an example to explain indexing in Python. Suppose we have a string "universe". The indices are simply the positions of the characters of this string.

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

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

Now, let's dive into slicing in Python.

The idea of slicing is simple. You carve out a subsequence from a sequence by defining the start index and the end index. While indexing retrieves only a single character, slicing retrieves a whole substring within an index range.

For slicing, use the bracket notation with the start and end position identifiers. For example, `word[i:j]` returns the substring starting from index i (included) and ending at index j (excluded).

You can also skip the position identifier before or after the slicing colon. This indicates that the slice starts from the first or last position, respectively. For example, `word[:i] + word[i:]` returns the same string as word. Note here that the "+" sign concatenates the two substrings together.

Here is a slicing example.

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

The result is the string "iv". We start from the character on position 2 (the third character) and end the slice at position 4 (excluded from the slice).

For the sake of completeness, we shortly explain the advanced slicing notation [start:end:step]. The only difference to the previous notation is that you also specify the step size. For example, the command `‘‘python’’[:5:2]` returns every second character up to the fourth character, i.e., the string "pto". See the following example.

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

The result is the string “ies”. You start from the third character (included) and select every other character until you reach the end of the string.

Let’s dig a bit deeper into slicing to make sure that you are getting it 100%.

We have searched Quora to find all the little problems new Python coders are facing with slicing. Here are the answers to six common questions.

1) How to skip slicing indices (e.g., `s[::2]`)?

The Python interpreter assumes certain default values for `s[start:stop:step]`. They are: `start=0`, `stop=len(s)`, and `step=1` (in the slice notation: `s[::]==s[0:len(s):1]`). So writing `s[::2]` assumes that the missing start and the end indices for slicing take their default values of 0 and len(s), respectively, whereas the step is set to 2 as specified in the statement.

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

A single colon (e.g., `s[1:2]`) allows for two arguments, the start and the end index. A double colon (e.g., `s[1:2:2]`) allows for three arguments, the start index, the end index, and the step size. If the step size is set to the default value 1, we can use the single colon notation for brevity.

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

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

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

In this case, the default indices are not `start=0` and `end=len(s)` but the other way round: `start=len(s)-1` and `end=-1`. Note that the start index is still included and the end index still excluded from the slice. Because of that, the default end index is -1 and not 0.

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

Slicing works the same for all sequence types. For lists, consider the following example:

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

Slicing of tuples works in a similar way.

6) Why is the last index excluded from the slice?

The last index is excluded for two reasons. The first reason is language consistency: e.g., the range function also does not include the end index. The second reason is clarity—here's an example of why it makes sense to exclude the end index from the slice.

```
customer_name = 'Hubert'
k = 3 # maximal size of database entry
x = 1 # offset
db_name = customer_name[x:x+k]
```

In the above example, there is a requirement that the `db_name` string must not exceed the maximum length of k, which is 3. By slicing `customer_name` using an intuitively understandable index-range of [x:x+k], we ensure that a sliced-string of no more than "k" length is assigned to `db_name`. This is possible because the end index is not included in the slice. Now suppose the end index were included. In this case, we would need to specify an index range of [x:x+k-1] to ensure that the slice assigned to `db_name` is of length k. Since writing the additional "-1" in the index range is counter-intuitive, its very nice of Python to not include the end index in the slice automatically, which makes the code easy to write as well as read, with the simpler index range [x:x+k].

Now you are able to understand indexing and slicing in NumPy. For further reading we recommend this free resource: `http://bit.ly/slicing-book`.

4.8 How Does Indexing and Slicing Work in NumPy?

In NumPy, you have to differentiate between one-dimensional arrays and multi-dimensional arrays because slicing works differently for both.

Let's start with one-dimensional NumPy arrays. They are similar to numerical lists in Python, so you can use slicing in NumPy as you used slicing for lists. Here are a few examples that should be familiar to you from the last section of this tutorial. Go over them slowly. Try to explain to yourself why these particular slicing instances produce the respective results.

```
import numpy as np


a = np.arange(0, 10)
print(a)
# [0 1 2 3 4 5 6 7 8 9]

print(a[:])
# [0 1 2 3 4 5 6 7 8 9]

print(a[1:])
# [1 2 3 4 5 6 7 8 9]
```

```
print(a[1:3])
# [1 2]

print(a[1:-1])
# [1 2 3 4 5 6 7 8]

print(a[::2])
# [0 2 4 6 8]

print(a[1::2])
# [1 3 5 7 9]

print(a[::-1])
# [9 8 7 6 5 4 3 2 1 0]

print(a[:1:-2])
# [9 7 5 3]

print(a[-1:1:-2])
# [9 7 5 3]
```

Pause a moment to reconsider the last two examples. Did you really understand why `a[-1:1:-2]` is exactly the same as `a[:1:-2]`? If you have read the last section about Python's slicing thoroughly, you may remember that the default start index for negative step sizes is -1.

But in contrast to regular slicing, NumPy is a bit more powerful. See the next example of how NumPy handles the assignment of a value to an extended slice.

```
import numpy as np


l = list(range(10))
l[::2] = 999
# Throws error --> assign iterable to extended slice


a = np.arange(10)
a[::2] = 999
print(a)
# [999   1 999   3 999   5 999   7 999   9]
```

Regular Python's slicing method is not able to implement the user's intention, unlike NumPy. In both cases, it is clear that the user wants to assign 999 to every other element in the slice. NumPy has no problems implementing slice assignments.

Let's move on to multi-dimensional slices.

For multi-dimensional slices, you can use one-dimensional slicing for each axis separately. You define the slices for each axis, separated by a comma. Here are a few examples. Take your time to thoroughly understand each of them.

```
import numpy as np


a = np.arange(16)
a = a.reshape((4,4)) # converts a from a single-axis arr
print(a)
# [ 0  1  2  3]
# [ 4  5  6  7]
# [ 8  9 10 11]
# [12 13 14 15]]

print(a[:, 1])
# Second column:
# [ 1  5  9 13]

print(a[1, :])
# Second row:
# [4 5 6 7]

print(a[1, ::2])
# Second row, every other element
# [4 6]

print(a[:, :-1])
# All columns except last one
# [[ 0  1  2]
# [ 4  5  6]
# [ 8  9 10]
# [12 13 14]]

print(a[:-1])
# Same as a[:-1, :]
# [[ 0  1  2  3]
# [ 4  5  6  7]
# [ 8  9 10 11]]
```

As you can see in the above examples, slicing *multi-dimensional* NumPy arrays is easy - if you know NumPy arrays and how to slice *one-dimensional* arrays. The most important information to remember is that you can slice each axis separately. If you don't specify the slice notation for a specific axis, the interpreter applies the default slicing (i.e., the colon :).

Also note that you can "fill in" remaining default slicing colons by using three dots. Here is an example:

```
import numpy as np


a = np.arange(3**3)
a = a.reshape((3,3,3))
print(a)
##[[[ 0  1  2]
##  [ 3  4  5]
##  [ 6  7  8]]
##
## [[ 9 10 11]
##  [12 13 14]
##  [15 16 17]]
##
## [[18 19 20]
##  [21 22 23]
##  [24 25 26]]]
```

```
print(a[0, ..., 0])
# Select the first element of axis 0
# and the first element of axis 2. Keep the rest.
# [0 3 6]
# Equivalent to a[0, :, 0]
```

Having mentioned this detail, we will introduce a very important and beautiful feature of NumPy indexing. This is critical for your success in NumPy so stay with me for a moment.

Instead of defining the slice to carve out a sequence of elements from an axis, you can select an arbitrary combination of elements from the NumPy array. How? Simply specify a boolean array with exactly the same shape. If the boolean value at position (i,j) is True, the element will be selected, otherwise not. As simple as that. Here is an example.

```
import numpy as np


a = np.arange(9)
a = a.reshape((3,3))
print(a)
# [[0 1 2]
# [3 4 5]
# [6 7 8]]

b = np.array(
[[ True, False, False],
[ False, True, False],
[ False, False, True]])
print(a[b])
# Flattened array with selected values from a
# [0 4 8]
```

The matrix b with shape (3,3) is a parameter of a's indexing scheme. Beautiful, isn't it?

In the example, you select an arbitrary number of elements from different axes. How is the Python interpreter supposed to decide about the final shape? For example, you may select four rows for column 0 but only 2 rows for column 1—what's the shape here? There is only one option: the result of this operation has to be a one-dimensional NumPy array.

If you need to have a different shape, feel free to use the `reshape(...)` operation to bring your NumPy array back into your preferred format.

Congratulations—you made it through this NumPy tutorial. Mastering Python's NumPy library is a critical step on your path to becoming a better Python coder and acquiring your data science and machine learning skills.

4.9 NumPy Cheat Sheet

On the last page of this book, you'll find a NumPy cheat sheet with the most important NumPy functions. This NumPy cheat sheet summarizes everything you need to know in order to be able to solve the upcoming code puzzles. You can also download the high-resolution PDF online here:

`https://blog.finxter.com/numpy-cheat-sheet`.

Study this cheat sheet very thoroughly—it will determine your learning success with this book. Feel free to use the cheat sheet for the upcoming NumPy puzzles.

— 5 —

Twenty Puzzles to Strengthen Your NumPy Basics

In the previous chapters, you have learned about the benefits of puzzle-based learning. You have built the foundation of your NumPy education by reading through the tutorial. This chapter strengthens your understanding of the NumPy basics.

Now take your pen, fill your cup of coffee, and let's dive into the practical part of this book: solving code puzzles. The puzzles are very basic in the beginning but will become harder and harder as you proceed with the book. Again, take your time, and try to understand each and every line until you move on to the next puzzle.

If you don't know the syntax of a puzzle, consult the NumPy cheat sheet first before looking up the solution.

5.1 Extracting Array Dimensionality

Some pieces of knowledge are nice-to-know but optional in NumPy. But the `ndim` property is not one of them. Don't read on until you understand it thoroughly.

