Programming with Gorski: Junior Python Developer

What you should know as a newbie junior dev.

Help with editing: Weronika Sosnowska

Contents

1 Introduction

Grandly and with vigor. Besides this note: if you don't want to delve into the details of Python's history and aren't interested in who the author is, what this book is for, and just want to jump right into action, I'll summarize the next ~50 pages for you: Python is cool, install SublimeText/Vim/Pycharm/VisualStudio Code (whatever in the beginning, seriously) and skip to chapter 4.

1.1 What is this all about?

This book is essentially yet another work on Python, but not only. It's a consolidation of all my technical and soft knowledge that I had when I started as a Junior Developer, or even now, a little further, when I no longer see myself as a budding junior. I will address various issues, from the recruitment process, through contracts, soft skills, description of my development, useful things, and the programming itself - the proportion will be roughly half and half. Apart from the text itself, I will also provide references and materials for self-study.

I don't think it's particularly groundbreaking or innovative, but on the other hand, it's a bit like that. Why? What sets this book apart? The fact that while writing it, I had one goal - to make it as easy to understand as possible, yet thoroughly analyze the topics discussed in an interesting way and the fact that this will not be just a book about Python, no.

It should be a complete introduction to a wide range of issues related to programming and the work of a programmer, acquiring it, the tools used in it, but also learning for you about the most important skill for a programmer - the ability to search for and process information, because that's what computer science is about.

In short, it's an introduction to a complete set of skills, traits, tools, mastering which will make you ready to some degree, male or female, to start your adventure as a programmer. But don't think that you will read one book and bang, you're working. Oh no. This is the first step on a very long road.

I don't want this to be another ordinary Python textbook that only covers Python, there are already many of them. I want you, my reader, male or female, to know a little more than before after wading through this book, to have a clear picture, however general, of what a programmer's work looks like, what this world looks like, whether it's worth entering it, what are the pros and cons of this job, what specific skills are needed, where to learn them, and so on.

My dream is for you to be like a curious child after reading this book, shown a new toy - full of interest and resolve to play with it. Whether for a moment or perhaps for longer, it doesn't matter.

I just hope that this book will somehow positively influence your knowledge, or perhaps your life, allowing you to take, for example, the first step in changing your career or starting it. Simply, I hope to be of some help. That's its purpose. It stems from the fact that my biggest dream, my personal sense of life, is to impact as many people as possible, to bring something positive to their life.

Many people also helped me, so I want to pass the baton on, that's my intention.

I hope that I will somehow manage to achieve it. If so, remember, I'll be glad to hear your story, or even publish it on the blog – grski.pl – maybe thanks to it you will then help others? So if this book in any way expands your knowledge, or gives you something useful, reach out to me!

olafgorski@pm.me, @grskii on wykop, grski on 4programmers, grski on github

When I was giving tutoring, long before I became a programmer, I learned that first, teaching others, you also teach yourself, two, dry facts are not always enough for someone to understand a given issue. Interesting examples, unconventional analogies, and exercises are needed. That's how I'll try to convey knowledge in this book, straightforwardly, yet interestingly.

I am therefore directing it to individuals who are rather beginners, who want to slowly and thoroughly understand certain concepts related both directly to Python and to computer science in general, gain a little technical savviness, but I believe that an experienced person will also find some titbits here that may be of interest. This book will, rather, be full of simplifications that I will make to make it easier to understand, so I apologize in advance for all inaccuracies, but it's necessary. Sometimes when reading Polish scientific, technical books, it seems to me that their authors fear that if they write something straight, understandable, people will consider them fools, or something. As a result, we get texts that later make people wonder how deep the author of such a work sat down on a stick.

This book is also a summary or maybe a small summary of what is worth knowing when we start for example to the first job. So if you are looking for a short guide to Python's key functions, well, this book is probably not for you. If you are a beginner programmer and you want to broaden your horizons or get to know the basics of Python more broadly, I invite you further. I will try to put answers to all the questions that I had at the beginning of my journey here.

This text is also the result of the fact that I asked myself one question. Which one? You know what I mean? It was asked in the context of my knowledge of Python, as well as programming in general, and despite the fact that I was already working as a programmer in that language - I was supposedly a full-fledged junior, unfortunately, I had to answer... No.

I suspect that in the core of things there are many such people, despite everything. We often do not realize how shallow our knowledge is, despite the fact that we are managing at work.

Especially if we take into account old, mature and developed languages like Python, Java, or C++. Even the creator of the latter, is afraid to say that he knows it better than 8/10. This says something. Anyway. After I answered this question to myself and realized that my understanding of Python is shallow, I decided to change this, to take it to the next level.

I always had a desire for progress inside me, which I neglected for some time, which ended up in stagnation or even regression. It couldn't be like this any longer, so I fixed it.

And once I did, I decided to write down my thoughts, knowledge, and experiences right here, but that's not the only reason. To write this book, I will have to learn quite a few things to present everything reliably and sensibly, because I would be ashamed to give you, dear reader, a junk book made offhandedly.

I don't know how many pages this book will eventually count, but I am sure of one thing, it will not be short. Mentally, I have an initial image that tells me I won't go below 250 pages. Especially considering the fact that once I start talking, it's sometimes hard to stop because one topic leads to the next and so on and on.

One more note – my writing style is specific. As you can see. It just is. I have my own manner, I have been reminded of that by various people many times and I am aware of that, however, I consider it as part of the specifics of my works.

I'm not going to change that. Just like the fact that in this book there may appear both weird and lofty words as well as those simple, colloquial ones. I'm just like that. If colloquial language, an infantile style scares you, I advise you to quickly put this book away. I'm not saying this from an arrogant position but from a well-advised one.Sometimes there may be some distasteful jokes, dark humor, and personal comments, a description of my experiences, not necessarily politically correct.

That's probably it when it comes to my short foreword, or something along those lines. All in all, I don't really know what all this should look like - I am a simple man who just wants to share with you a fragment of, perhaps, useful knowledge. Only that much and as much.

1.2 In short

... speaking, this book will also be a guide and a reference to other materials, which you have to process on your own. I only discussed more deeply and thoughtfully, in my opinion, those topics that are often neglected.

1.3 Acknowledgements

At the very beginning of my book, I will reserve for myself, and by the way for you too, a bit of humility - it might seem that these words, because this is after all a book, and often the books are written by serious people, are written by some successful man, a great programmer, a know-it-all.

Nothing could be more wrong, in itself, I am a slight and simple little man, who has come here, where it is, largely thanks to the help of many people, some of whom are listed below and I would like to thank all of them.

If you're not interested in this, like the odd dedication that is here, skip a few pages.

I address some directly, others in the third person, differently. For no reason. So I thank...

...my dearest and only Grandma. For everything, for the person I am. For love. For sacrifices.

...you, **Wiktoria Chmielewska**, for the two happiest trips in my life, for your dedication, for your care, and friendship, for saving my skin and not giving up when others had given up a long time ago.

. . . you, **Łukasz and Batu**. Cool that you exist. I won't write much, God is love and a high five to the foreman.

. . . you, **Rafał**, along with the family, whom I won't mention here for your privacy - your family, of which I essentially feel a part of, for surrounding me with care, teaching me many things, and a real, big influence on me and my life. I don't have a father, but if someone asked me who is the closest to him, I wouldn't think for a long time.

. . . you, **Dearest**. C R A B B I N G.

. . . **Jakub Gąsiorski**, a person who had an interview with me for the first job. I think I owe it to him that eventually the decision was made to hire me, because he decided to give me a chance then and later, many times. Here also thanks to **Maciej Mondrzycki**, CEO YumaSoft, because the final decision whether to invest in me belongs to him.

. . . **Sebastian Opałczyński**, as a person who I consider my first mentor, a model of how to grow as a programmer, and a person who to a large extent contributed to the creation of this book. He lifted me above what I could achieve myself and I owe him a lot, really. Thanks Łysy. Sometimes he pointed me in the right direction in a few things, patiently answered various questions, I also fondly remember our various conversations. You are a very good teacher and a man. Have you ever wondered what alpha male is?

. . . **Piotr Nietrzeba** for being a human being when necessary, scrapper on other occasions, for showing certain imperfections and that they can be overcome, or at least you can try, for keeping the promise.

. . . **my school director Pawel Boryczka** from the School Complex in Kozienice, for caring for me for 3 years and helping me a lot in my life. Because of this, I was able to develop certain skills, meet different people, who I probably wouldn't have discovered in other circumstances. I will remember this.

..you, **Emilio**. You know why. Eryk sends greetings to Blanca from the beyond, he has already faded into oblivion.

..you, **Falo**, you beloved old f***. For being a warm lovable dumpling, who will deduct and won't have anything, but will give it to someone. You're a wonderful person, don't change, if the old folks you know from, say "you're too good for this place", then know that something's happening.

. . . you, **Łukasz and Ola**. I love like family, send warm thoughts to you and children.

. . . you, **Alina**. You taught me a lot, gave me a lot, you made my life better, just like that. I had a wonderful summer with you.

. . . you, **Kinga**, that you are and were.

. . . you, **Zimmerman.** for showing understanding when I needed it.

. . . you **Paweł Poldec**, for taking me under your roof in difficult times. I remember.

2 About the author

Since I've already dragged you through so much of my ramblings, dear Readers, perhaps it is time to say a few who words about who I am and what wisdom, if any, I have to share?

Well, I am no one more than a regular guy named Olaf.

2.1 Górski, or a land-dwelling wise creature

I am just a guy who doesn't hold a high school diploma, started working right after school as a programmer for quite small money, but still. A year and a half later, when I had already gained considerable experience and was working in my fourth company, I reached the supposedly mythical level of 15k, working remotely for a foreign client. Fulfillment of dreams. Supposedly. Plus, I must admit I had more luck than brains. Three and a half years later, we are here. I have about five years of experience. I work as a Tech Lead, managing several people. I had the adventure of running my own company, at its peak employing about 8 full-time people. Contract amounts like 40k per month or a bonus in the form of shares for half a million are now commonplace for me. Funny thing, life. Not long ago 3k seemed like a lot.

The journey has been interesting. Before all of this, while still in middle and high school, I took on a variety of jobs. Here some finishing, construction work, handing out flyers, welding, tutoring, selling glue, anything. Anything that brought in money. And perhaps you now find it surprising that an old laborer like me is creating books, programming. Why am I even talking about this? It's just that I am a rather authentic person, I want my readers to know who I am.

Halfway through high school, I started creating simple websites on commission, until I turned eighteen. A month later, I managed to find a regular job in IT. I went to school part-time and worked remotely as a junior python. Then just a quick refusal to take the high school exams as a protest and a fight against the system, a few company changes and voila, 15 thousand, I would like to say euros, but unfortunately not yet. 15k euro was there, but when I ran my own company. Not anymore, but maybe another time. Now I am a good full-time employee.

2.2 Currently

I work as a Lead, I code in Python, Django, Architecture. I manage JIRA from time to time, some bits and pieces of Product Owner duties, managing or mentoring members of my team, and that's pretty much it. There's no bunker, but it's still awesome.

I run a blog grski.pl in my spare time, I go to the gym, write, survive, sober up, struggle with demons. Life is cool, but not always colorful.

I don't know, I feel like I'm just a pretty normal guy. There's a bit of cringe here and there, but hey, that's just me.