Puzzle 1

```
import numpy as np

# salary in ($1000) [2015, 2016, 2017]
dataScientist =     [133, 132, 137]
productManager =    [127, 140, 145]
designer =          [118, 118, 127]
softwareEngineer =  [129, 131, 137]

a = np.array([dataScientist,
              productManager,
              designer,
              softwareEngineer])
print(a.ndim)
```

What is the output of this code puzzle (413)?

This puzzle uses data about the salary of four jobs: data scientists, product managers, designers, and software engineers. We create four lists that store the yearly average salary of the four jobs (unit: one thousand dollars) for the three years 2015, 2016, and 2017.

Then, we merge these four lists into a two-dimensional array (denoted as *matrix*). You can think about a two-dimensional matrix as a list of lists. A three-dimensional matrix would be a list of lists of lists. You get the idea.

In the puzzle, each salary list for a single job becomes a row of a two-dimensional matrix. Each row has three columns, one for each year. The puzzle prints the dimensions of this matrix. As our matrix is two-dimensional, the solution of this puzzle is 2.

The correct solution »

```
2
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

5.2 Accessing Array Shape

A basic building block of your proficient NumPy education is the shape of a NumPy array. A multitude of NumPy functions allow you explicitly specify the shape object—and provide a different semantics for you depending on your choice. Don't rush over this puzzle, because it is vital for your effective work as a data scientist.

Puzzle 2

```
import numpy as np

# salary in ($1000) [2015, 2016, 2017]
dataScientist =     [133, 132, 137]
productManager =    [127, 140, 145]
designer =          [118, 118, 127]
softwareEngineer =  [129, 131, 137]

a = np.array([dataScientist,
              productManager,
              designer,
              softwareEngineer])
print(a.shape[0])
print(a.shape[1])
```

What is the output of this code puzzle (412)?

Again, this puzzle uses the same salary data as in the previous puzzle. We create four lists that store the yearly average salary of the four jobs for three contiguous years.

Then, we merge these four lists into a two-dimensional array. Each salary list for a single job becomes a row with three columns, one for each year.

The puzzle prints the shape of this matrix, which is the number of elements in each dimension. For example, a matrix with `n` rows and `m` columns has shape `(n,m)`. As our two-dimensional matrix has 4 rows and 3 columns, the solution to this puzzle is 4 and 3.

The correct solution »

```
4 3
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

5.3 Averaging 1D Arrays

Rows, columns, shape, and reshape—this puzzle is all about strengthening your understanding so that you can grasp the meaning of NumPy code FAST.

Puzzle 3

```
import numpy as np

# apple stock prices (May 2018)
prices = [ 189, 186, 186, 188,
           187, 188, 188, 186,
           188, 188, 187, 186 ]
prices = np.array(prices)

data_3day = prices.reshape(4,3)

print(int(np.average(data_3day[0])))
print(int(np.average(data_3day[-1])))
```

What is the output of this code puzzle (418)?

This puzzle performs a miniature stock analysis of Apple stock in three steps.

First, create a NumPy array from the raw price data. In this case, the raw price data is a simple list of numerical prices (one-dimensional).

Second, create a new array `data_3day` for more convenient analysis: This array bundles the price data from three days into each row. For example, your goal may be to search for coarse price patterns in the data set (beyond the fine-grained level of a single day).

Third, average the 3-day price data of the first and last row using the NumPy `average()` function. This results in data points that are more robust against outliers. Comparing the first and the last 3-day price period reveals that the Apple stock price remains stable in this mini data set.

The correct solution »

```
187 187
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

5.4 Working with Not a Number the Wrong Way

Sometimes we want to indicate that a certain value is missing. For this purpose NumPy provides the `NaN` value. But be careful, it has it's own specific behavior!

Puzzle 4

```
import numpy as np

# Students' quiz results.
# NaN for students that were absent.
data = np.array([10, 3, np.NaN, 7, np.NaN, 5])
students_present = 0
students_absent = 0

for result in data:
    if result == np.NaN:
        students_absent += 1
    else:
        students_present += 1

print(students_present)
```

What is the output of this code puzzle (503)?

In this puzzle we iterate over the `data` array with numbers and `NaN` and count all the values depending on the result of the comparison to `NaN`. The puzzle's result seems to be obvious, but it isn't! NumPy's `NaN` value can't be compared to any other value using the `==` operator since this comparison will always be `False`. NumPy offers the `isnan` function, which has to be used to compare a value to `NaN`. Thus in the puzzle the comparison always results in `False`, and all values of the array are counted as `students_present`.

The correct solution »

```
6
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

5.5 Working with Not a Number the Right Way

Now we want to count the students again, but this time we want to do it the correct way. For comparisons with `NaN`, NumPy provides the `isnan` function. Remember, do not compare `NaN` using the `==` operator! It will always return `False`, even if you compare it to itself.

Puzzle 5

```python
import numpy as np

# Students' quiz results.
# NaN for students that were absent.
data = np.array([10, np.NaN, 7, np.NaN, 5])
students_present = 0
students_absent = 0

for result in data:
    if not np.isnan(result):
        students_present += 1
    else:
        students_absent += 1

print(students_present)
```

What is the output of this code puzzle (504)?

This time our code worked as expected. Since we used `isnan`, the comparison returned `True` only if the value was `NaN`. Yet the puzzle's logic is inverted because of the `not` operator, which inverts the result of the comparison. To achieve the correct result, we increase the `students_present` variable in the `if`-branch and `students_absent` in the `else`-branch. Compare it once again to the puzzle before.

The correct solution »

```
3
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

5.6 Creating Numerical Sequences

The focus of this puzzle is one thing: it wants to teach you to be thorough with edge cases. If you have any experience with coding, you know that edge cases are the source of many bugs.

Puzzle 6

```
import numpy as np

# save $122.50 per month
x = 122.5
net_wealth = np.arange(0, 1000, x)

# how long to save >$1000?
print(len(net_wealth))
```

What is the output of this code puzzle (416)?

This puzzle is about the popular NumPy arange function. You may know the Python built-in `range(x,y,z)` function that creates a sequence of linear progressing values. The sequence starts from `x`, increases the values linearly by `y`, and ends if the value becomes larger than `z`.

The `arange(x,y,z)` function is similar but creates a NumPy array and works with float numbers as well.

Here is the final output of this puzzle:

```
[ 0.   122.5 245.   367.5 490.   612.5 735.   857.5
980.  ]
```

Note that a common mistake in this puzzle is to not account for the first value of the array: 0. Missing these edge cases can inject bugs into your code that are difficult to find. Be especially careful with them, therefore.

The correct solution »

```
9
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

5.7 Creating Numerical Intervals

Sometimes things are as simple as they seem. But are they?

Puzzle 7

```
import numpy as np

year = np.linspace(0, 365, 366)
print(int(year[-1] - year[-2]))
```

What is the output of this code puzzle (417)?

This puzzle is about the useful linspace function. In particular, `linspace(start, stop, num)` returns evenly spaced numbers over a given interval `[start, stop]`. Both integer values start and stop are included in the sequence. Hence, the linspace function is different to most sequential NumPy functions where the stop value is usually not included.

For example, the function call `linspace(0,3,4)` returns the NumPy array sequence 0,1,2,3 (i.e., 4 evenly spaced numbers).

This function is particularly useful when plotting (or evaluating) a function f. For example, you can test output values for evenly distributed input values for function f. In other words, the result of the function f, applied to evenly spaced inputs values, shows how it behaves for growing input values.

The correct solution »

```
1
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

5.8 Initializing Multi-Dimensional Arrays

This book is not only about understanding NumPy. It's about burning the patterns into your head until you don't need to think consciously about code snippets anymore. Let's train your linear algebra understanding of the Hadamard Product and how to initialize NumPy arrays.

Puzzle 8

```
import numpy as np

n = 100 # dimensionality

W = np.zeros((n,n))
for i in range(len(W)):
    W[i][i] = 2

X = np.ones((n,n))

Y = W * X
print(int(Y[-1][-1]))
```

What is the output of this code puzzle (415)?

When working with NumPy, you must be fluent with matrix operations (e.g., multiplication).

This puzzle performs a simple calculation. It tests your understanding of three NumPy concepts.

First, specify the shape of the NumPy array as a tuple `(n,m)` where `n` is the number of rows and `m` the number of columns.

Second, initialize NumPy arrays of a specified shape using the functions `ones()` and `zeros()`. The initial values of such a NumPy array are 1s and 0s, respectively.

Third, perform element-wise multiplication of the matrix values with the operator '*'. Each cell `(i,j)` of the resulting matrix is the product of cell `(i,j)` of the first matrix with cell `(i,j)` of the second matrix. Another term for element-wise matrix multiplication is the *Hadamard Product*.

Finally, we print the last element of the two-dimensional matrix `Y` (bottom-right) using negative indexing.

The correct solution »

```
2
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

5.9 Revisiting Linear Algebra

This puzzle should look familiar to you. Yet there are new things to be discovered! The eye method from the NumPy library can also be used to create matrices, like the ones or zeros methods. Eyes creates a quadratic matrix of zeros with ones on the diagonal. Can you figure out what the k parameter of `eye(n, k)` does if you set it to 1 or -1?

Puzzle 9

```
import numpy as np

# dimensionality
n = 10

W = np.zeros((n, n))
for i in range(len(W)):
    W[i][i] = 2

# Create nxn matrix
M = np.eye(n, k=0)

S = W + 3 * M
print(int(S[-1][-1]))
```

What is the output of this code puzzle (506)?

Instead of using the NumPy `ones` function we used `eye` to create a matrix that consists of zeros and ones on the middle diagonal. If you set the parameter k = 1 or -1 you can move the diagonal up and down; for example, `eye(3, k=1)` creates the matrix

```
[[0, 1, 0],
[0, 0, 1],
[0, 0, 0]]
```

The correct solution »

```
      5
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

5.10 Understanding the Hadamard Product

The purpose of this puzzle is to save you a lot of time debugging your NumPy code. The default multiplication operator performs the Hadamard Product. You don't know what the Hadamard Product does? Learn how it works by solving this puzzle!

Puzzle 10

```
import numpy as np

# salary in ($1000) [2015, 2016, 2017]
dataScientist =     [133, 132, 137]
productManager =    [127, 140, 145]
designer =          [118, 118, 127]
softwareEngineer =  [129, 131, 137]

# Salary matrix
S = np.array([dataScientist,
              productManager,
              designer,
              softwareEngineer])

# Salary increase matrix
I = np.array([[1.1, 1.2, 1.3],
              [1.0, 1.0, 1.0],
              [0.9, 0.8, 0.7],
              [1.1, 1.1, 1.1]])
```

```
# Updated salary
S2 = S * I
print(S2[2][0] > S[2][0])
```

What is the output of this code puzzle (425)?

Consider the salary data you have already seen before. There are four jobs: data scientist, product manager, designer, and software engineer. We create four lists that store the yearly average salary of the four jobs in thousands of dollars for three subsequent years. We merge these four lists into a two-dimensional array (denoted as *matrix*). Each salary list of a single job becomes a row of this matrix. Each row has three columns, one for each year.

Now suppose your company changes the salary for the different job descriptions. For example, every data scientist will enjoy a salary raise of 30% in 2017.

In the puzzle, we create a second matrix `I` that stores the salary changes as weights. Then, we update the salaries according to these weights. As designers in 2015 got a salary decrease, i.e., the weight is smaller than 1.0, the new salary is smaller than the old salary.

The multiplication operator '*' creates a new matrix by multiplying the two values at position `(i,j)` of the two matrices. We call this the *Hadamard Product.* This often leads to confusion because NumPy beginners expect this to be standard matrix multiplication: *multiply the i-th row of the first matrix with the j-th column of the second matrix.*

The correct solution »

```
False
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

5.11 Broadcasting

Broadcasting, i.e., automatically modifying the shape of NumPy arrays, is the source of much confusion and many bugs in NumPy. But it can be avoided—by solving this puzzle.

Puzzle 11

```
import numpy as np

# salary in ($1000) [2015, 2016, 2017]
dataScientist =     [130, 132, 137]
productManager =    [127, 140, 145]
designer =          [118, 118, 127]
softwareEngineer =  [129, 131, 137]

A = np.array([dataScientist,
              productManager,
              designer,
              softwareEngineer])

# salary raise for data scientists
B = np.ones((4,1))
B[0] = 1.1
C = A * B
print(int(C[0][0]))
```

What is the output of this code puzzle (414)?

This puzzle assumes that you already have basic knowledge of the NumPy library. You know how to create a NumPy array (or matrix) as a list of lists. The matrix A holds the salary data (in $1000) of four job descriptions for the three years 2015, 2016, and 2017.

Suppose that data scientists got a salary rise of 10% for the past three years. How do we achieve this?

A convenient way is to use matrix multiplication. In particular, vector `B` holds the factors with which each job in matrix `A` should be multiplied. The goal is to multiply the first row (index 0) by 1.1 and all other rows by 1, keeping them equal.

The array creation routine `np.ones()` takes the matrix shape (e.g., `(n,m)` for `n` rows and `m` columns) and creates a new matrix with value 1 for each cell.

Please note that the *-operator calculates the Hadamard Product and not standard matrix multiplication. In fact, it would not even be possible to multiply a `(4,3)` matrix with a `(4,1)` vector because you can only multiply `(n,m)` with `(m,o)` matrices. Instead, NumPy uses the useful feature of *broadcasting*. In other words, it automatically fixes the shapes of incompatible arrays by *expanding* the smaller one. In this example, NumPy creates a (virtual) matrix consisting of three times the column vector `B`.

Finally, we print the top-left element of the resulting matrix. This is the increased salary of a data scientist in the year 2015. Previously they earned $130,000, which is raised by 10% to $143,000.

The correct solution »

```
143
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

5.12 Practicing Simple Indexing

Let's keep your inoperative brain cells busy—by playing Python interpreter in your head. Perform these simple numerical computations and beware the traps in this puzzle!

Puzzle 12

```
import numpy as np

# air quality index AQI data
hong_kong = np.array(
    [ 42, 40, 41, 43, 44, 43 ])
new_york = np.array(
    [ 30, 31, 29, 29, 29, 30 ])
montreal = np.array(
    [ 11, 11, 12, 13, 11, 12 ])

hk_mean = (hong_kong[0] +
           hong_kong[-1]) / 2.0
ny_mean = (new_york[1] +
           new_york[-3]) / 2.0
m_mean = (montreal[1] +
          montreal[-0]) / 2.0

print(hk_mean)
print(ny_mean)
print(m_mean)
```

What is the output of this code puzzle (431)?

This puzzle strengthens your understanding of basic indexing NumPy array elements.

The puzzle analyzes data from the real-time air quality index (AQI) for the three cities Hong Kong, New York, and Montreal. The index data aggregates various factors that influence air quality such as respirable particulate matter, ozone, and nitrogen dioxide. The goal is to compare the air quality data for the three cities. To show how indexing works, we use different indexing schemes to access two data values for each city. Then, we divide the sum of those values by 2.0 to normalize the result.

To access NumPy array elements, you can use positive or negative indices. For positive indices, use 0 to access the first element and increment the index by 1 to index the next element. For negative indices, use -1 to access the last element and decrement the index by 1 to access the previous element. It's as simple as that.

The correct solution »

```
42.5 30.0 11.0
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

5.13 The Boolean Indexing Trick

Do you like tricks and hacks? This puzzle shows you a beautiful way of using boolean indexing for fine-grained array access in NumPy.

Puzzle 13

```
import numpy as np


years = np.array(
    [[ 2000, 2001, 2002],
    [ 2003, 2004, 2005],
    [ 2006, 2007, 2008]])

leap_years_indices = np.array(
    [[ True, False, False],
     [ False, True, False],
     [ False, False, True]])
print(years[leap_years_indices][-1])
```

What is the output of this code puzzle (434)?

How to select specific array elements? In the introductory tutorial, you have already learned about Boolean indexing to do this.

Indexing elements of a NumPy array is as easy as defining an indexing array with exactly the same shape. Then, you pass this indexing array into the bracket indexing operator.

You will need this feature for advanced NumPy functions such as filtering out values. So learn the basics now, and you will find it much easier to master advanced NumPy code bases later.

The correct solution »

```
2008
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

5.14 Slicing Matrices Like Paper

Are you prepared to earn you PhD in slicing? Here we go!

Puzzle 14

```
import numpy as np

# Prints out a multi dimensional numpy array as
↪  one line
# Example: np.array([['a'], ['b'], ['c']]) -->
↪  abc
def glow(snippet):
    for c in snippet:
        print(*c, end='')


newspaper = np.array([[' ', 'a', 'p', 'p'],
                      ['b', 'a', 'n', 'a'],
                      ['t', 'o', 'm', 't'],
                      ['c', 'u', 'c', 'a'],
                      ['l', 'e', 's', 'e'],
                      ['p', 'r', 'e', 's'],
                      ['c', 'a', 'r', 's'],
                      ['y', 'e', 'a', 'r']])

snippet1 = newspaper[2:5, -1:][::-1]
snippet2 = newspaper[0, :]
snippet3 = newspaper[4, :-1]
```

```
glow(snippet1)
glow(snippet2)
glow(snippet3)
```

What is the output of this code puzzle (505)?

Did you solve the puzzle correctly? If so, you have just earned your PhD in slicing! Congratulations!
For those who need more explanation, keep reading:

The glow function takes a slice of the given array, which is itself a NumPy array, and prints out all the letters in the array. Don't worry if you do not understand it right now—it is pretty complicated.
The slicing of the three snippets works as follows:
1.) `newspaper[2:5, -1:][::-1]`
In the first pair of brackets, `2:5` means that we cut out rows number 2, 3 and 4. The second value, `-1:`, indicates the columns to cut out—in this case it's only the last column. So this first part returns a NumPy array which contains `"tae"`. With the second pair of brackets, this array is reversed and results in `"eat"`.
2.) `newspaper[0, :]`
The number before the comma indicates the first row of the matrix; the : says that we take all the columns, without restriction.
3.) `newspaper[4, :-1]`
This time we select the fourth row and columns starting

from the first up to the last. The last column is not contained since the rule is always: start index is included, end index is excluded.
In conclusion, the general rule for matrix slicing is:
`matrix[rows_start:rows_end, cols_start:cols_end]`
where start is included and end is excluded. To take a row or column including the last index, use:
`matrix[rows_start:, cols_start:]`
Type in the code and try it out to get a proper understanding of how it works.

The correct solution »

```
eat apples
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

5.15 Simple Array Logic

What are some of the most important Instagram accounts? Find the answer in this puzzle—and learn how to use boolean operators on NumPy arrays as if they were simple numerical values.

Puzzle 15

```
import numpy as np

# popular instagram accounts
# (millions followers)
inst = [232, #"@instagram"
        133, #"@selenagomez"
        59,  #"@victoriassecret"
        120, #"@cristiano"
        111, #"@beyonce"
        76]  #"@nike"

inst = np.array(inst)
superstars = inst > 100
print(superstars[0])
print(superstars[2])
```

What is the output of this code puzzle (420)?

The following handy NumPy feature will prove useful throughout your career. *Use comparison operators directly on NumPy arrays.* The result is an equally-sized NumPy array with Boolean values. Each Boolean value in cell `(i,j)` indicates whether the comparison evaluates to `True` for its respective value in cell `(i,j)` of the original array.

The puzzle creates a list of integers. Each integer represents the number of followers of popular Instagram accounts (in millions). First, we convert this list to a NumPy array. Then, we determine for each account whether it has more than 100 million followers.

We print the first and the third Boolean value of the resulting NumPy array. The result is `True` for `@instagram` with 232 million followers and `False` for `@victoriassecret` with 59 million followers.