Among the successes in my life, under the slogans `tinder statistics` and `python advantages`, I rank pretty high in Google. Sometimes even in the first place. I used to always be first, but I forgot to renew my domain and had to restart the indexing xD

I have come a long way from a foolish lad who did not show up for his own high school exam, arriving in Warsaw with just 400 zł in his wallet, to Mr. Programmer, who's still a little foolish, but with a richer mind at least. Now I don't worry if there is something to put in the pot tomorrow, or calculate calories in milk, which has a better calorie-to-price indicator to fit in within the day's calorie needs.

I think that if I could make it, many of you could too. Not everyone. Most will probably give up along the way.

2.3 It won't be easy

Those who don't give up are likely to become at best average code scribblers. A handful of the determined will achieve cool things. I don't say this looking down on you. It's just that I know what it looks like. It's not an easy piece of bread. Do you think that for the first year and a half, I worked only eight hours, on some light project, and then went home? I did as much overtime as I could, gaining experience from senior workers. I have even slept at the office because it wasn't worth it to go home. This kind of thing isn't normal, but neither are the results that I've achieved. However, I generally wouldn't recommend it and I'm saying this not to encourage you to do the same but to make you realize that extraordinary results often require extraordinary efforts. Though from another perspective, it wasn't really an effort for me, I rather enjoyed it. Still, something had to be sacrificed - free time in this case.

I often used to sit and code for 12 hours, on Fridays, Saturdays, Sundays. Luckily, I found the perfect mentor, a golden person because otherwise I don't know. I've waded through shit, in legacy projects that I wouldn't wish on anyone, and above all, I was extremely lucky. Don't expect it to look the same for you. It might, but it probably won't.

In my free time, I try to help others learn to code, leading different workshops, projects, or mentoring, usually absolutely free. From experience, I know that most people simply give up. By the way, speaking of the people who I helped take their first steps on the programming path, so far there have been about five who, starting from scratch, persisted, and got their first job. The shortest time it took was just under half a year, the longest about two years. Everything depends.

2.4 So in conclusion

At the end of the day, I only want this book, the knowledge it contains, the dose of specific and sometimes offensive humor, and the biased and infantile style, to help someone in some way. Perhaps I will succeed.

Plus, it's a result of realizing that after a certain threshold, money loses its importance and more important things come to the surface. The need for some sense, for doing something higher than earning another thousand on the paycheck. Ok, enough of my babbling.

You can find my precise income history and such details in another chapter, somewhere at the end of the book. If not my words and my personality, then let at least they convince you that perhaps it's worth reading this book.

3 Language Choice

In this book, we will primarily work with Python. Why? I've listed the reasons for choosing this language in another chapter, but I'd like to mention one other language that every programmer must know and should start with if they're 100% certain they want to pursue a career in IT. What language is that?

3.1 The Essential Language

Interesting, what language could it be! C, Python? Maybe C++ or JavaScript? Not at all! I'm talking about... the English language.

We live in times of globalization. Everything is becoming somewhat standardized. The same TV shows, the same movies, the same clothes, brands, and... language. The English language has long been present in our native language - Polish. Anglicisms have existed for many, many years. However, in recent years, during the existence and development of global civilization, the number of anglicisms and borrowings appearing in our speech, especially among young people, is growing exponentially. And not without reason. And I say this as someone writing a book in Polish. You can manage without it if you insist, yes, but if you're serious about a career... No. Seriously.

English is the absolute hegemon when it comes to international communication or communication on the Internet. And we, as people closely related to this world, are somewhat forced to know it well. Practically in every company, English is the standard, and to be a good programmer, we need to know it well enough to communicate freely and read documentation. That's it on this topic. This implies that we need to know it well. Technical documentation is usually not that simple, contains specialized vocabulary, and has a difficult, formal style.

English is probably even more important at the beginning than the programming language we choose, which is why, before we start coding, it's worth improving in this area. What are the advantages of knowing this language?

3.1.1 Materials

The matter is quite clear. The most high-quality materials are available in English. This book is an attempt to slightly change this fact and open the way for those who, despite everything, don't speak English, don't want to speak it, and maybe aren't thinking about a career in IT. Nevertheless, currently...

Want to learn more? Best from English materials. Moreover, sometimes English materials are the only ones available. Almost every framework or library I know has documentation. In English. In Polish, there's usually none, or if there is, the translation leaves much to be desired or is fragmentary.

That's right, translation. Certain terms in our world become quite strange and incomprehensible when translated. Can you imagine working daily with an "Integrated Programming Environment" (ZŚP) instead of an IDE? A "bug remover" instead of a debugger? A "kernel panic" instead of... well, kernel panic. For us, experienced programmers, this is unimaginable. No one outside your country would understand you.

Instead of pulls and commits, we'd have "withdrawals" and "confirmations" (XD). Why have inits when we could have "starters"? Examples could be multiplied.

And documentation is fundamental, by the way. Whether when creating a serious project or when using one, you'll not only read it but also create it. How will you encourage other developers to work on your open-source project if they can't understand it because the source code and documentation are in Polish? In this way, you severely limit your horizons.

3.1.2 Communication

Again, a key thing when creating projects. In most companies today, you collaborate with foreign clients or contractors from overseas. Many things can be streamlined and made easier when we can talk directly with someone and clarify certain things. Often, English is essential for this. Especially when someone who doesn't speak Polish joins your team.

This isn't as rare as it might seem, quite the opposite.

3.1.3 Programming Itself

Learning programming itself is easier if we know English. Practically every programming language has keywords in English.

When we're just learning, this is quite helpful because even without seeing what a given keyword does or what a given function does, as long as they're properly named (and current good code standards generally require this), we can practically guess what the code does without any knowledge, without looking at the source, by inferring from the translation of the given fragment, the name of the function/class.

3.1.4 Exceptions

I know, someone will immediately say, 'but how, I've been in the profession for five years and somehow manage without English!' Well, yes, but for how long and in what company? Want to work in Januszsoft? Be my guest, but I won't pay for the chemotherapy you'll need after experiencing the scale of stupidity and onion-like behavior that takes place while working for a typical Janusz. And believe me, that's how it will be because people who completely don't know English aren't really hired in other places.

What I wrote above is rather obvious to everyone somewhat familiar with the matter, but some people, at the beginning (including me), are quite stubborn about learning English.

Because I want to learn programming!!! Great, but to do it as efficiently as possible, English is useful, and then it's an unparalleled help in finding and doing the work itself.

That's about it on this issue. Although the question remains - how to learn?

3.2 How to Get Started?

Let me explain.

3.2.1 Well, How?

Let's start with how not to learn. A simple and perfect example of this is seen in our schools. How English is taught there is a joke. A sad one, but a joke.

The focus there is solely on the middle school exam, then on the high school graduation exam. Nothing else is thought about in 90% of cases. And what's on the exam and graduation? Not necessarily things used in everyday situations.

In high school, at least in my case, in the first year, a student has 5 hours of English, if they choose the advanced level in this language (rarely does someone not do this), then from the second year, they have 7 hours. Quite a lot, right? Yes, but how this time is used is a tragedy.

What does an average lesson look like? You get a photocopy with grammatical structures that you fill in. Sometimes you write/say something. Thank you. Focus on grammar and... Nothing else. It can't be like this. I don't want to get into a discussion about how bad the current education system is, what needs to be changed in it, and so on. Of course, there are probably teachers who stand out from the norm and try, because they haven't been eaten by the concrete yet, but how many of them are there... Let's move on to the issue of...

3.2.2 How Should It Look Then, Mr. Know-It-All?

I'll tell you. First of all, you can't focus all the time on something completely unnecessary for the most part - grammar.

Let's look at small children. Let's say from grades 1-2 of elementary school. Can they speak in their language? Yes, they can. Usually even relatively correctly. And let's ask this child what they know about parts of speech and sentences. We'll probably hear as much as a deaf person.

So the child doesn't really know grammar formally, but uses the language correctly. Interesting.

We should take an example from this and primarily remember not to focus on learning grammar during language learning. Of course, there are tiny exceptions to this rule, but they rather confirm the fact.

We should focus on using the language, not theorizing. Only this way will we learn it most effectively. Let me give an example. In English school, children usually start learning from kindergarten. It was no different with me - in so-called "zerówka" (pre-school), the first slow lessons began. In any case... 7 years later, my English still left much to be desired.

Fortunately, my approach to learning this language changed from forced to more practical. Textbooks faded into oblivion, writing and reading in this language came, communicating in it.

Suddenly, in two months, I made greater progress than in the previous 7 years, amazingly surprising myself. Grammar should result from practice. After some time, it comes naturally, and that we don't know the theory behind it...

Well. And do we know it in Polish? At least I don't. Only those complete basics that a child learns in elementary school. As for the rest... Hah. Maybe I'm a brainlet with 30 IQ, and everyone else knows, but I doubt it.

3.3 Implementing This in Practice

In my opinion, several methods can be distinguished that should be applied.

3.3.1 Translation

Let's start with the thing I like most and have the best opinion about. It's translation. What translation? Normal - you take a text in English and chop it into Polish.

Don't worry if you don't know the words. I didn't know them at all at the beginning. About 80% of words I had to check in a dictionary. But don't jump to it right away - try to infer something from the context first.

You can translate anything. I particularly recommend comics/manga/movie subtitles/some pictures. Why?

Because you see what's happening on the scene and can infer more things from the context. Thanks to this, you'll learn faster. Besides, you can also check your translation with translations by other people - for example, take some cartoon in English, translate a fragment and compare it with the Polish version, translated by a professional.

This way, you'll learn very quickly. Moreover - you'll start picking up certain phrases characteristic of this language and learn that we don't translate everything literally, that we speak somewhat differently in English because certain words have a different weight, structure, and meaning there than in our language, and literal translation is pointless. I recommend it. Additionally, you'll also learn to 'think' in English, which is essential at a certain, somewhat higher level.

I, for example, quite often have it that I want to express a certain thought, but I lack Polish equivalents for thoughts that I can express without problems in English because such a phrase exists there.

3.3.2 Learning Words/Phrases

I'll write a bit about this too. It's nice to have a rich vocabulary in a given language, but we don't need too many words at the beginning. Do you know how many exactly are enough?

Just 25. Yes, 25 words. That's enough for the very beginning. Because according to research, these 25 words account for 33% of the content spoken by an average English speaker. So knowing 25 words, you can understand one-third of what a foreigner is saying to you.

Add to this understanding certain things from context or similarity in word sounds, and it turns out you can almost communicate. Your hands would probably hurt a bit from the conversation, but I think it would be possible. But 33% isn't such a satisfying result. Let's move on then - to understand 89%, 1000 words are enough. Significantly more, but still little.

Knowing 3000 words, we understand 95%. So basically, as if we understood in Polish. Because that 5% is, I suspect, made up of specialized words used in various strange professions. After all, it's the same in Polish - not everyone needs to know what a square, anchor, or trowel is. That is, most people who have worked on a construction site at some point, like this humble author of this book, will associate, but the rest? Not necessarily.

Similarly, in English, there are certain specialized words that are rarely useful and can be learned if needed.

So our goal here is to reach 1000 words. That's enough in my opinion for the beginning. We aim for at least 30 words a day. After a month, we're practically home. And how to learn them? Because just sitting and reading is pointless. I offer two great tools:

Memrise Duolingo

Both these platforms are available for free, offering apps for Android or iOS.

I think there might even be a version for, horror of horrors, Windows Phone somewhere.

These two great platforms will help you learn English and other languages without any problems. Which one to choose, how do they differ?

Both help in learning, but Memrise is more diverse and dependent on content created by users. You can learn not only vocabulary there but also things like geography, history, literature, and many, many others. It works great for learning vocabulary.

What about Duolingo? Here we have a somewhat more systematic approach. Courses are prepared by a professional team (Memrise has these too, but not as many) and cover both vocabulary and grammar basics.

Nevertheless, both services are similar, teaching words, pronunciation, grammar. The choice is yours, although I admit that I use both.

Or rather used, because now I'm a lazy old man who doesn't want to learn new languages, arguing to myself that in a few years we'll have AI for translation.

3.3.3 Listening

This probably goes without saying. I recommend movies and cartoons with subtitles, but English ones. Forget about Polish ones. If you don't understand something, pause, turn on subtitles and translate slowly. Definitely. Although actually, I don't know if I should recommend learning listening - once you start understanding a bit, it turns out that 90% of your favorite English-language songs lose their meaning. That sucks.

3.3.4 Speaking

Talk to yourself, talk to family, friends, trees, dogs, microphones. There are special groups where you can arrange to talk with someone who's also learning or who will teach you in exchange for teaching Polish.

You can write to me and arrange a chat on Skype or TS, I'll be happy to help. You'll find contact on the blog - email, GitLab, GitHub, or on the 4programmers forum. Heck, we can even create a small group helping with language learning. I'm up for it, what do you say? And remember one thing - don't be afraid.

No one cares about your accent and pronunciation. Being in London or abroad in general, I've heard so many different versions of English that it's mind-boggling. Often even those for whom English is a national language can't understand each other due to accents and slang.

It's like talking with a Silesian. Only three times worse and besides the Silesian, you have 15 other groups. So relax. They'll understand you, no one will judge you, but rather encourage and appreciate the effort you put into trying to learn the language. Seriously.

3.3.5 Reading

Consume content in English and that's it. Don't worry about the level of understanding at the beginning, relax. It will grow. You'll find a list of interesting blogs on programming topics at the end of this book, check the table of contents.

3.3.6 DeepL

A good exercise is writing sentences in Polish, let's say 10, then translating these entire sentences using DeepL. Simple sentences are enough. DeepL is something like Google Translate. Then we reverse the process. We write 10 other sentences ourselves and check correctness in DeepL.

3.4 To Sum Up

Following these tips and dedicating about 2 hours a day, after a month you'll understand about 70-80% of everyday conversations in a foreign language quite fluently!

Of course, we're talking about simple conversations conducted with a normal accent, but still. The specialized vocabulary you'll need while learning programming, you'll pick up along the way.

Remember especially these two links to Memrise and Duolingo and that you should translate, read, listen, and speak as much as possible.

Besides, I strongly encourage you to check out Radek Kotarski's book "Hack Your Brain" - it's a collection of interesting techniques and tips on how to learn to really learn, to remember quickly and effectively.

No, Kotarski didn't slip me any cash for advertising. A shame, I'm always happy to hug some shekels.

In any case. Let's finally move on to describing the language we'll mainly deal with here.

4 Python - What's It All About?

4.1 Python - What's It All About?

Before we dive into learning Python, let me tell you a bit about the history of the language, where it comes from, its goals and assumptions, how it evolved and changed over the years, and what it looks like today, where it's used and why.

4.2 How to Use This Book

Hold on a moment. I'd like to tell you how I think you should use this book. First, I tried to write it in a way that naturally progresses from one topic to the next - from simple topics to more complex ones. Thanks to its linear structure, it should be easy to understand.

However, if you're already a more experienced programmer, or if you just want to refresh certain concepts, feel free to jump around - most chapters are relatively self-contained components.

Another thing I want to mention, or rather repeat, is that in this book you won't find information exclusively about Python. Beyond information about the language itself, I'll also try to introduce you to certain concepts from computer science in general, so you'll know a bit more, understand how things work, and why they work that way.

I believe this is essential knowledge to become a good programmer who grows and goes far. Such chapters will be somewhat more theoretical in nature, but that doesn't mean they're boring - quite the opposite, in my opinion.

If you want to properly absorb the information contained here and really learn something new, I strongly advise: first, take notes - short, concise, and simple ones.

Second - type out the code. Don't use the Copy-Paste method. Type it out yourself, period.

Finally, number three - independently complete the exercises that I'll place at the end of chapters, but that's not all - experiment with the code. Change it, see what effects these changes have. Experience it in practice, analyze your modifications, think about them and their results, how they affect the program's operation. This is the best method of learning. This is the basic assumption that guided me while writing - that you will independently complete exercises, read additional articles, books, and conduct experiments with code.

You can read more about how one should learn and what methods work in Gynvael Coldwind's blog post - Beginner Programmer's Guide. Grski recommends it. In fact, I recommend the entire blog. There are few places on the Internet where you can find such good and interesting content. Google it, because it's worth it.

Additionally, I want to point out that I won't write about every tiny detail in this book. This isn't a Python encyclopedia; the assumption here is that you'll look up the details of various things yourself. I want to draw attention to often overlooked things. **Therefore, I recommend that alongside this book, you read materials from Django documentation, Python documentation, the excellent book Learning Python, 5th edition, and the CS50 course**. I more want to make you aware of certain things and explain some concepts, additionally dealing with this often overlooked practical knowledge, an honest account of my experiences.

4.3 Interactive Part

On my GitHub, you'll find the repository `junior-python-exercises - https://github.com/grski/j`
If you'd like your exercise solutions and answers to be checked, fork this repo (you'll learn what that means later), then in the appropriate chapter folder add a folder with your GitHub nickname and put your answers inside. I'll do a code review/look through these answers and give some feedback :)

4.4 Python 2 - Python 3?

Python currently exists mainly in two versions - Python 2 and Python 3. These are two 'main' releases of the same language, however version 3 is newer and introduces certain new things that are not backward compatible with version 2, hence the version number jump.

The introduction of Python 3 happened many years ago, and currently, we're in times when Python 2 is no longer being developed. It's dead and that's it. The only projects in it are some heavy legacy ones. Besides that, somewhere there are slowly some whispers about Python 4.

What does this mean from your perspective as a beginner? Nothing. Just know that you're currently learning Python 3 and that's it. It's still the same language, but there are some differences between it and Python 2 that you can easily learn in a few moments if needed. I choose the newest version to present you with the freshest information, and honestly, Python 2 is slowly becoming a relic of the past, and whoever creates new software in it now is making a mistake, although I don't think such cases can still be found anywhere, except for a few corporate exceptions.

If anything, Python 2 is now only used to maintain old applications written in it, and believe me, more often than not, you don't want to work on maintaining old behemoths. Unless they pay you a lot of money. A lot, lot of money. Even more than a lot. Seriously. Although it's still not worth it. What good is money when you literally change professions from programmer to sewage diver, diving in the programming excrement of some people who hated their lives more than usual, so they created a monster?

Python 2 is passing into history and that's good, nevertheless I'll occasionally mention the differences, basically as a matter of interest.

4.5 A Brief History of Python's Long Journey

Python is quite an old language, so to speak. Older than me, although that's no achievement actually.

Currently, there are plenty of new languages in the market - children like Scala, Dart, Elm, Elixir, Kotlin, and many, many others. Compared to them, Python is an old-timer, as it appeared at the beginning of 1991 at CWI - the Center for Mathematics and Computer Science in Amsterdam. While it's not as ancient as C from 1972, it's definitely a full-fledged middle-aged language.

Its main creator was Guido van Rossum, who until recently had the nickname "Benevolent Dictator for Life" (or rather had, as we'll discuss shortly) and is generally considered the highest authority in the Python world.

Speaking of unusual facts, if you manage to find Guido's email address somewhere on the web (it's not difficult) and send him an interesting email with a question or anything, there's a good chance he'll respond. At least that's what many other Python users have experienced.

The language name itself doesn't come, as one might think, from the snake species, but from a TV series broadcast in the seventies by BBC - "Monty Python's Flying Circus", of which Guido was a fan and thought that naming his programming language 'Python' would be something. And well, it is. Just this alone should be enough to encourage you to use this language.

Python began its public life in version 0.9.0, but didn't stay in it for long, with new versions coming out quickly. This involved various changes, as van Rossum himself, as well as the most important team members, changed their 'home' many times, moving from one organization to another. Code is code, but you have to live on something.

Everything settled down with the arrival of version 2.1, which was released under the Python Software Foundation - a non-profit foundation that operates to this day and is modeled after the Apache Software Foundation.

Python was intended to be the successor to the ABC language - another prehistoric creation that we won't discuss much, despite its evident influence on Python.

Besides ABC, Python shows clear influences or elements borrowed from languages such as: C, C++, Haskell, Java, Perl, and Lisp. Should this mean anything to you? Definitely not if you're a beginner. Just know that Python has some elements in common with other languages and isn't some strange oddity.

4.6 Guido's Abdication

Around the time of writing this book, right at the beginning (well, I can say I took long breaks. . .), something unprecedented happened: Guido van Rossum, Python's creator, decided to distance himself from the Python decision chain and drop his BDFL title, slowly retiring altogether. This was all caused by PEP 572, which was proposed by Guido himself, among others, and which triggered quite unfavorable reactions from the community. What was it about?

About the := operator and assignment in expressions. A large part of the community started criticizing this idea very loudly and strongly, often without any basis, as, at least to me, the PEP itself seems rather well-thought-out and nice, this functionality will definitely be useful somewhere in Python. We'll talk about this PEP later, so no details for now.

Below I include the text of the email published by Guido, translated by me:

"Now that PEP 572 is done, I don't ever want to have to fight so hard for a PEP and find that so many people despise my decisions.

I would like to remove myself entirely from the decision process. I'll still be there for a while as an ordinary core dev, and I'll still be available to mentor people - possibly more available. But I'm basically giving myself a permanent vacation from being BDFL, and you all will be on your own.

After all, this is what you all want. And it even fits the natural pattern of things - Guido's retirement has been anticipated for more than a decade, and there's no reason to believe that I'm going to get smarter (or less stubborn) as I age.

I am not going to appoint a successor.

So what are you all going to do? Create a democracy? Anarchy? Dictatorship? A federation?

I'm not worried about the day to day decisions in the issue tracker or on GitHub. Very rarely I get asked for an opinion, and usually it's not actually important. I'm happy to leave that to the core devs.

The big issue is how are PEPs going to get decided. We may need a voting process. There might be a bunch of small decisions to make, like: - establish the voting process - establish the criteria for voting (e.g. who gets to vote) - establish criteria for accepting/declining PEPs - decide how to select new core developers - decide how to kick people out - establish a CoC (if it's not there already) - establish a ban process

We may be able to write up processes for these things as PEPs (maybe those get accepted by a different process).

Note that there's still the CoC - if you don't like that document your only option might be to leave this group.

I'll still be here, but I'm trying to let you all figure something out for yourselves. I'm tired, and need a very long break."

What more can be said here.

It was a sad moment in Python's history, especially since Guido is Python and Python is Guido.

That's it. Shocking news.