The correct solution »

```
True
False
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

5.16 Mastering Slice Assignments

You may know about slicing in NumPy, i.e., carving out sequences of values from NumPy arrays. But do you use *slice assignments* in your code? If used properly, they are a powerful tool in your data science tool belt.

Puzzle 16

```
import numpy as np

# The fibonacci series
F = np.array([0, 1, 1, 2, 3, 5, 8])
F[::3] = 0
print(sum(F[:4]))
```

What is the output of this code puzzle (426)?

This puzzle demonstrates advanced indexing and slicing methods in NumPy arrays. You most likely know indexing of Python lists or strings. Indexing of NumPy arrays works in a similar way.

There are two interesting twists in this puzzle. First, we overwrite each third value of the NumPy array with the concise slice assignment `F[::3]`, starting from the very first entry. Note that slice assignments happen if you use slicing on the *left-hand side* of the assignments. Second, we print the sum of the first four values using slicing to access them. As we have overwritten the first and the fourth value, the sum is only $0 + 1 + 1 + 0 = 2$.

The correct solution »

```
2
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

5.17 Sorting an Array (Part 1)

Imagine for a moment that you are in a library containing thousands of books. What makes it possible for you to look up and read the book of your choice? The *sorted* placement of the books of course.

Sorting is one of the most fundamental operations that inhibits chaos and brings order in all computing algorithms. Solve the following puzzle to master this important operation using NumPy. You will also learn how Python's indexing tricks can add detail to the sorting operation.

Puzzle 17

```
import numpy as np

# Quiz scores for different students
scores = np.array([7, 6, 8, 5, 9])

scores = np.sort(scores)
scores = scores[::-1]

print(scores[-2])
```

What is the output of this code puzzle (994)?

In this puzzle, we are given an input array `scores` containing student-scores in a class quiz. We sort this array in ascending order using NumPy's `sort()` function. In the next step, you reverse the order of elements in the sorted array by using Python's slicing with a negative increment. After this operation, the `scores` array contains the quiz-scores in descending order of their values. Hence, printing the second last of these scores gives 6 as the solution for this puzzle.

The correct solution »

```
6
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

5.18 Sorting an Array (Part 2)

For some data science problems, it is not enough to just sort your raw data. It may be equally important to also know the exact re-arrangement of the raw data, i.e., the re-arranged indices of its elements that resulted in its sorted form. To find out why this is so, challenge yourself by solving the following puzzle.

Puzzle 18

```
import numpy as np

# Sensor IDs and the corresponding
# sensor values
ids = [56, 61, 33, 17, 82]
values = np.array([10, 6, 8, 7, 9])

indices = np.argsort(values)

print(ids[indices[2]])
```

What is the output of this code puzzle (995)?

Initially, we are given two arrays: `ids` and `values`. Then, using NumPy's `argsort()` function, we sort the values of sensor readings in ascending order. However, instead of returning the sorted array, this function returns the indices of elements from the original `values` array in the order in which they occur in the sorted array. For example, for the value 10, which is the largest value in the `values` array, its index 0 becomes the last element of the indices array.

By accessing `indices[2]`, we access the index of the third smallest value in the `values` array, i.e., 8. So `indices[2]` gives us the index of 8, which is 2. Finally, in the last line, we want to print the corresponding ID of the sensor with the third smallest reading, i.e., 8. Therefore, we print `ids[indices[2]]`, which is 33.

The correct solution »

```
33
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

5.19 Computing Array Element Differences

In this puzzle, your understanding of an important NumPy function, `diff()`, will be put to the test. You will realize in the later part of this book that the `diff()` function can help you in neatly solving various real-world data science problems.

Puzzle 19

```
import numpy as np

# Fibonacci Sequence with
# first 8 numbers
fibs = np.array([0, 1, 1, 2, 3,
                 5, 8, 13, 21])

diff_fibs = np.diff(fibs)

print(diff_fibs[4])
```

What is the output of this code puzzle (991)?

We are given a NumPy array of the first eight Fibonacci numbers (`fibs`). We find the difference between consecutive numbers in `fibs` using the NumPy `diff()` function. These differences are stored as another array (`diff_fibs`). Finally, we print the 5^{th} element of `diff_fibs`, which represents the difference between the 6th and the 5th element of the `fibs` array. Hence, the solution is 2 (5-3). It is important to remember that since the `diff()` function returns an array of differences, this array's size is smaller than the length of the input array by 1 element.

The correct solution »

```
2
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

5.20 Computing Array of Cumulative Sums

Cumulative sums represent another important tool for a data scientist to assess the distribution of values in the datasets of interest. For instance, these sums could be used to answer various questions such as *after how many donations did the sum of the gathered amount pass* $ 1000?

Puzzle 20

```
import numpy as np

# Per-day charity donations in thousands
# of dollars received in a charity compaign
donations = np.array([2, 3, 2.5,
                      3.5, 4, 3.5])


cum_donations = np.cumsum(donations)

print(cum_donations[-4])
```

What is the output of this code puzzle (992)?

In this puzzle, we are given a NumPy array named `donations`. Next, we use NumPy's `cumsum()` function to generate the cumulative sum of this array. Hence, the i^{th} element of the `cum_donations` array contains the sum of the donations starting from the first element of the `donations` array until its i^{th} element.

In the end, the puzzle prints the 4^{th}-last element of the `cum_donations` array, which basically represents the sum of the first three elements of `donations`, i.e., $2+3+2.5 = 7.5$.

The correct solution »

```
7.5
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

— 6 —

Eleven Puzzles to Kickstart Your Understanding of Linear Algebra and Statistics in NumPy

How to enter the next level in NumPy? Your NumPy skills have now reached an intermediate level. Now you need to learn about the basics of linear algebra—and how to calculate basic statistics on one-dimensional and multi-dimensional NumPy arrays. This chapter will polish your NumPy skills by presenting you with 11 puzzles about linear algebra and statistics.

6.1 Calculating 1D Dot Product

What's a simple and robust machine learning algorithm? Correct, *linear regression*. In this puzzle, you will learn how to perform simple stock price predictions with NumPy.

Puzzle 21

```
import numpy as np

# simple regression model
W = np.array([0.7, 0.2, 0.1])

# Google stock prices (in US-$)
# [today, yesterday, 2 days ago]
x = np.array([1131, 1142, 1140])

# prediction
y = np.dot(W, x)

# do we expect growing prices?
if y > x[0]:
    print("buy")
else:
    print("sell")
```

What is the output of this code puzzle (421)?

This puzzle predicts the stock price of Google (now: Alphabet). We use three-day historical data and store it in the NumPy array `x`.

The NumPy array `W` represents our prediction model. More precisely, `W` contains the weights for the past three days—how much each day contributes to the prediction.

In machine learning, you would denote this array as the weight vector.

The code predicts the stock price for tomorrow based on the stock prices of the most recent three days. But today's stock price should have a higher impact on the prediction than yesterday's stock price. Thus, it weights today's stock price with the factor 0.7.

In the puzzle, the stock prices of the last three days are \$1132, \$1142, and \$1140. The predicted stock price for the next day is y$= 0.7 * \$1132 + 0.2 * \$1142 + 0.1 * \$1140 = \$1134.8$.

You can implement this linear combination of the most recent three days' stock prices by using the dot product of the two vectors.

To get the result of the puzzle, you do not have to compute the result of the dot product. It is enough to see that the predicted stock price is higher than today's stock price.

The correct solution »

```
buy
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

6.2 Multiplying 2D Matrices

Professional NumPy code often contains the '@' sign. What's its purpose? Learn about the concise @ operator in this puzzle.

Puzzle 22

```
import numpy as np

# graphics data
a = [[1, 1],
     [1, 0]]
a = np.array(a)

# stretch vectors
b = [[2, 0],
     [0, 2]]
b = np.array(b)
c = a @ b
d = np.matmul(a,b)
print((c == d)[0,0])
```

What is the output of this code puzzle (422)?

This puzzle shows an important application domain of matrix multiplication: computer graphics.

We create two matrices `a` and `b`. The first matrix `a` is the data matrix (consisting of two column vectors `(1,1)` and `(1,0)`). The second matrix `b` is the transformation matrix that transforms the input data. In our setting, the transformation matrix simply stretches the column vectors.

More precisely, the two column vectors `(1,1)` and `(1,0)` are stretched by factor 2 to `(2,2)` and `(2,0)`. The resulting matrix is therefore `[[2,2],[2,0]]`. To get the final result, we access the first row and first column.

The puzzle uses matrix multiplication to apply this transformation. NumPy supports two syntaxes for matrix multiplication: the matmul function and the @ operator. Note again that the * operator performs element-wise matrix multiplication and is therefore different.

Comparing two equal-sized NumPy arrays results in a new array with Boolean values. As both matrices `c` and `d` contain the same data, the result is a matrix with only `True` values.

The correct solution »

```
True
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

6.3 Enhancing Vector Operations

Working with vectors has never been so convenient as with NumPy arrays. Any vectors of any dimensions can be added or subtracted in a single line! Remember, you add two vectors by adding each of the entries with it's corresponding entry from the other vector.

Puzzle 23

```
import numpy as np

# Two points in 3D space
a = np.array([5, -3, 2])
b = np.array([1, 0, 2])

c = b - a
vec_len = np.sqrt(sum(c ** 2))

print(int(vec_len))
```

What is the output of this code puzzle (512)?

This puzzle shows how the power of NumPy arrays can be leveraged to perform complex mathematical computations. First the distance between two given points in 3-D space is computed. Given a point $a = (a_1, a_2, a_3)$ and a point $b = (b_1, b_2, b_3)$, the distance vector between them is equal to $(b_1 - a_1, b_2 - a_2, b_3 - a_3)$. With NumPy arrays this complex computation is reduced to one simple line `c = b - a`. In the following line we compute the length of the distance vector c. The mathematical formula is $sqrt(c_1^2 + c_2^2 + c_3^2)$. Once again the power of numpy arrays makes it possible to write this complex computation in one line. `c ** 2` squares each entry of the vector, the `sum` function adds them up and `sqrt` computes the square-root of the sum. Imagine how long our lines would have become with vectors from 10D space if we had written each entry specifically! The code from the puzzle works for vectors from n-D space without any change. This is the true power of NumPy!