Well, may you fare well, Mr. Guido.

While I don't fully sympathize with him due to various strongly politically correct behaviors, bordering on paranoid I'd say, but still. He did a great job and deserves respect for that.

But the question is how will the Python world look without this man? What direction will it choose? It's an opportunity for growth, but also a threat that we might lose the direction that is currently quite nice.

Anyway... A new wind will blow in the sails, where will it lead us? Time will tell. In a year, two, five. We'll decide about it, the community that creates Python, which means soon you too, dear reader.

Note: from the perspective of time, it's still good. Guido's departure hasn't changed Python for the worse.

4.7 Python's Goals

With Python, the situation is that it's really worth using. Personally, I believe it's one of the best languages for learning programming basics, which was its intended purpose. It allows you to quickly move on to understanding certain programming concepts because the student doesn't have to focus too much on mastering complicated syntax or complex expressions, as sometimes happens in other languages.

Python is concise and simple - most code can be very easily understood by anyone who speaks even a little English or has a dictionary handy, and the code itself is usually short and elegant.

Only a few symbols need explanation, used to shorten notation. Beyond that, Python reads very much like sentences in English. This is also the first snake goal - simplicity over complexity.

Python was meant to be simple, pleasant, and elegant. I believe it absolutely is.

Another goal is portability. Python can run on almost all currently popular platforms - Windows, almost any Linux, macOS. Until version 3.7, released on 27.06.2018, something as strange as FreeBSD version <= 9 was even officially supported.

Thanks to this, Python can run almost anywhere, as long as the hardware resources allow it.

The next goal that Python had was openness. Some languages are quite 'hermetic' - they can only be used after paying for a license or only under specific conditions, purposes.

Python doesn't have this problem - it's completely free to use, modify, distribute, and whatever else the user wishes.

Additionally, Python is open-source, and its development is community-driven. What does this mean in practice? Everyone has access to the language sources for one thing, and two, if you don't like something in Python, you think something could be done better, then...

No problem. Take that function and just add it, change it. If the community finds your changes justified and useful, they'll end up in the language itself. Therefore, everyone can have a real impact on what Python looks like, how it works. Great thing.

These are just a few of the ideas behind Python; I won't describe all of them here, but I think I've managed to cover the most important ones.

4.8 The Snake Sheds Its Skin from Time to Time

What do I mean? About the fact that Python and its applications are constantly changing. Like a snake, it sheds its old skin and gains a new one.

Originally, it was a language that was used rather as a scripting language. Automation of certain processes on servers, some operations on files, text, and so on. Boring things in general. Python didn't take the market by storm; it took some time. At the very beginning, it was rather niche. With time and evolution of the language itself, its advantages and beauty began to be widely recognized.

That's why over the years, Python became increasingly popular when it comes to developing backend parts of web applications, and as we know, these have been experiencing constant growth since the beginning of the last millennium, really. The matter was facilitated by the appearance of various Python frameworks on the market, designed specifically for creating web applications, such as Flask, Pyramid, Pylons, Web2py, and finally Django, which was a real game-changer. Here's an interesting fact. When you browse Instagram, do you know what it runs on? Well, on Django precisely.

Thanks to this, Python became quite popular as a language used for writing web applications, but that's not all. In recent years, we can observe a continuous increase in demand for various specialists related to Data Science, Artificial Intelligence, Machine Learning, or Neural Networks.

All these and related industries are developing incredibly, and the language that essentially rules there is Python. It has a competitor in the form of R, and a rapidly growing contender in the form of Julia, but still, our snake holds strong.

Why? Look at the ideas behind Python - it explains itself. An analyst doesn't have to be a good coder; their task is to process data, so they need a language that will quickly allow them, without unnecessarily delving into syntax or how the language itself works, to translate their thoughts into code.

Python is perfect for this due to its simplicity and versatility. I can already imagine such an analyst sitting and checking whether they've definitely freed all the previously allocated memory in their super beautiful code written in C or C++. There's just no such option. And that's good.

True, this creates a certain problem in the form of performance, but more on that later, because it's something that can be overcome and solved much more easily than trying to teach everyone C/C++.

Of course, Python is still used in various scripts, automation, and so on, nevertheless I believe that this is no longer its main application, as it was years, years ago.

So, as you can see, Python is developing and appearing in an increasing number of projects, fields, and areas related to broadly understood computer science. Personally, I believe that this trend will rather continue, just as it has so far, and Python will gain more and more popularity year by year, but let's get into the details - why?

4.9 Python's Advantages

4.9.1 Expressiveness

Python is very expressive. What does this mean? Well, in Python, you can achieve with relatively little code what in other languages might take several times as much. To not be unfounded, let's look at an example of the classic program that begins programming learning - Hello World, or in Polish, Witaj Świecie. Which is actually ironic, because when you start programming, you should rather say goodbye to the world, because you won't be seeing much of it anymore from your basement. In Python it looks like this:

```
print('Hello World')
```

Pretty simple and understandable, right? One line and done. Let's look at other languages though. Let's start with Java:

```
public class HelloWorld{
    public static void main(void) {
        System.out.println("Hello World");
    }
}
```

Here it's still fairly clear, despite a few seemingly mysterious commands, you can still easily read what the program does, but let's take, say, C++ in our sights:

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello World!";
    return 0;
}
```

Here it's a bit less obvious what the program does, right? Plus look at the number of lines used to perform the task. There's no comparison.

I note, however, that in both these languages, you could print hello world with shorter code, I don't know these ways because I'm not a Java guy or C/C++ person, simultaneously too lazy a bum to look for them, nevertheless I'm trying to show the idea here.

Another example of how short code can be in Python compared to comparable code in C++/Java or other languages is below, this is slightly more complicated code, but maybe you'll understand something from it. It was provided to me in a comment by @jacekw on Steemit and it's his authorship. Thanks :)

So what will our assigned code do? Its task is simple:

1. create a list of numbers in descending order 19 to 0
2. Skip even numbers
3. Square each number
4. Sort in ascending order
5. Print the result

By the way, this is a form of algorithm, presented in step form, something we'll return to in the future. Currently remember one thing - an algorithm is simply an unambiguous set of instructions serving to accomplish some goal.

Anyway.

Let's start with *C++* maybe this time.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;

int main() {
    vector<int> s, a;
    for (int i = 20; i > 0; i--) s.push_back(i);
    copy_if (s.begin(), s.end(), back_inserter(a), [](int x) {
        return x % 2 == 1;});
    transform(a.begin(), a.end(), a.begin(), [](int x){
        return x*x;});
    sort(a.begin(), a.end());
    for (auto&& i : a) cout << i << " "; return 0;
}
```

I don't know about you, but for me this is quite a convoluted piece of code. As a beginning programmer, I would have trouble understanding it. Lots of strange symbols, marks. What exactly is going on here?

Hard to say at first glance.

The next participant in this comparison is. . .
Java

```java
import java.util.Arrays;
import java.util.stream.IntStream;

public class Main {
    public static void main(String[] args) {
        int[] a = IntStream.range(0, 20)
                .map(i -> 20 - i - 1)
                .filter(x -> x % 2 == 1)
                .map(x -> x * x)
                .sorted()
                .toArray();
        System.out.println(Arrays.toString(a));
    }
}
```

Python:

```python
a = filter(lambda x: x % 2 == 1, reversed(range(20)))
a = list(map(lambda x: x*x, a))
print(list(sorted(a)))
```

Alternatively, my version in Python would look like this:

```python
print(sorted([i*i for i in reversed(range(20)) if i % 2]))
```

I'll leave it without comment. Finally, I'll add, as a curiosity, an example from another language - Haskell.

```haskell
sort $ map (^2) $ filter odd [20, 19..1]
```

Also interesting, right?

Does this mean these languages are worse and Python is king? Absolutely not, never think that.

Each language is like a tool - it has its applications where it's good, excellent, but it also has ones where it's completely unsuitable. That's how it is here. That's how it is everywhere. Yes, sometimes there are fanatics of certain technologies or solutions whose language-technological brain inflammation obscures objective judgment, but that's nothing. We don't want to be like that. Let's be wise and reasonable, make our lives easier by using appropriate tools for appropriate tasks.

Nevertheless, Python allows us to write more in less code. This of course comes at a certain price that must be paid, and which makes Python good in certain situations and not in others.

4.9.2 Simplicity

Let's return to the previous point - if we look at Python code, you could say that it's basically just written commands in English. What does the word print mean? Nothing other than print/output.

You can immediately guess that the programmer wants the computer to output something to the screen. The case is similar with other elements of the language; really, just knowing English and we can almost understand a large part of Python. Many languages are similar, but they're not as close to English as Python.

Really, I haven't encountered another language this simple, and I've happened to use quite a few, at least superficially - whether JavaScript, Java, C, C++, Dart, Scala.

The only language that might compete with Python in simplicity is probably C - but that's because the core of that language is just tiny. When it comes to memory management, pointers, and other equally fun things from C, you start missing Python.

4.9.3 Python as a Dynamically Typed Language

What does this mean? Well, if you're a programming novice, you might have absolutely no idea what this is about, but that's okay.

In short, the matter concerns the fact that in statically typed languages, during variable declaration, you need to specify what type of data this variable will store. Let Java serve as an example here:

```
int someNumber = 123;
```

The above notation tells the 'program' that will execute our code that we want to create a variable named someNumber that will contain data of type int - integer, which is nothing other than whole numbers. And what does it mean that it has to create a variable?

I'll try to explain it quite simply, but I might not succeed, and if you don't fully understand how this mechanism works, or if it's difficult to imagine how it functions, don't worry, we'll return to the topic later.

This command will cause 'telling' our computer something like this:

Listen computer, here you have some data, (in this case '123'), remember this value, write it down somewhere, because I'll want to use it for something in the future, and from now on, whenever I write `someNumber` in the program, know that I mean exactly the value stored in that place.

Trying to save other data to this variable, let's say, an array or a floating-point number, won't end too well or according to our wishes.

In Python, there are no such restrictions and requirements. First, during variable initialization, we don't need to specify its type, and second, later we can easily change the kind of data we store in a given variable, so the equivalent of the above notation in Python would be the code:

```
some_number = 123
```

Later we can easily write:

```
some_number = 'Hi there'
```

What this results from, you'll learn a bit later in the book, but mainly it's about the fact that in Python variables are actually references to an object, not the object per se. As I said, we'll return to this, you don't need to worry about it for now. Just remember that Python makes your life easier and doesn't impose too much, does the work for you, the good snake.

4.9.4 Community

Python has one really big advantage. It's its community, which one, is really helpful, and two, its size is impressive. Thanks to this, the amount of available materials, tutorials, libraries, frameworks, and scripts can simply positively surprise you.

Thanks to Python's open-source culture, many amazing tools are daily put into our hands for use, completely free, just like that.

This causes that often we don't have to reinvent the wheel - just import some library that someone has already written. This often saves time and worries, allowing us to focus on what's important in our implementation.

Besides, what to do when we get stuck somewhere in program writing and don't know what next, when we encounter some error that we can't solve? Well, due to Python's age and the size of its community, we can assume in most cases that someone before us has already encountered a similar problem and asked about it on the Internet, or described the solution to that problem.

People are quite willing to share knowledge contrary to appearances. Thanks to this, we don't have to search for a solution for hours, digging through documentation, sources, or just experimenting. We can simply ask someone, because many people know Python, or find answers from others who solved this problem before us. Not every language has such a developed knowledge base and community.

As a counter-example, I'll give the Dart language. A rather new language, not very popular, overall small community. Nevertheless, I sometimes created in this language and it happens not rarely that I encounter some problem about which I can't find information anywhere, because simply no one else has encountered it yet, or no one else has described it, so I have to search for a solution myself, combing through documentation, sources and just experimenting.

This, in turn, is often a road through torment.

Additionally, sometimes there's the inconvenience that some things that have already been written by someone else in Python or Java, nicely packaged into a package and distributed for use, aren't necessarily available in Dart and need to be written independently.

Similarly, when it comes to learning the language itself - there are significantly fewer materials, they're often outdated due to the fact that Dart is a constantly developing language and strongly at that, new versions come out every week, at least they used to, whether it's still like that, I don't know, and due to low popularity few people write about it, even fewer create books or tutorials about it.

This doesn't make learning easier, especially for beginning programmers. Good thing that at least the documentation is quite good, although not as good as Django's documentation, for example, but still - it's not bad.

That's why I claim that Python's community is its greatest advantage and it's this community, literally, that creates this wonderful language, making it what it is.

4.9.5 Multiple Applications

Python is a general-purpose language. You can create practically anything in it, except for a certain, rather narrow group of applications for which it's completely unsuitable and for which it wasn't designed.

Knowing Python, we can create desktop applications, games, web applications, scripts, emulators, interpreters, compilers, scientific calculation applications, data visualization and scraping applications, machine learning, and so on. The list is really long.

Of course, Python is better for some tasks and worse for others, because for example, it's rare that desktop applications or games are created in Python, as there are better languages for that, but anyway, it's possible and not too difficult honestly.

What helps in the fact that Python can be used for many things is what I wrote about above - namely the large community creating huge amounts of libraries, frameworks, and ready scripts.

This allows for convenient use of Python in various fields and it's thanks to this and the language's simplicity itself that it's taking other fields by storm, beyond webdev and devops, like Data Science, Artificial Intelligence, Neural Networks, or scientific calculations in general.

Sometimes creating a program in Python really comes down to importing some module and adding a few small commands telling it what to do for us. It can't get simpler than that.

4.9.6 Readability

Python was designed with readability in mind. In Python, code block membership is indicated by indentation, which is somewhat different from most languages where brackets or parentheses are usually used for this purpose, or alternatively keywords like BEGIN or END.

In Python, indentation counts, and its incorrect use causes runtime errors. This results in the fact that practically every correct Python code is relatively elegant and easily readable. Of course, there are exceptions to this norm, but I'm talking about the general code where good practices or standards are applied, such as PEP8, for instance, which we'll talk about later.

When we add to this the simplicity and expressiveness of the language itself, it quickly turns out that code in Python is often simply beautiful, easy to read, modify, and friendly to novices.

Yes, people coming from other languages might find it strange, at least at the beginning, that in Python we use indentation instead of brackets or braces, but it's a nice solution in my opinion.

Additionally, Python lacks one more thing - semicolons at the end of expressions aren't necessary. One character less writing per line and cleaner code.

Of course, we sometimes use semicolons in Python, but these are rare situations and predetermined, really few.

This is probably also another thing that might surprise programmers of other languages, although in current times it's not such a rare practice for a language not to require semicolons.

Why is readability important at all? A programmer's time is expensive, our brains have very limited capabilities. It's good when certain things are immediately visible, when we don't have to think about something because it's obvious.

If a program is very readably written, we'll be able to understand it faster, and this is critical in completing tasks - contrary to appearances, a programmer's work doesn't consist of constant code typing, quite the opposite.

Personally, I spend most of my time at work reading other people's code - whether it's colleagues' code, or authors of libraries, frameworks, and sometimes even my own. Readable appearance helps a lot, and this is important because reading and understanding code is much harder than writing it.

4.9.7 Automatic Memory Management

In Python, memory management happens automatically - the programmer has no part in it, the language itself does it for us through such a mechanism as the Garbage Collector, it takes care of properly freeing resources and memory for objects we no longer use.

So we don't have to worry about things like memory allocation and de-allocation, as is the case in C or C++, for example.

Why is this an advantage? Because improper memory management can lead to very serious errors that put the entire system at risk, and ensuring that such errors don't occur is on the programmer's shoulders and often isn't a simple thing, ha! Sometimes even simple constructions related to memory allocation and de-allocation, things that seem obvious, have complicated backgrounds that lead to serious errors if misunderstood.

In Python's case, it's not like that - the programmer generally doesn't even have access to direct memory operations. This is a very wise limitation, useful in this type of language. It's similar in, for example, Java.

4.9.8 Support for Various Programming Paradigms

There are languages that strongly support basically only one programming paradigm - like Java or Smalltalk, which are designed to strictly fulfill the assumptions of the object-oriented paradigm, or Haskell, which is a functional language and only functional, but there are also ones like Python that support many paradigms. What's this all about, in human terms?

In Java or Haskell, you have somewhat predetermined how you should 'think' and in what key you should implement solutions to certain problems through code. What this means exactly, we'll discuss another time.

In Python, however, you have freedom of choice. You yourself decide which approach you like and which you'd like to use. I consider this an advantage because again — in some situations, some solutions work better, in others, different ones. Having a choice, you can use the right one and that's it.

4.9.9 Many Supported Platforms

As I mentioned somewhere earlier, Python supports practically any platform used today. Windows, Linux, AIX, IBM, macOS, OS/390, z/OS, Solaris, VMS, HP-UX. Whatever someone wishes for, almost certainly exists. Okay, supposedly now we only use Windows, Linux, and macOS, practically speaking, but even these systems aren't supported by some languages.

4.9.10 Maturity

Python is a language that has been around since 1991 - it's almost 30 years old now. During this time, its ecosystem, tools, and libraries have had time to mature, go through the growing pains that some languages still have ahead of them.

This means more or less that Python can usually be trusted. As long as the programmer doesn't cause something themselves, the language rather won't let us down, because it has survived the test of time, and most errors and glaring bugs have long been caught and patched.

Does this mean it's a perfect language or without errors? In no way. Nevertheless, because it has been used in hundreds of thousands of important business applications, it can be safely said that it's worth giving our snake a certain dose of trust.

4.9.11 Ease of Integration with Other Languages

Python can be quite easily integrated with other languages on various platforms. Programs written in Python usually quite easily cooperate with other programs, written in, for example, different languages.

Not every language has this feature, as some languages create quite hermetic, specific, and closed culture, where connecting or integrating them with other environments is extremely or unnecessarily difficult.

An additional advantage of Python is that you can write 'extensions' for it in C or C++, which will work much faster than Python itself. Thanks to this, we can have most of the application written in Python - code simple, short, and pleasant where it can be, and just some narrow bottleneck of it that needs to execute really fast, in C or C++. This isn't usually used, but sometimes there are various, strange reasons why it's worth it.

4.9.12 Speed of Code Creation

Due to Python's simplicity and multitude of libraries, applications, as well as the code itself, can be created in it literally lightning fast. This is undoubtedly an advantage, especially in times when most clients want their product to be done yesterday, and deadlines are always tight.

Moreover, usually this quickly made code is also of quite decent quality.

And it's also a fact that even if we don't want to use Python in production, we can still use it to create a tiny MVP. What's MVP? Minimal viable product - that is, such an app that will have minimum functionality, but somewhere someone will already want to pay for it because it will be useful for something, which will please investors and people in general, because it's super, we have MVP, VCs will throw money again, another round of financing, cash and hype checks out, our ship called startup sails on.

Remember this abbreviation - MVP is a hot buzzword in the crazy world of STARTUPS!

Ending the digression, even if we don't use Python in production but only for MVP, or creating some prototype simply, in Python we can do it lightning fast, check if a given solution works, if so, well, we can always implement the production version in another language. Some faster one.

4.10 Python's Disadvantages

Now let's talk about those worse sides.

4.10.1 Python as a Dynamically Typed Language

Wait a moment, just a second ago, I wrote that this is an advantage. What's going on? Do you have split personality, or what? Who said that? Hello. But seriously. . .

Python's dynamic typing is an advantage that allows us to create some great mechanisms, but also, in the hands of an inexperienced programmer, a disadvantage.

It allows for creating code that will cause completely unexpected, difficult to debug errors, which could be prevented in a statically typed language, where such code wouldn't even compile.

In Python, or other dynamically typed languages, there is no such mechanism, so you need to be somewhat careful here not to create an error that will later be difficult to diagnose and debug.

Of course, currently we have tools that make this task easier for us, or even somewhat make Python similar to statically typed languages, as there are, for instance, type annotations, allowing us to specify what type a variable/function should have.

However, this is not a mandatory or necessary element of the language and it won't cause an error during application startup attempt, at most a warning from the IDE or code analyzer, which can be simply ignored.

So dynamic typing is somewhat like a knife, on one hand you can use it to do something nice, make a good meal for example, and on the other hand, you have to live with the awareness that you need to pay special attention when handling it because you might cut yourself.

However, should we give up the benefits and applications it has because of this fact? I'll throw in a classic - Nothing could be more wrong!

I don't know about you, but I read this sentence with a special accent and certain voice in my head.

4.10.2 Performance

One thing is clear - when it comes to strictly performance issues, Python is far from being king. Generally, nice this Python, such not too fast one might say.

Of course, this is changing now, but the very nature of Python as an interpreted language means that it will never be as fast as C compiled to native code, or other languages of this type. You have to accept this and that's it.

Of course, I'm not claiming here that Python is very slow or sluggish. No. Python isn't slow, quite the opposite - thanks to various optimizations made over the years, Python has really gained in speed and today I firmly state that it's a language fast enough, but it must be strongly emphasized that it's not the fastest language. And that's it.

And while we're talking about performance, I'll mention sizes too - Python's hardware requirements mean that we simply won't run it on some platforms. There are certain areas of the embedded world where C or Assembly rules, Python doesn't exist there and there's no point in discussing this.

Of course, there are also projects like RaspberryPi, where actually, Python also rules everything.

So if you want to write highly efficient games with beautiful graphics, or maybe multi-threaded applications that handle huge amounts of calculations in real-time, or maybe tiny micro-controllers, then well, Python isn't really a good choice then.

In other cases, Python will handle it and you don't need to worry about execution speed/resources. Why? Well, we live in times when server time is much cheaper than developer time. This means it's better if the language is maybe a bit slower, but if you write in it much faster, we choose it. It's just cheaper, better, healthier this way.

This doesn't mean we have permission here to write any kind of code that works clumsily and slowly, but works. Absolutely not! You need to respect user time, hardware resources we have, and several other things. Save RAM wherever you are. Like in everything - you need to know moderation and limits. I'm talking more about theoretical situations here, where we have some code handling a server request.

Let's say the request passing through the network takes a second. Python execution and returning the response will take about 0.1 seconds. Then back to the user, another second. Total 2.1 seconds. We can rewrite this code in another language, let's say, Java - the code will be several times longer, writing it will take more time, but it will execute, say, 10 times faster. So instead of waiting 2.1 seconds, the user will wait 2.01 seconds, because usually it's not the server itself and our application code that is the bottleneck, but for example the database, network connection, or disk.