The correct solution »

```
5
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

6.4 Linear Algebra Made Simple

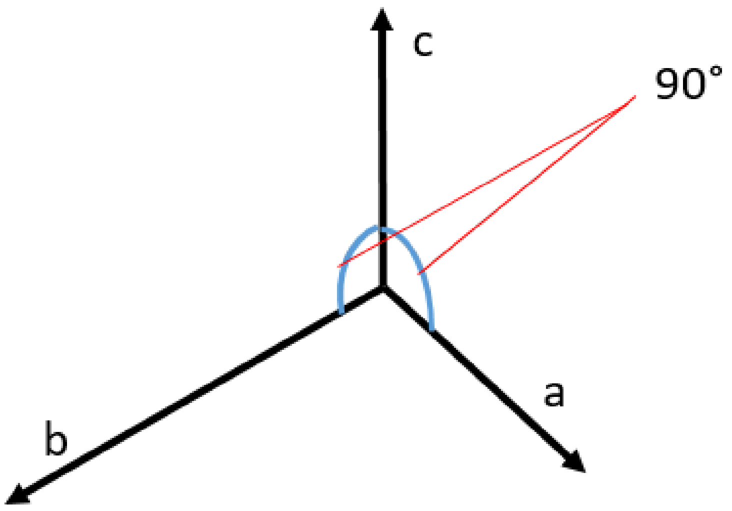
A good understanding of the underlying methods is very important to succeed in data science. Do you understand what is happening here?

Puzzle 24

```
import numpy as np

# Two vectors in 3D space
a = np.array([5, -3, 2])
b = np.array([1, 0, 2])

c = np.cross(a, b)
scalar_prod = np.dot(a, c)

print(scalar_prod)
```

What is the output of this code puzzle (513)?

This puzzle shows a simple fact of linear algebra: The scalar product of two orthogonal vectors is zero. In the first step we use the `cross` function which computes the cross product of vectors a and b. The result of the cross product is a third vector c that stands at a right angle to the other vectors. In the second step we use the `dot` function and compute the scalar product of the vectors. Since vector c was computed to be orthogonal to vector a, the scalar product is 0.



Do the math yourself:

$$
\begin{aligned}
&a \times b \\
&= \begin{pmatrix} (-3 \cdot 2) - (2 \cdot 0) \\ (2 \cdot 1) - (5 \cdot 2) \\ (5 \cdot 0) - (-3 \cdot 1) \end{pmatrix} \\
&= \begin{pmatrix} -6 \\ -8 \\ 3 \end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
&a \cdot c \\
&= (5 \cdot -6) + (-3 \cdot -8) + (2 \cdot 3) \\
&= -30 + 24 + 6 \\
&= 0
\end{aligned}
$$

The correct solution »

0

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

6.5 Revisiting Average

I wrote this puzzle during the last football world cup. During five consecutive games, which team scored more goals on average—Brazil or Germany?

Puzzle 25

```
import numpy as np

# Goals in five matches
goals_brazil = np.array(
    [1,2,3,1,2])
goals_germany = np.array(
    [1,0,1,2,0])

br = np.average(goals_brazil)
ge = np.average(goals_germany)
print(br>ge)
```

What is the output of this code puzzle (424)?

This puzzle introduces a new feature of the NumPy library: the average function. When applied to a one-dimensional NumPy array, this function returns the average value of all values in this NumPy array.

In the puzzle, the average of the goals of the last five games of Brazil is 1.8 and of Germany is 0.8. On average, Brazil shot one more goal per game.

The correct solution »

```
True
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

6.6 Reshaping 1D Arrays

If you read NumPy code from data science professionals, you will soon realize that they are masters in preprocessing, shaping, and reshaping multi-dimensional array data. NumPy arrays are like clay for the craftsman—manipulate them to fit your imagination.

Puzzle 26

```
import numpy as np

def makeTwoDim(x):
    if len(x.shape)==1:
        x = x.reshape(len(x) // 2, 2)
    return x

# apple stock prices (May 2018)
prices = [ 189, 186, 186, 188,
           187, 188, 188, 186,
           188, 188, 187, 186 ]
data = np.array(prices)
data = makeTwoDim(data)
print(np.average(data[-1]))
```

What is the output of this code puzzle (419)?

This advanced puzzle combines multiple language concepts and NumPy features. The topic is a miniature stock analysis of Apple stock.

The puzzle creates a one-dimensional NumPy array from raw price data. Note that the shape property of an n-dimensional NumPy array is a tuple with n elements. Thus, a one-dimensional NumPy array has a `shape` property of length one.

The function `makeTwoDim(x)` uses this to detect one-dimensional NumPy arrays. Then, it transforms these into two-dimensional arrays using the reshape function. For each of the two dimensions, the reshape function needs the number of elements of this dimension. In particular, it creates a NumPy array with two columns and half the number of rows.

Finally, we access the last row of this NumPy array, which contains the two numbers 187 and 186, and average them.

The correct solution »

```
186.5
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

6.7 Averaging 2D Arrays

Do you know Elon Musk's SolarX company? Here is another puzzle that helps you train your understanding of basic NumPy functionality.

Puzzle 27

```
import numpy as np

# stock prices (3x  per day)
# [morning, midday, evening]
solar_x = np.array(
    [[2, 3, 4], # day 1
     [2, 2, 5]]) # day 2

print(np.average(solar_x))
```

What is the output of this code puzzle (432)?

This puzzle again addresses the average function of the NumPy library. When applied to a 1D NumPy array, this function returns the average of the array values. When applied to a 2D NumPy array, NumPy simply flattens the array. The result is the average of the flattened 1D array.

In the puzzle, there is a matrix with two rows and three columns. The matrix gives the stock prices of the `solar_x` stock. Each row represents the prices for one day. The first column specifies the morning price, the second the midday price, and the third the evening price.

Note that NumPy calculates the average as the sum over all values, divided by the number of values. The result is a float value.

The correct solution »

```
3.0
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

6.8 Weighted Averaging Along Axes

How do you calculate the weighted average of a 2D Numpy array along an axis? You may think that this puzzle is only about the average operator in NumPy. But, in fact, it teaches you general-purpose NumPy concepts that are valid for a multitude of functions.

Puzzle 28

```python
import numpy as np

# daily stock prices
# [morning, midday, evening]
solar_x = np.array(
    [[2, 3, 4], # today
     [2, 2, 5]]) # yesterday

# midday - weighted average
print(np.average(solar_x, axis=0,
                 weights=[3/4, 1/4])[1])
```

What is the output of this code puzzle (433)?

You have already learned about the `average()` function in the NumPy library. This puzzle shows you an advanced usage of this function using the axis and weights parameters. I decided to include this usage because it is relevant not only for averaging but also for many different NumPy functions, such as the variance or standard deviation.

Again, there is a matrix with two rows and three columns. The matrix gives the stock prices of the `solar_x stock`. Each row represents the prices for one day. The first column specifies the morning price, the second the midday price, and the third the evening price.

Now, say you do not want to compute the average of the flattened matrix but the average of the midday prices. Moreover, you want to overweight the most recent stock price. Roughly speaking, today accounts for three-quarters and yesterday for one-quarter of the final average.

NumPy gives you control over this via the weights parameter in combination with the axis parameter. The weights parameter defines the weight for each value participating in the average calculation. The axis parameter specifies the direction along which the average should be calculated. In a 2D matrix, the rows are specified as `axis=0` and the columns as `axis=1`. Say you want to know three average values, for the morning, midday, and evening. So you calculate the average along the rows, i.e., `axis=0`. This results in three average values—one for each time of day. Now you take the second element to get the midday variance.

The correct solution »

```
2.75
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

6.9 Calculating 1D Variance

Variance is a powerful concept in data science and machine learning. Think about it: variance measures the spread of your data along certain directions. Roughly speaking, the variance is the basis of all patterns in your data set that are exploited by machine learning algorithms.

Puzzle 29

```
import numpy as np

# Goals in five matches
goals_croatia = np.array(
    [0,2,2,0,2])
goals_france = np.array(
    [1,0,1,1,0])

c = np.var(goals_croatia)
f = np.var(goals_france)
print(c>f)
```

What is the output of this code puzzle (427)?

This puzzle introduces a new feature of the NumPy library: the variance function. The variance is the average squared deviation from the mean of the values in the array. When applied to a 1D NumPy array, this function returns the variance of the array values.

In the puzzle, the variance of the goals of the last five games of England is 0.96 and of France is 0.24. But you do not need to know the exact values to see that the variance in goals scored by England is larger.

The correct solution »

```
True
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

6.10 Axis Variance of a 2D Array

Say, you are working as a financial analyst in an investment firm. Your client fears risky stock investments. How can you convince him that Apple stock is a low-risk, low-variance investment?

Puzzle 30

```
import numpy as np

# stock prices (3x  per day)
# [morning, midday, evening]
APPLE = np.array(
    [[50,60,55], # day 1
     [60,60,65]]) # day 2

# midday variance
y = np.var(APPLE, axis=0)[1]

print(int(y))
```

What is the output of this code puzzle (428)?

This puzzle shows you an important feature of the NumPy library: the variance function. The variance is the average squared deviation from the mean of the values in the array. When applied to a 1D NumPy array, this function returns the variance of the array values. When applied to a 2D NumPy array, NumPy simply flattens the array.

The result is the variance of the flattened 1D array.

In the puzzle, there is a matrix with two rows and three columns. The matrix gives the stock prices of Apple stock. Each row represents the prices for one day. The first column specifies the morning price, the second the midday price, and the third the evening price.