Does this make sense in most cases? The jump from 2.1 to 2.01s? Answer that yourselves. These are the situations I'm talking about - then usually there's no point in playing with optimizations and Python is simply fast enough.

That's at least how it is in the vast majority of projects, because those that can't afford this minimal slowdown are not very many. Besides, you - as a beginning programmer, probably won't even see such projects at the start of your career, because it's not time for that.

4.10.3 GIL

In Python, we have something called GIL - Global Interpreter Lock. I won't go into details of this mechanism here, it's enough that you'll know that because of it Python isn't exactly an ideal choice when it comes to multi-threaded applications, because only one thread at a time can have access to the interpreter in a process, because GIL blocks the rest.

And what's this about with multi-threading and so on? In short and great simplification, this is about performing many things simultaneously, most often using multiple processor cores, to speed up the operation of some application.

Because I don't know if you're aware, but you have something called CPU in your computer - the so-called processor. This processor is responsible for most calculations, computations, and executing your commands, in great shorthand.

During technology development, we reached a point at some stage where it was difficult to make one core faster. So, to make everything work even faster, and you could play five applications in the background, currently processors have several cores.

Cores are like small processors inside processors. Imagine a worker. 1 core = 1 worker. And going back to this earlier - a worker as a worker has limited capacity, because they're limited by physics, for example. Well, one person, no matter how strong, can't carry more concrete bags than X per hour. We, after some time, came to this moment when technologically we created a worker, that is, a processor, that approached this X, let's say.

So a problem appeared, because we have worker efficiency X. We have one worker on the construction site, we want to finish work faster, how can we do it, since getting more than X bags per hour out of this one worker will be difficult or impossible at the moment? We can try to make them even more efficient and, for example, fund them a good steroid treatment to make them stronger, or feed them cocaine/amphetamine, making efficiency increase by those 5%, but the cost of this undertaking would be completely disproportionate to the results obtained. So what to do? Hire more workers. Then, when we employ several workers, they can even be weaker than that one, but there will be several of them. Total processing power will increase.

More or less this is how the situation looks with processors and the fact that they are currently multi-core.

Here comes Python, which is kind of a handicapped foreman. He handles managing 1 worker well, but if he has to handle, for example, 4, he already has certain limitations that you need to be aware of.

True, this can now be worked around quite easily, but still something like this remains and you need to learn to deal with it, because you can fall into a trap.

4.10.4 High Expressiveness

Again, something that is an advantage is also a bit of a disadvantage. Why? Python hides certain things from you, the programmer, which causes that you don't always know how it's done 'under the hood'. This isn't entirely good, because sometimes it's useful to know how certain things were implemented and why just like that.

This explains a lot. A simple example of this is the frequent question - why do we index lists or arrays from 0? If you're a C/C++ programmer, you most likely know the answer.

High-level language programmers, however, don't always know it. Ha! I would even say rarely. Don't be afraid if you don't know, we'll address this topic in the book, but a bit later.

However, this is a small price to pay compared to what this expressiveness and high abstraction offers.

It's just an easy problem to fix - just need a bit of willingness to read a bit more. And the time we'll spend on exploring these various topics is much shorter than the time we would spend writing our program in a language with a lower degree of abstraction/expressiveness.

4.10.5 Python Doesn't Exist in the Mobile World

Mobile applications and Python are rather two different worlds. Just like that and that's it. Sure, there are projects trying to achieve something in this area, but there's no point in kidding ourselves that knowing only Python, we'll create a nice app for Android.

If someone tells you otherwise, better ignore the guy, because Python has gotten too strong in them and they're talking nonsense.

4.10.6 Too Much Comfort

It can often be that after you start writing in Python, switching to other languages, where you have to do certain things completely differently, is a bit painful. This is also a potential disadvantage of Python.

You write your programs happily in Python, you do a lot of things with one line of code, it's nice and beautiful, but suddenly you have to write something in Java and there's a brutal collision with reality, which causes you to land in the depths of darkness, despair and depression, your life loses meaning and your wife has to throw you out of bed in the morning to get you up. No, I'm joking, writing in Java isn't that bad, I have nothing against the language. It's just that few languages are as nice as Python.

4.10.7 That's It

Writing about Python's advantages and disadvantages, I tried to be relatively objective. Of course, this is not very possible due to the fact that this book is about Python, and I myself am its enthusiast. Nevertheless, I believe I managed to present you with Python's strengths and weaknesses, thanks to which you can decide whether it's worth learning it. In my opinion, absolutely yes!

Besides, darn it, since you already have this book, use it and learn!

4.11 Who Uses Python?

In this case, it would be better to ask who doesn't use Python.

For example, however, I'll give you a few more or less known companies that use Python, these are: ILM, Google, Facebook, Instagram, Spotify, Quora, Netflix, Dropbox, Reddit, NASA, NSA, Red Hat, Nokia, IBM, Nasdaq, Sephora, Citi, Toyota, Gartner, Atlassian, Evernote, Lego, WebMD, Telefonica.

The entire YouTube basically stands (stood) on Python. At Google they say: 'Where we can - Python, where we must - C++' (supposedly).

Quite a lot, right? Well, no wonder, given that Python, according to the TIOBE index, is currently the 3rd most popular programming language in the world. Above it are only Java, C, C++. Additionally, Python gains more popularity every year and grows in strength. As someone once said, not necessarily wise, `you can't stop this force anymore`.

Below you can see a table with the TIOBE index, which, let's say, is a standard in the programming world when it comes to measuring the popularity of certain technologies, trends, and so on.

Sep 2019	Sep 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.661%	-0.78%
2	2		C	15.205%	-0.24%
3	3		Python	9.874%	+2.22%
4	4		C++	5.635%	-1.76%
5	6		C#	3.399%	+0.10%
6	5		Visual Basic .NET	3.291%	-2.02%
7	8		JavaScript	2.128%	-0.00%
8	9		SQL	1.944%	-0.12%
9	7		PHP	1.863%	-0.91%
10	10		Objective-C	1.840%	+0.33%
11	34		Groovy	1.502%	+1.20%
12	14		Assembly language	1.378%	+0.15%
13	11		Delphi/Object Pascal	1.335%	+0.04%
14	16		Go	1.220%	+0.14%

As you can see, Python easily beats languages like C#, PHP, JavaScript, SQL, R, or Ruby. The latter actually by 8 times.

This snake has a future.

I really think that we are currently at a point, or quite close to such a point, where too much code has been written in Python for it to be displaced in the future. Let's look at how Python gains popularity over the years.

Programming Language	2019	2014	2009	2004	1999	1994	1989
Java	1	2	1	1	9	-	-
C	2	1	2	2	1	1	1
Python	3	7	5	6	23	21	-
C++	4	4	3	3	2	2	2

Impressive, right? 20 years ago Python wasn't even in the top 20. In 2004 already 6th. And now? On the podium.

Look also at the old monster - COBOL - despite the fact that it wasn't a very well-thought-out or beautiful language, it is still being developed and used today because at one time a lot of software was written in it, especially for banks, and simply getting rid of all systems operating based on this language and rewriting them to something newer would be too costly.

So despite COBOL being quite a specialized language with narrow application, it's still used, and yet the first mentions of COBOL come from 1959, so if someone wants to, they can program today in technology from basically 60 years ago and find work in it.

It's not among the pleasant ones, but well. Can you? You can.

As I mentioned earlier about Python - you can't stop this force anymore. Python came, made itself at home, and for now isn't going anywhere, and probably will stay for good.

I'm not sure about the latter, of course as a Python programmer I'll try to make it happen - adding my brick and creating as much code in Python as possible, nevertheless I (still) can't give a guarantee that it will stay with us forever.

I can give it for something else - that within the next 15 years Python won't be displaced from generally understood popular programming and it won't be a problem to find work knowing this language, so if you're afraid of that, you have my word that it won't be like that.

So going into Python, you're making quite a smart move when it comes to your career. So do it, listen to Górski, and get to work.

4.12 Python Compared to...

4.12.1 Java/C

I'll throw these two languages into one bag, to some people's outrage. Fan boys will come at me with hate right now, but whatever.

Nevertheless, let's start with the fact that both these languages are supported by/directed by large corporations. Python isn't. For some it's a disadvantage, for others an advantage. Python is also definitely simpler to learn than these two languages, without two words.

It's also much more expressive - the code is usually shorter, much shorter.

Unfortunately, often also slower than both.

Python is also less popular than Java, which is currently at the very top and that's without two words, but then again also clearly more popular than C#.

Additionally, Java/C# is more often used by large corporations than Python. Huge projects are often the domain of Java or C#. For me another disadvantage.

4.12.2 Perl

Well, few people currently use Perl, but because these two languages were often compared in the past, I'll mention it too. Personally, I've never written in Perl, but I've happened to see code written in it.

It can evidently be sometimes not very readable, and the amount of parentheses used in Perl can overwhelm a person. Python is also much more popular than Perl now. Overwhelmingly more popular. I won't elaborate further on this comparison then, because it makes no sense.

4.12.3 C

C is a 'small' language - its core is small, but giving amazing possibilities, nevertheless it's still a language with certain limitations.

C rules in places where Python has no reason to exist - drivers, embedded, applications where speed or performance is really critical. In C the programmer is able to manage memory used by their program down to the byte. In Python? Forget about something like that.

Python is therefore much slower than C, takes up more memory. In return, we gain much greater expressiveness, speed of writing code and safety - Python is simply a simpler language compared to C, but rather they're never compared. Why?

Because the default implementation of Python is written in... C for one thing, and two, they serve completely different tasks. These are languages that I would say are complementary, not opposite, as they well complement each other in their weaknesses.

Many libraries that are written in C for speed have their wrappers written in Python precisely. What are these wrappers? Think about your car. Inside it probably has quite a complicated computer. You, as a user, are able to simply interact with it through various buttons, switches, menus and so on - easy thing. However, you're not able to modify by yourself how this system specifically works, what it does in what situation etc., but this doesn't really bother much, because in 99% there's no such need, and for that 1% there's no point in abandoning the car and starting to walk on foot. And when you already need to change something in how this computer works... Such things can be done by a mechanic using specific tools and software modifications.

Python and its programmer is you here, and C is the car's onboard computer.

Where incredible performance is needed, which probably only Assembly can beat, you can use C, where readability and speed of code production are more important, you can use Python. Super combo. By the way, a large part of Python programmers also knows C. This is probably the natural order of things.

End of these comparisons, because you could go on endlessly. At this point you should know well Python's strong sides, as well as those weak ones.

4.13 Python Implementations

Python has several implementations. The most popular and default one is CPython - written in C, hence the name. This is the implementation we'll be using.

There are also others, like:

- Jython - Python implementation for the JVM (Java Virtual Machine)
- IronPython - Python implementation for .NET
- PyPy - Python implementation written in Python (RPython to be precise)
- Stackless Python - Python implementation focused on concurrency
- MicroPython - Python implementation for microcontrollers

Each of these implementations has its advantages and disadvantages, but we won't go into details here. Just know that they exist and that's enough for now.

4.14 Questions

1. What are Python's main advantages?
2. What are Python's main disadvantages?
3. What is GIL and why is it important?
4. Why is Python's dynamic typing both an advantage and a disadvantage?
5. What makes Python a good choice for beginners?
6. In what areas is Python particularly strong?