Now, we do not want to know the variance of the flattened matrix but the variance of the midday prices. On both days, the midday price was $60. Therefore, the variance is 0.

You have already learned about the `axis` parameter in NumPy. In a 2D matrix, the different rows are specified as `axis=0` and the different columns within a row as `axis=1`. Do you want to know three variances, for the morning, midday, and evening? Simply calculate the variance along the rows, i.e., `axis=0`. This results in three variance values. Now we take the second element to get the midday variance.

The correct solution »

```
0
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

6.11 1D Axis Standard Deviation

You may know about standard deviation—one of the most important concepts in statistics—but can you compute it in your head? Well, maybe you can play to your strengths as a human to solve this puzzle.

Puzzle 31

```
import numpy as np

# daily stock prices
# [open, close]
google = np.array(
    [[1239, 1258], # day 1
     [1262, 1248], # day 2
     [1181, 1205]]) # day 3

# standard deviation
y = np.std(google, axis=1)

print(y[2] == max(y))
```

What is the output of this code puzzle (429)?

This puzzle introduces the standard deviation function of the NumPy library. As for the variance and the average functions, when applied to a 1D NumPy array, this function returns its standard deviation. When applied to a 2D NumPy array, NumPy simply flattens the array.

The result is the standard deviation of the flattened 1D array.

The puzzle shows a matrix with three rows and two columns. The matrix stores the open and close prices of Google stock for three consecutive days. The first column specifies the opening price, the second the closing price.

We are interested in the standard deviation of the three days. How much does the stock price deviate from the mean between the opening and the closing price?

Again, NumPy provides this functionality via the `axis` parameter. By now you should know that for a 2D matrix, the rows are specified as `axis=0` and the columns within a row as `axis=1`. Say you want to compute the standard deviation along the column, i.e., `axis=1`. This results in three standard deviation values—one per day.

Clearly, on the third day, you observe the highest standard deviation.

The correct solution »

```
True
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

— 7 —

Thirteen Puzzles for Practical Data Science

How are you doing so far? Maybe you wonder how you can make practical use of the skills you have acquired. This chapter shows you 13 practical data science tasks and how to solve them with your recently acquired NumPy skills.

7.1 Statistical Operations

The internet of things is coming. Soon, there will be smart sensors all around us. Here is one such "smart" computation performed by an IoT sensor.

Puzzle 32

```
import numpy as np

temp_sensor = np.array(
    [ 18, 22, 22, 18 ])

mean = np.mean(temp_sensor)
std = np.std(temp_sensor)

print(str(int(mean - std))
      + "-" +
      str(int(mean + std)))
```

What is the output of this code puzzle (430)?

This puzzle is about the standard deviation function in the NumPy library.

In the puzzle, we have four temperature values as measured by a temperature sensor. The goal is to determine the temperature interval in which 68.2 percent of the temperature values fall. This can be interesting for outlier detection. By definition, these are all points that fall within one standard deviation around the mean. In other words, what is the range of normal temperature values based on our data?

The mean value of our data is 20. The standard deviation function calculates the squared distances from each data value to the mean and sums them together. In particular, the function performs the following computation: (i) It sums the squared distances to the mean, with the result 4+4+4+4=16. (ii) It normalizes the previous result by the number of data values, i.e., 16/4 =4. (iii) It calculates the root of the previous result, i.e., sqrt(4)=2. Thus, the resulting interval ranges from 20-2 to 20+2, i.e., 18–22.

The correct solution »

```
18-22
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

7.2 Data Cleaning or Living in an Unperfect World

When you are working with real world data, it will not always be ready to be processed—for example, values could be missing. In this puzzle we replace the missing data point with the mean value of the other data points. Filling in missing data points or removing data with too many missing values is called data cleaning. It is important to mention that data cleaning is not about cheating. We do not want to change the data's statement!

Puzzle 33

```
import numpy as np

# Data from faulty sensor.
sensor_data = np.array([np.NaN, 5, 4, -3])

sensor_data[0] = sensor_data[1:].mean()
int(np.mean(sensor_data))
```

What is the output of this code puzzle (502)?

In data science, most real world data contains missing values and has to be cleaned before it can be processed. Cleaning means either removing data with too many missing values or, if possible, substituting the missing values with a computed value. In the puzzle we substitute the missing value with the mean that we compute from the other values. Once the `NaN` value is replaced, we can work with our data. Computing the mean on an array with `NaN` values results in `NaN`; thus we use slicing to cut out the `NaN` value before computing the mean. Finally, we compute the mean again, this time on the whole array.

The correct solution »

```
2
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

7.3 Understandig the Basics of Filters

A filter or Boolean index is a NumPy array with Boolean values. In the following puzzles we will explore filtering in more depth. In this puzzle you will learn a handy way to work with Boolean values.

Puzzle 34

```
import numpy as np

bool_array = np.array([True, False, True])

print(sum(bool_array) > sum([1, 0, 1]))
```

What is the output of this code puzzle (511)?

In Python, the Boolean value `True` counts as 1 and `False` as 0. Therefore `sum([True, False, True])` is equal to `sum([1, 0, 1])`. You can use this to get the number of values that satisfy a certain condition, e.g., `sum(np.array([3, 1, 4]) > 2)` = 2.

The correct solution »

```
False
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

7.4 Creating Filters

NumPy makes it easy for programmers to create filters. Look at this puzzle; how many lines do we need to create the filter? In other programming languages such as C, we would have needed a loop to achieve the same outcome!

Puzzle 35

```
import numpy as np

# Students' grades for an examn
grades = np.array(['A', 'B', 'D', 'A', 'E'])
# Filter students who did not pass
grades_filter = grades > 'D'

print(grades_filter[1])
print(grades_filter[-1])
```

What is the output of this code puzzle (509)?

This puzzle shows the impressive power of filters on NumPy arrays. The idea is to find all grades from the grades array that are better than a `‘D’`. With NumPy we do not have to write a loop to iterate over the whole array, we can simply write: `grades > ‘D’`. This returns an array of `True` and `False` values that indicate whether the letter in this position is ‘bigger’ than ‘D’. In our case the filter array is this:
`[False, False, False, False, True]`.

Thus `grades_filter[0]` is `False` and `grades_filter[-1]` is `True`.

The correct solution »

```
False True
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

7.5 Mastering the Power of Filters

Now we put it all together: Using list comprehension, we create a filter for NumPy arrays of tuples. These are pretty advanced concepts! Yet we are confident that by now you will solve this puzzle easily!

Puzzle 36

```
import numpy as np

# Book ratings
ratings = np.array([('Numpy Book',   4.6),
                    ('Harry Potter', 4.3),
                    ('Winnie Pooh',  3.4),
                    ('Python Book',  4.7)])

bestseller_filter = [float(x[1]) > 4.5 for x in
 ↪  ratings]
number_of_bestsellers =
 ↪  len(ratings[bestseller_filter])

print(number_of_bestsellers)
```

What is the output of this code puzzle (508)?

In this puzzle we use list comprehension to create a filter array that is `True` for ratings with a value above `4.5` and `False` if not. Then we pass the filter array to an array of book rating tuples and count the number of results. Can you think of a way to write this code in one single line?

Example solution:
`len([x for x in ratings if float(x[1]) > 4.5])`

The correct solution »

```
2
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

7.6 Applying Filters

In the follwing puzzle you learn how filters or Boolean indexing work. A filter is an array of Boolean values with the same dimensions as the array from which you want to extract values. Using the notation `arr[filter]` you apply the filter or Boolean index to the array `arr`. Any values whose index corresponds to a `True` in the filter will be taken and values whose index correspond to a `False` will be omitted.

Puzzle 37

```
import numpy as np

# Filter for 3x3 matrices
filter_3x3 = np.array([[False, True, False],
                        [True, False, True],
                        [False, True, False]])

inverted_filter = np.invert(filter_3x3)
numbers_matrix = np.arange(1, 10).reshape(3, 3)

print(sum(numbers_matrix[inverted_filter]))
```

What is the output of this code puzzle (510)?

In this puzzle, we define a filter for a matrix with three rows and three columns. The filter is a NumPy array of Boolean values and can be passed to a NumPy array in squared brackets. For each `True` in the filter, the value in the matrix at this specific position is returned. With the `invert` function we inverted the values in the Boolean matrix, so every `True` becomes a `False` and vice versa. This inverted filter matrix is passed to a 3 x 3 matrix with the numbers from 1 to 9 and filters out all odd numbers. Finally all the odd numbers from 1 to 9 are summed up: $1 + 3 + 5 + 7 + 9 = 25$

The correct solution »

```
25
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

7.7 Finding Array Elements

To derive meaningful information from data, the ability to assess and locate useful values within data arrays is imperative. The following puzzle focuses directly on this ability using NumPy.

Puzzle 38

```
import numpy as np

# Sensor values
values = np.array([815, 767, 554,
                    141, 605, 346])

indices = np.nonzero((values < 200) |
                     (values > 800))

print(indices[0][1])
```

What is the output of this code puzzle (993)?

In this puzzle, we are given a NumPy array of various sensor values. Using NumPy's `nonzero()` function, we find the indices of those elements of the `values` array that satisfy the given condition of being less than 200 OR greater than 800.

In more detail, the `nonzero()` function returns the indices along each dimension of the input array as a *tuple of array indices*. Since the `values` array in our puzzle is a one-dimensional array, the `nonzero()` function returns a tuple containing a single array. The puzzle ends by accessing this array using `indices[0]`. The elements of this array are `[0, 3]`, which are the indices of the numbers 815 and 141 in the `values` array. Thus, `indices[0][1]` prints 3, which forms the solution of this puzzle.