7. In what areas should Python not be used?
8. What are the main differences between Python and C?
9. What companies use Python?
10. What is Python's position in the TIOBE index and what does this mean?
11. What are the different Python implementations available?
12. Why is Python's readability considered an advantage?
13. How does Python's memory management differ from languages like C?
14. What makes Python's community special?
15. Why is Python's expressiveness both an advantage and a potential drawback?

These questions should help you review and understand the key points discussed in this chapter. Try to answer them without looking back at the text - this will help you assess how well you've understood the material.

5 Setting Up Your Environment

Let's finally get to something concrete - let's get our hands dirty, as so far I've just been talking and talking. We'll start by setting up the environment/installing Python and something for text editing. Just a moment, what exactly? There are two options - a simple text editor or an IDE. Which one will we choose?

5.1 What to Use for Writing Code Initially?

What should you use for writing code at the beginning? IDE/Text editor? PyCharm? Sublime? VS Code?

When I think back to my first days of learning programming, one of many things that constantly confused me and caused quite a bit of trouble in deciding was choosing the environment in which I would write my first programs.

This shouldn't be surprising - nowadays, practically every language has at least several free IDEs, plus paid ones, and then there are text editors and so on. It's easy to get lost. At least that's what happened to me, and instead of learning programming, I spent several days sitting and thinking about which environment to choose.

5.1.1 IDE? Text Editor?

So how is it really? IDE or text editor? And what is an IDE anyway? Find this out yourself - as an IT professional, your basic task is processing information. So look up the meaning of this acronym on the Internet, search for the differences between an IDE and a text editor.

I won't hand this to you on a silver platter because the ability to search for information is a key trait that makes a good IT professional/programmer, which needs to be practiced from the beginning, and this is what I'll try to develop in you throughout the pages of this book.

So what to choose? The answer might surprise you - it's: it doesn't really matter. Let me explain.

Whether you choose CodeBlocks, VisualStudio, or Atom doesn't make much difference in my opinion, as these are just tools. Tools in the hands of a programmer, and it's up to them how well these tools will be used. It's similar with languages.

Even the best IDE won't help you if you don't know how to use it. On the other hand, if you master seemingly prehistoric programs by today's standards like Vim or Emacs, you can work wonders.

Therefore, I recommend not paying too much attention to which environment you choose, as practically any can be adapted and made to work excellently.

Despite this, we can probably outline one path that's worth taking and which I think is quite reasonable. What path is that?

5.1.2 Basics

Regardless of whether we use an IDE or some text editor, it's good to know how everything works under the hood, which I've already repeated several times and will repeat many more times.

So it's not like someone without their IDE can't compile a piece of C++ code or run a Python script from the console. Such basic things need to be learned because they allow you to better understand how programs are created and how everything works. Sometimes this information is simply essential.

Besides, at the beginning, code completion should be turned off. By typing out instruction names, methods, or classes ourselves, we'll learn them faster. At least that's how it was for me. Using auto-completion, often it was enough to type two or three letters, hit tab, and bam - done.

The problem appeared when auto-completion was missing, for example, when I had to write something quickly in a regular text editor/use someone else's computer or, horror of horrors, went for a job interview where I had to write code on a whiteboard with a marker. I won't tell you how stupid I looked several times because of this.

Suddenly everything flew out of my head and I forgot half the commands I supposedly knew so well. Therefore, at the very beginning, I gave up auto-completion in favor of manual typing.

So at the beginning, it's better to forget about syntax suggestions, especially if you're planning to go for a job interview sometime soon. Although I doubt it, since you're reading this book.

Later, when I had memorized enough keywords and instructions, I returned to it. However, I would advise against auto-completion at the very beginning.

5.1.3 How It Looked for Me

What environment did I choose?

To be honest, I tried many different ones, following roughly this path while learning Python:

At first, I wrote in the browser, doing a course at one of the interactive programming academies - I didn't need to download any program, but later, when I wanted to start writing something of my own on disk, I looked online and found various threads where PyCharm and several other popular IDEs were usually recommended. So I downloaded them. But not for long.

They were too overwhelming for me - the vast number of options, complexity. Besides, my aging laptop struggled with the hardware requirements of such IDEs. Working in them was basically not very possible.

So I switched to SublimeText, where I stayed a bit longer, slowly getting used to the console and the fact that I had to do some things myself. After a while, I started missing something in Sublime, so I started searching again - I found Vim and used it for some time.

A wonderful tool. A Swiss army knife - you can do everything with it. It can be a light text editor or basically a full-fledged IDE with the right number of plugins. I liked it for its configurability, the magic you can perform in it.

At first it was difficult, but then I got somewhat used to it. Nevertheless, I rather wouldn't recommend Vim as a first text editor, as it's still somewhat different in operation than standard programs.

I changed Vim to SublimeText for short and simple scripts, and I write all other applications in PyCharm. It offers too many useful options not to use it, and two, it has become somewhat of a standard in our industry and there are really few companies that don't use it.

I started missing something in Sublime, so... Back to Vim+PyCharm. And this has been my final stack for several years now.

For the purposes of this book, SublimeText will be completely sufficient for you, then we might switch to PyCharm. Let's move on to installation, since we already know what we'll need.

5.2 Windows?! Linux?! macOS?! What to Choose?

It doesn't matter.

Soon there will appear here or somewhere else fanatics of one or another system who will try to convince you that it's precisely their chosen system that's the best and only suitable for use/programming/anything.

Don't bother with them. The system is just a tool and it's up to you whether you use it efficiently or not. Unless you're interested in languages that somehow limit the freedom of choice, because, for example, wanting to write iOS apps in Swift, you probably won't be sitting on Windows, similarly writing software in pure C#, you probably won't be using macOS. Simple matter.

This doesn't concern us though. We're simple Python folks. And it runs quite nicely everywhere.

However, it's good to have at least basic familiarity with Linux. Why? Practically every server where we host our applications on the web runs on Linux. Sooner or later you'll encounter it, and if you're somewhat familiar with it earlier, it won't be a painful collision with reality where you bang your head against a wall, but rather a meeting over a beer with a good friend you haven't seen in a while. That's one. Two, generally certain things on Linux you have at hand, where on Windows you have to bother with some things yourself, install them, etc. Does this mean you have to install a new system to get to know Linux, otherwise you can't live?

Absolutely not. A virtual machine with any distribution will be enough for this. I personally recommend Ubuntu, despite not being fond of Canonical, and the second option, Manjaro. Both are Linuxes, but with some differences. Personally, when I use Linux, it's Manjaro. On the desktop at least. Because on servers Manjaro rather doesn't count.

Virtual machine, what, how, where? Browse the table of contents, in this book you should find the answer, just probably it will be a bit further.

Moreover, there's another option available recently. WSL. Windows Subsystem for Linux. This is a Microsoft toy that enables you to have Linux integrated with your Windows. Linux in Windows from Microsoft. Boom. Advantages include ease of installation, easier integration, etc. Generally recommended.

To summarize briefly, both Linux and Windows have their advantages and disadvantages, they're just tools. I, personally, however, use (finally) Linux as the host. If you're on Linux, that's cool, if not then definitely install WSL.

5.3 Installation on Windows

Usually in Python books we find a step-by-step description of Python installation, helpful screenshots and so on. Well, not here. We'll install the things we need in a somewhat unusual way for Windows. Using the console. No, I haven't gone mad.

I don't know if you're aware, but Linux users usually install programs in a somewhat different way than Windows users.

Namely, every popular Linux distribution contains something called a package manager. You can think of it as a 'program' that manages all officially supported and available programs for a given distribution. Understandable, right?

Thanks to these so-called package managers, Linux users can do something cool - most programs can be installed on Linux with a single command, similarly with their updates, system updates, or removing 'programs'. On Windows it's a bit different. In my opinion, Linux's way of managing packages is more convenient and probably many people will agree with me at this point.

Unfortunately, by default, Windows doesn't have a sensible package manager, however, I didn't use the word 'by default' without reason. Namely, a very nice tool called chocolatey was created, which enables us to obtain functionality similar to what pacman or apt provide.

In short, thanks to it, installing a large part of programs is reduced to:

```
choco install program_name
```

And that's it. Convenient, right? Therefore, we'll use this package manager to install what we need.

5.3.1 Installing Choco

Detailed instructions can be found on the chocolatey website and that's where you need to go to find out how to install choco.

5.3.2 Installing What We Need Using Choco

Chocolatey installed? Excellent, now just one command separates us from the finish line.

```
choco install python sublimetext3 -y
```

And done. What about when we want to update all/selected programs?

Again, nothing simpler:

```
choco upgrade all
```

or

```
choco upgrade program_name
```

I probably don't need to explain. The list of packages available for installation this way can be found here. Most popular programs are there.

If we don't want to click 'y' for every upgrade question, we can just add `-y` at the end of the command, similarly to the example with installing SublimeText.

Now we just open SublimeText, create a directory anywhere, and in SublimeText click File → Open Folder, selecting the previously created folder that will serve as the main folder for files from this book.

5.4 Installation on Linux/macOS

As for instructions for Linux or macOS users. Well, there aren't any for you.

But how? What kind of discrimination is this? Well, you probably already have Python installed, just check which version. How?

```
python --version
```

(two hyphens) in the console and done, or, depending on the distribution, it might be

```
python3 --version
```

Depends on which distribution you're using and which Python version it uses as default.

Best if it's Python version >= 3.7.0. If not, upgrade it.

How to do it? Let's be honest, if you're using Linux, you probably don't need to ask me about this. Similarly with installing SublimeText. And if you really have no idea then... Remember what I said about the most important skill of an IT professional - processing information. So search for it.

And if you're on a Mac, use the `brew` tool. That's it.

And as I mentioned, it doesn't have to be SublimeText - you can use your favorite editor. Doesn't matter. Just please, don't use Windows Notepad for this. Every time you do this, a kitten dies somewhere in the world. Don't do it.

5.5 Let's Get to Programming! Finally!

Well, almost. Now we still need to, using the console of course, navigate to the main code directory for this book that we created recently.

To do this, you need to open the console again, this time you won't need administrator mode, so open it as a regular user, whether on Linux or Windows.

And now what? Time to get acquainted with the basic commands that might be useful. Mainly it's about four basic ones and these are what I'll discuss, there are of course many more and some of them that you might need in daily work, we'll discuss later.

I simply don't want to overwhelm you at the very beginning with information that won't be needed right now, and will only add things to the list of topics you need to understand.

5.6 The Four Horsemen of the Console

1. `dir` (Windows) or `ls` (Linux). This command lists the contents of the directory where we currently are in the console. How to know where we are? Simple - our working directory (CWD - current working directory) is displayed to the left of our terminal cursor or by typing pwd on Linux/iOS. Windows users google it. They wanted systems from MS. Now you have it. What does it give us that we're in some directory? Namely, our commands will be executed relative to this path. So if we type `python file.py`, if we're in the folder, let's say, `C:\Users\Olaf`, then Python will look for this file, file.py, in this folder.

2. `mkdir` - common for both systems, creates a directory with the given name, e.g., mkdir folder will create a folder named folder in the current working directory. Makes sense.

3. `del` (Windows) and `rm` (Linux) these commands are used to delete files. If we want to delete a folder, we can use rd on Windows or rm -rf on Linux. Example usage: rm -rf file.txt.