The correct solution »

```
3
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

7.8 Leveraging Data Science to Boost Revenues I

A common practice to increase sales in a store is placing products that are frequently bought together close to each other on the shelf. But how can one figure out which these products are? Data science has the solution! It's called *association analysis* and is applied to identify frequent subsets in a number of sets. In the following two puzzles you will analyze a set of transactions in order to identify which products are frequently bought together. Each transaction is a set of zeros and ones that indicate if the product was bought or not. Even if we have a very small amount of data in the examples, these are real world data science problems!

The first measure we compute is the so called support value. It indicates how often a certain item occurs in a given set of transactions.

Puzzle 39

```
import numpy as np

# Each row is one customer's shopping basket
# 1 indicates that a certain item was bought.
transactions = np.array([[0, 1, 1, 0],
                         [0, 0, 0, 1],
                         [1, 1, 0, 0],
                         [0, 1, 1, 1],
                         [1, 1, 1, 0],
                         [0, 1, 1, 0],
                         [1, 1, 0, 1],
                         [1, 1, 1, 1]])

total_count = transactions.shape[0]
occurrences = 0
subsets = transactions[:, 1:3]

for subset in subsets:
    if np.all(subset == 1):
        occurrences += 1

s = occurrences / total_count
print(s > 0.25)
```

What is the output of this code puzzle (500)?

Each row of the matrix is one transaction, this is to say, it shows which items were bought. Each column of the matrix indicates for four different items if they were contained (1) or not (0) in a given transaction. In the code we get the total number of sets, in our case transactions, using the shape method of the NumPy array. Then, we want to count all the transactions where item 2 and item 3 were present. Therefore we use slicing to cut out the two columns in the middle and store them in a new matrix called `subsets`. The matrix `subsets` has two columns now, the first and the last one from the matrix `transactions`. Next, we loop over the matrix `subsets` and compare each entry in the set to 1. This is to say, we check if both item 1 and item 4 are contained in the current transaction. The result is a Boolean array with two values, for example `[False, True]` if the subset was `[0, 1]`. Then, we apply the NumPy `all` function to each of these Boolean arrays, which returns `True` only if all values in the Boolean array are `True`. When both values are `True`, we increase the `occurrences` variable, which means we count all sets in which both items were present. The final value is computed by dividing the number of sets from `subsets` where both values were 1 by the total number of transactions. The result `s` is greater than 0.25, which means that the subset of our two items appears pretty often. In data science this value is called support.

The correct solution »

```
True
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

7.9 Leveraging Data Science to Boost Revenues II

In this puzzle we compute the `confidence` value. It indicates how frequently item B occurs given that item A is contained in a transaction.

Puzzle 40

```
import numpy as np

# each row is one customer's shopping basket
# column 1 = lemonade, column 4 = pizza -> if
# item was bought 1, else 0
transactions = np.array([[1, 0, 1, 1],
                         [1, 0, 0, 1],
                         [1, 1, 0, 0],
                         [0, 1, 1, 1],
                         [1, 1, 1, 0],
                         [0, 1, 1, 0],
                         [1, 1, 0, 1],
                         [1, 1, 1, 1]])

occurrences = 0
lemonade = transactions[:,:1]
pizza = transactions[:,-1:]
lemonade_transactions = sum(lemonade == 1)[0]
subsets = np.concatenate((lemonade, pizza),
                          axis=1)
```

```
for subset in subsets:
    if np.all(subset == 1):
        occurrences += 1

c = occurrences / lemonade_transactions
print(c > 0.5)
```

What is the output of this code puzzle (501)?

Once again the transactions are given as a matrix with a column for each item and a value that indicates if this item was contained in the given transaction (row). Let's say that the item of the first column is lemonade bottles and the last one is pizza. Again, the first step is to extract only the two columns for the items we want to analyze. Since it is the first and the last column of the `transactions` matrix, this time we take another approach. Using slicing we take the first and the last column and store each column separately in one variable. To create the matrix containing only the subsets with lemonade and pizza, we use the `concatenate` function from the NumPy library. `axis=1` indicates that we stitch the two columns together from top to bottom. To compute the number of transactions containing lemonade, we compare the whole column `lemonade` with 1 (each single value) and obtain an array with Boolean values which are `True` if the value in the column was 1 and `False` if not. Next, we use `sum` to count how many `True`s there were. As mentioned previously, being able to sum over Boolean arrays comes in very handy when working with filters. The index [0] has to be used since we sum over a NumPy array so our result is also contained in a NumPy array. The `occurrences` are computed in exactly the same way as in the previous puzzle, and the *confidence* is the result of dividing the number of transactions (or sets) containing lemonade and pizza by the number of transactions containing lemonade. In the transactions from the puzzle, the subset where the first and the last column contain a 1 occurs more than 50 percent of the time. Thus it is a frequent subset—in other words: Put the lemonade next to the pizza! If both items are close to each other on the shelf, people who want to buy one of the items will end up buying both.

The correct solution »

```
True
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

7.10 Finding and Locating Maximum Elements

The following puzzle represents a typical classroom data-science problem that deals with quiz scores of a set of students. Although NumPy makes the puzzle look short and concise, a significant amount of complex operations are handled by each of the code statements. So you are advised to carefully understand each step in order to correctly solve the puzzle.

After solving this puzzle, you are also advised to briefly go through the code again. Then think about how the shape of the input data, e.g., the quiz-scores matrix, as well as the different NumPy functions can help you in writing similarly compact and understandable code for your own data science problems.

Puzzle 41

```
import numpy as np

students = ['Alice', 'Bob', 'Carl', 'David']
subjects = ['Maths', 'Physics', 'Biology']

# Quiz scores
#     - Columns -> subjects
#     - Rows    -> students
scores = np.array([[10, 8, 5],
                    [6, 9, 8],
                    [9, 9, 8],
                    [7, 5, 9]])

# Find highest scores in each column
max_subject_score = np.max(scores, axis=0)

# Find row indices with highest scores
row_indices = np.argmax(scores, axis=0)

# Print some output:
for subj_ind, st_ind in enumerate(row_indices):
    print((subjects[subj_ind],
           students[st_ind],
           max_subject_score[subj_ind]))
```

What is the output of this code puzzle (996)?

Here, we are initially given two arrays: `students` and `subjects`. A third `scores` 2D array (or a matrix) provides quiz scores for each of the four students as the four rows of the matrix. Moreover, the columns of this `scores` array represent the quiz scores in each subject. Hence, the first row of the matrix represents Alice's scores in each of the three subjects.

The goal of this puzzle is to print, for each subject, the name of the student who scored the highest along with the obtained highest score value. To achieve this, we carry out two important steps. Firstly, we find the highest scores in each column of the `scores` matrix and save these scores in `max_subject_scores`. Secondly, we also save, inside a new array `row_indices`, the indices of the rows in each column that gave us the maximum subject scores. Note that these indices correspond to the indices of the names in the `students` array. Thus, by doing these two steps, we not only have the best scores in each subject, we also have indices of the names of the students who obtained these scores. In the final `for` loop, we go over each of these indices and print the corresponding student's name, the subject's name, and the score obtained.

The correct solution »

```
 ('Alice', 'Mathematics', 10)
('Bob', 'Physics', 9)
('David', 'Biology', 9)
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

7.11 Computing Number of Hospital Patients

As the title says, this puzzle offers more practice for you in solving real-life data-science problems. With your experience from the previous puzzles in this book, you should already be familiar with the individual NumPy functions (e.g., `argsort()`) that are used in this puzzle. However, the trick here is to understand the goal being achieved by the collective use of these functions.

Puzzle 42

```
import numpy as np

# Hour-of-day and the corresponding
# number of patients at a hospital
hour_of_day = np.array([10, 9, 12,
                        8, 7, 11, 6])
num_patients = np.array([9, 6, 9,
                         8, 3, 8, 3])

# Sort
indices = np.argsort(hour_of_day)
hour_of_day = hour_of_day[indices]
num_patients = num_patients[indices]

# Cumulative sum
cum_patients = np.cumsum(num_patients)
```

```
nine_index = np.nonzero(
                hour_of_day == 9)[0][0]

print(cum_patients[nine_index])
```

What is the output of this code puzzle (998)?

In this puzzle, we are initially given two arrays: `hour_of_day` and `num_patients`. As mentioned in a comment, the entries in the `num_patients` array represent the number of patients in the corresponding hour entries in the `hour_of_day` array.

Our goal is to determine the total number of patients that have visited the hospital up until 9 o'clock. To do this, we first need to sort the two arrays by the hour-of-day. So, we use `argsort()` to achieve this sorting. Next, for the sorted patients array, we determine the array of cumulative patient counts using NumPy's `cumsum()` function. To display the count of patients up to 9 o'clock, we determine the index of 9 in the sorted hour_of_day array. Corresponding to this index, we display the cumulative number of patients using the print statement, which is 20.

The correct solution »

```
20
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

7.12 Finding Chunks of Allocated Memory

In this puzzle, we try to find the different chunks or patches of *used* memory inside a computer's total space of memory locations. In other words, this problem is the same as finding groups of cars that are parked side-by-side in a parking lot.

Now that you know the problem scenario, try to carefully determine the correct output for this tricky puzzle's code. You are advised to use a pen and paper to write down your thoughts as you go through the different steps of this puzzle.

Puzzle 43

```
import numpy as np

# Memory allocation:
# '1' means allocated, '0' means free
memory = np.array([0, 1, 1, 0, 0,
                    1, 1, 1, 0, 1,0])


# start and end indices of
# contiguous memory allocations
diff = np.diff(memory)
starts = np.nonzero(diff ==  1)[0] + 1
```

```
ends = np.nonzero(diff ==  -1)[0]

sizes = ends - starts

indices = np.argsort(sizes)

for i in range(len(sizes)):
    print((starts[indices[i]],
           ends[indices[i]],
           sizes[indices[i]]))
```

What is the output of this code puzzle (997)?

In this puzzle, we are given a memory map inside a computer. The goal is to print the size of memory allocations along with their start and end indices in the `memory` array. Also, we want to print these chunks of memory in ascending order of their size (smallest first).

To do this, we first exploit the NumPy `diff()` function to determine the boundaries of contiguously free and allocated memory. The trick here is that the difference between consecutive values in the `memory` array will be 0 if both values are equal. However, as soon as a change from allocated to free or free to allocated memory takes place, the `diff()` function will return values of -1 or 1 for those indices, respectively. Using these 1 and -1 values, we determine the start and end of the allocated memory chunks (contiguous '1's).

After determining the starts and ends of the memory allocations, we can simply determine the size of these allocations by subtracting start indices from end indices. Next we use the `argsort()` function on these size values to determine the indices of those allocations in ascending order of their size. Finally, we print these allocations according to this order.

The correct solution »

```
 (9, 9)
(1, 2)
(5, 7)
```

Add this to your Elo rating »

Your Elo	**Correct**	**Incorrect**
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
≥2200	10	-60

Your new Elo rating »

7.13 Giving Meaning to the Mean

In this puzzle, the emphasis is not on learning new NumPy functions. However it is very important since it shows how the NumPy mean function can be used in a more complex algorithm. The algorithm is called Binary Search and is a powerful tool to search items in ordered arrays. It is basic knowledge for programming, so make sure you understand how it works!

Puzzle 44

```python
import numpy as np

# Bookshelf ordered by book titles
bookshelf = ['A', 'B', 'C', 'F', 'X', 'Z']

def binary_book_search(title):
    lower = 0
    upper = len(bookshelf)
    idx = int(np.mean([lower, upper]))

    while not bookshelf[idx] == title:
        if lower == idx or upper == idx:
            idx = -1
            break
        elif title < bookshelf[idx]:
            upper = idx
        else:
            lower = idx
            idx = int(np.mean([lower, upper]))

        return idx

    # Look for book Q and book B in the shelf
    print(binary_book_search('Q'))
    print(binary_book_search('B'))
```

What is the output of this code puzzle (507)?

We use the NumPy `mean` function to build the Binary Search algorithm. Suppose you are looking for a book on a shelf. If the books are not sorted but put on the shelf in a random order, you always have to start at one side of the shelf and check the books one by one in order to find the one you are looking for. If there are 10 books on the shelf, in the worst case you have to check 10 books to find yours; on average it will be 5 checks. If there are 100 books, the worst case would be 100 checks—or in a more general way: if there are n books, in the worst case, if your book is at position n, you have to check n books. This is called linear search and it takes linear time.

A more intelligent and faster approach is possible if the books are sorted, e.g., by title. Now we can start our search in the middle of the shelf. If the book's title in the middle starts with a letter that comes before the first letter of our book's title, we have to look left from the middle, else we have to look to the right. In the next step the position in the middle becomes the new upper or lower position, depending if our book is to the left or the right of the one in the middle. Now we check again the book at the new middle position and move to the left or to the right again. This process continues until we find our book or the upper or lower boundary are equal to the middle, which means that our book is not on the shelf. Can you guess how much faster you are if you keep your books sorted on the shelf?

Now, let's go through the code. The array on which the `binary_book_search` function operates is called `bookshelf` and contains six ordered elements. In this case it's only letters; imagine these are the first letters of the books' titles. First, we start by setting `lower = 0` and `upper = len(bookshelf)`. Thus the `idx` at which we take the first look is the middle of the shelf. In the loop we keep resetting `lower` and `upper` to the last value of `idx` and set `idx` to be the mean value of `lower` and `upper` until we find the book or until we know that it is not on the shelf. When we find the value, we return the index of the position in the array it is at. If `lower` or `upper` becomes equal to `idx`, the book is not on the shelf because there is no other book between those two values. For example `lower = 4, upper = 5, idx = 4` would loop forever since `idx = int(np.mean([4, 5]))` will always set `idx = 4`. Thus we return -1 and exit the loop.

The correct solution »

```
-1 1
```

Add this to your Elo rating »

Your Elo	Correct	Incorrect
0 - 1500	60	-4
1500 - 1600	56	-8
1600 - 1700	52	-12
1700 - 1800	46	-18
1800 - 1900	36	-28
1900 - 2000	28	-36
2000 - 2100	18	-46
2100 - 2200	12	-52
$\geq$2200	10	-60

Your new Elo rating »

— 8 —
Final Remarks

Congratulations, you made it through this whole NumPy book. This proves two things: First, you have now left behind 99% of all Python coders—most of them have never read even a single book about Python (and most of those that did have not read a book about NumPy). And second, you have persistence and dedication to becoming a better coder.

In other words, you have both—technical expertise and the right mindset. Let me tell you that for you, the sky is the limit. Because those two skills will largely determine your value to the marketplace. You can and should start selling your skills fast.

Why should you *start selling your skills*? The reason is simple: selling your skills to clients will force you to leave the ivory tower. The purpose of learning is not learning itself—but leaving a footprint in the real world. You must apply what you have learned or it's like you have never learned it. Even more so, if you don't apply your skills fast, they will start deteriorating again. Use it or lose it!

By reading this book, you have now acquired a rare and precious skill: using the NumPy library to make sense of data. I know that you think you need to learn even more before you start selling your skills. Who would buy them anyway?

The answer is simple: the vast majority of people who have *not* read a textbook about data science in Python. The vast majority of people are not even capable of writing source code to solve problems.

This is your huge advantage: you *can* write source code and you *can* solve problems with your skill. This is your key to bring value to people and companies, and ultimately make a lot of money. A simple formula for success is to choose a problem, and solve it to the best of your abilities. And repeat. There is not much more to it. Stop procrastinating, stop finding excuses, and start making the world a better place.

Your skill level

By now, you should have a fair estimate of your skill level in comparison to others—be sure to check out Table 3.1

again to get the respective rank for your Elo rating. This book is all about pushing you from beginner to intermediate NumPy coding level. In follow-up books, I'll address the advanced level with more difficult puzzles.

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

Where to go from here?

I publish a fresh code puzzle every couple of days on our website `https://finxter.com`. All puzzles are available for free. My goal with Finxter is to make learning to code easy, individualized, and accessible.

- For any feedback, questions, or problems where you are struggling or need help, please send an email to `admin@finxter.com`.
- To grow your Python skills on autopilot, register for the free Python email course at the following

url:
`https://blog.finxter.com/subscribe`.

- If you want to start selling your Python skills as a Python freelancer, join the free *"How to Build Your High-Income Skill Python"* webinar replay at `https://blog.finxter.com/webinar-freelancer/`.

- This is the third book in the *Coffee Break Python* series, which is all about pushing you—in your daily coffee break—to an advanced level in Python's NumPy library. Please find the first book at `http://bit.ly/coffee-break-python`.

Having read this book, you can be confident using NumPy in your everyday life. Please rate the book on Amazon to help others find it.

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

Python Cheat Sheet: NumPy

"A puzzle a day to learn, code, and play" ➜ Visit finxter.com

Name	Description	Example

`a.shape`	The shape attribute of NumPy array a keeps a tuple of integers. Each integer describes the number of elements of the axis.	`a = np.array([[1,2],[1,1],[0,0]])`
		`print(np.shape(a))        # (3, 2)`

`a.ndim`	The ndim attribute is equal to the length of the shape tuple.	`print(np.ndim(a))          # 2`

`*`	The asterisk (star) operator performs the Hadamard product, i.e., multiplies two matrices with equal shape element-wise.	`a = np.array([[2, 0], [0, 2]])`
		`b = np.array([[1, 1], [1, 1]])`
		`print(a*b)          # [[2 0] [0 2]]`

`np.matmul(a,b), a@b`	The standard matrix multiplication operator. Equivalent to the @ operator.	`print(np.matmul(a,b))`
		`# [[2 2] [2 2]]`

`np.arange([start, ]stop, [step, ])`	Creates a new 1D numpy array with evenly spaced values	`print(np.arange(0,10,2))`
		`# [0 2 4 6 8]`

`np.linspace(start, stop, num=50)`	Creates a new 1D numpy array with evenly spread elements within the given interval	`print(np.linspace(0,10,3))`
		`# [ 0.  5. 10.]`

`np.average(a)`	Averages over all the values in the numpy array	`a = np.array([[2, 0], [0, 2]])`
		`print(np.average(a))        # 1.0`

`<slice> = <val>`	Replace the <slice> as selected by the slicing operator with the value <val>.	`a = np.array([0, 1, 0, 0, 0])`
		`a[::2] = 2`
		`print(a)              # [2 1 2 0 2]`

`np.var(a)`	Calculates the variance of a numpy array.	`a = np.array([2, 6])`
		`print(np.var(a))              # 4.0`

`np.std(a)`	Calculates the standard deviation of a numpy array	`print(np.std(a))              # 2.0`

`np.diff(a)`	Calculates the difference between subsequent values in NumPy array a	`fibs = np.array([0, 1, 1, 2, 3, 5])`
		`print(np.diff(fibs, n=1))`
		`# [1 0 1 1 2]`

`np.cumsum(a)`	Calculates the cumulative sum of the elements in NumPy array a.	`print(np.cumsum(np.arange(5)))`
		`# [ 0  1  3  6 10]`

`np.sort(a)`	Creates a new NumPy array with the values from a (ascending).	`a = np.array([10,3,7,1,0])`
		`print(np.sort(a))`
		`# [ 0  1  3  7 10]`

`np.argsort(a)`	Returns the indices of a NumPy array so that the indexed values would be sorted.	`a = np.array([10,3,7,1,0])`
		`print(np.argsort(a))`
		`# [4 3 1 2 0]`

`np.max(a)`	Returns the maximal value of NumPy array a.	`a = np.array([10,3,7,1,0])`
		`print(np.max(a))               # 10`

`np.argmax(a)`	Returns the index of the element with maximal value in the NumPy array a.	`a = np.array([10,3,7,1,0])`
		`print(np.argmax(a))            # 0`

`np.nonzero(a)`	Returns the indices of the nonzero elements in NumPy array a.	`a = np.array([10,3,7,1,0])`
		`print(np.nonzero(a))      # [0 1 2 3]`

finxter