4. `cd` - common for both systems. Cd, meaning change directory. Changes the current working directory to the specified one. Example usage - cd .. - this command will move us to the parent directory in the file structure. So if we're in `C:\Users\Olaf\Test\`, using cd .. will move us to `C:\Users\Olaf\`.

Useful information: pressing TAB causes the console to auto-complete directory or file names for us;

On Linux (usually) the ~ character denotes the user's home directory, so if you want to go to it, you don't need to provide the full path. Just type cd ~ and you're there.

One more note, not all Linuxes have the current working directory imported to PATH, so sometimes there might be a need to type, as the file location in the current CWD, ./file.txt instead of file.txt, depending on the distro. This can be easily fixed by adding CWD to PATH. The same might be with Python itself. Remember to add it to PATH during installation. Or if not during installation, add it manually.

How? Google it, Linux user. You wanted to install Linuxes, now you have it. Struggle, struggle, as you wanted!

We can navigate now, but we'll return to the console. How to run our source code?

Simple matter.

```
python filename.py
```

Note, if you're on Linux and had Python 2 installed earlier, you might need to use the python3 command to run it, but you probably know that.

And to run the Python interpreter, just type:

```
python
```

And that's it. This will launch the interpreter for us. The difference between the interpreter and running from a file is that in the interpreter we type commands and Python interprets them on the fly. In the case of a file, we have a less interactive approach. Play around and see for yourself.

Environment set up, console navigation too. Let's finally get to work!

6 Hello, World!

Finally, we're starting to code! Phew, it took us a while, didn't it? About 50 pages or so. Anyway.

6.1 Printing Text to the Screen

Hello, World! These words are quite popular in programming, at least their English version - Hello World!

It's traditionally considered the text that a beginner programmer's first program will print to the console - printing 'Hello World!' on the screen.

We'll do something similar, but slightly differently, in two ways. Why? Well, you often hear opinions on the internet that Python already has a solution for everything, ready-made code that someone else created, and we just import it and use it - and that's basically what programming in Python is all about.

Like building a program from ready-made blocks. The key is knowing which blocks to use.

Well, that's often true. Very often. And it's no different with hello world. The examples shown here should be typed into a file and run as described in the previous chapter, or typed directly into the interpreter. I recommend the second option, at least while we're dealing with one-liners/few lines. For longer things, I recommend working with files rather than just the interpreter.

In Python, the standard Hello World can be replaced with:

```python
import __hello__
```

What will appear before our eyes?

```
Hello world!
```

Yup. Python even has a ready solution for hello world, but that's more of a curiosity - now I'll show you how to print something to the screen using Python in the normal way.

```python
print("Hello, World!")
```

And that's basically it. The console will display: "Hello, World!". That's all there is to it.

A small note for complete beginners: if you don't know where to enter this code, it's simple - create any file with a .py extension in your working directory (CWD), check the chapter about the four horsemen of the console if you've forgotten what that means.

Then enter this line in that file using a text editor. After that, all you need is:

```
python your_file_name.py
```

and you're done. Or instead of python, you might need to type python3, depending on how you installed everything.

What happened here? We used one of Python's built-in functions that are included in the language's standard library - a collection of functions that every Python3 installation has. This function is called print - from English, meaning to print.

Hmm, so what might a function called “print” do? Good question. I think this will be the real test of whether you’re cut out to be a programmer. If you can figure out what the print function does, you probably have what it takes to be a programmer. Congratulations.

We pass this function an argument (something for the function to act on) in the form of what we want to print. It will be displayed on the screen, along with a newline character. The newline character, which Python automatically adds to what we print, means that if we now use another print to print something else, our value from the second print will be displayed on a new line.

Because the computer needs to know when to display a ‘new line’ and end the current line. To let it know when to do this, we have something called a ‘newline character’. When you press enter in Word, this character is inserted underneath. So:

```
print("Line 1")
print("Line 2")
```

Will give us, as expected:

```
Line 1
Line 2
```

The newline character is usually `\n`. So in reality, instead of just displaying “Line 1”, Python displays `"Line 1 \n"`.

Is all this simple? Anti-climactic? Yes. At least apparently. Because underneath, many very, very interesting things are happening that you have no idea about yet.

The fact that today, with one line of code, you can print some text in the console is the result of decades of work and foundation-building by the fathers of computer science. I know it might sound funny, but that’s how it is. Look at Assembly code, for example, the language everyone used to write in.

The code fragment below is Assembly, a very low-level language that allows very detailed direct interaction with computer memory, processor, basically everything. This gives the programmer the ability to manage and optimize almost everything, but it comes at a price - since you have to manage everything yourself, well... You have to do it yourself. It has its pros and cons. Let’s not worry about it too much right now, I’m just mentioning it so it stays somewhere in your mind. Assembly = fast, low-level language where you have to do a lot of things yourself, which is very close to the processor/memory, with a low level of abstraction.

```
segment .data
msg     db      "Hello World!", 0Ah

segment .text
        global  _start

_start:
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, msg
        mov     edx, 14
        int     80h

; exit program
        mov     eax, 1
        xor     ebx, ebx
        int     0x80
```

Wait... What? So yeah. Appreciate what you have now.

And I'll note right away - don't worry if you completely don't understand this code. Relax. I don't understand much of it either. It doesn't matter. It's just to show you interesting, old ways.

And you know what's even more interesting?

The fact that currently, underneath, Python looks exactly like this. That is, not Python itself - because Python is just a language - a set of rules, definitions, but I'm talking about CPython - the implementation of the Python interpreter, and those can be arbitrary, I wrote about this earlier, you can read about it. It's worth remembering that CPython is Python's default implementation.

The difference between Python and CPython is that Python is simply a language, meaning a set of instructions and a description of what functions this language has and how it should behave in given situations.

And CPython is already a specific implementation of this - translation into computer behaviors, a specific program executing commands in a specific way. So CPython != Python.

So let's repeat once again. Python is a language. The Python interpreter is already some program that interprets code written in Python and executes specific commands. Usually, when we talk about the Python interpreter, we mean its default implementation, CPython - a Python interpreter written in C, but there are others. Remember. Also remember that the implementation details of Python interpreters differ from each other. The language authors allowed certain behavioral decisions to be made by the people implementing the interpreter. Therefore, CPython may, but doesn't have to, sometimes behave differently than Jython. So it's good practice not to rely on interpreter implementation details but on the language specification itself. But let's get back to the topic.

6.2 Binary Language - The Only Thing a Computer Understands

I don't know about you, but I've always found it interesting how a computer works. How it happens that after pressing some magic button, electrical energy starts 'flowing' through this wonderful machine, various characters appear on the screen, and everything is so beautiful and nice.

Well, the matter is simple. At the very foundations of how computers work lies nothing else but two simple things: True and False, 0 and 1, yes and no. I'm talking about the binary system, machine language.

What do I mean by that? Well, a computer is nothing more than a huge, huge group of conductors/semiconductors stuck together, like 'switches' that can be 'on' or 'off' - they have two states. These states are regulated by the voltage of the current flowing through the conductors, it determines whether one is on or off.

Depending on what combination we have, which 'conductors' we have turned on, which turned off, the computer will do different things. It's like with a washing machine - depending on which buttons you leave pressed, it will do something different.

Since we have two states here, anything that has to do with this is often adorned with the adjective 'binary'. Binary number system. Binary choice. Binary tree. Binary people living in a probabilistic world, and so on.

Anyway. So we establish one thing. Our computer operates only on two values - 0 and 1, or no/low voltage and high voltage. That's all. In a great simplification, this is exactly what is the complete basis of the entire computer and nothing more.

I assume my readers are smart people. So a question should appear right away - But how? If a computer only understands 0 and 1, how can I type different letters here, read them later, move the mouse, enter numbers other than 0 and 1. What? You're making this up, Mr. Górski.

Well, nothing could be further from the truth. Your computer really only understands 0 and 1. Everything else is the result of various calculations, conversions, and encoding of other values to these 0s and 1s.

This text is a perfect example.

6.3 How a Computer Sees Letters - Binary System

Imagine that all the letters you see here are actually nothing more than a number, written in binary system.

For precision - what is a number written in binary system? It's nothing else than a normal number, just expressed using only 0s and 1s. We, as humans, chose decimal system as our basic one. Probably because that's how many fingers we have, but who exactly knows.

Anyway - we operate on the assumption that we have 10 digits, each order of magnitude can have 9 or 10 possible states, or may not exist at all, and orders of magnitude are based on powers of ten.

In the binary system, it's actually similar, except instead of 10 digits, we have only two and two values. Additionally, we calculate subsequent orders of magnitude not based on powers of ten but on powers of two.

Let's take, for example, the number 123. How do we calculate its value? Well.

1 - number of hundreds, third digit 2 - number of tens, second digit 3 - number of ones, first digit

$1 * 10^2 + 2 * 10^1 + 3 * 10^0 = 1 * 100 + 2 * 10 + 3 * 1 = 100 + 20 + 3 = 123$

As you can easily see, the exponent is a number equal to the digit number, counting from the right side, decreased by 1.

This all sounds complicated because we don't think about it this way - we do certain things naturally due to experience, so it might take some practice to switch to thinking about it this way.

The situation in binary notation will be analogous. How do we calculate the value of a given number in binary notation? Let's assume we want to know what value the number

101101 has.

```
1 - 6th digit of the number
0 - 5th digit of the number
1 - 4th digit of the number
1 - 3rd digit of the number
0 - 2nd digit of the number
1 - 1st digit of the number
```

Therefore:
$1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$
$1 * 32 + 0 * 16 + 1 * 8 + 1 * 4 + 0 * 2 + 1 * 1$
$32 + 0 + 8 + 4 + 0 + 1$=45

So 101101 is nothing else than the equivalent of 45 in decimal system. And how to convert from decimal to binary? Very simple. You divide the number by two and write out the remainder each time.
$45/2 = 22$ remainder 1
$22/2 = 11$ remainder 0
$11/2 = 5$ remainder 1
$5/2 = 2$ remainder 1
$2/2 = 1$ remainder 0
$1/2 = 0$ remainder 1

Now, we read the remainders from BOTTOM to top: 101101. Does it match? Yep.

Analyze this slowly and carefully for now. It's nothing if something isn't clear at first, it's just a way of converting from one system to another.

Knowledge of powers of two values comes in handy here. When you have some of them in memory, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16386 etc., some conversions become easier and faster.

Anyway. Let's get back to the topic. So... Ah yes. Letters are also numbers.

6.4 Character Encoding - ASCII

Right. Since a computer understands only and exclusively numbers, those in binary system, meaning essentially zeros and ones, how do we tell it to store some letter?

Can you imagine a world where instead of reading my words on the screen using letters, you would read each letter written somehow using binary system, manually translating it to human on paper? I can't, but that's more or less how it looks in reality, only the computer does it for us.

Well, a few smart gentlemen gathered once and decided that it would be a good idea to create a kind of translation - a translation, mapping of alphabet letters to... numbers.

Each letter of the alphabet received its unique code in the form of some number. Why? Because how do you tell a computer that 'k' is 'k'? You can't. The computer is stupid, it doesn't understand the concept of a letter. At least for now, in a few or several dozen years, who knows.

Numbers in decimal system, however, we can easily convert to numbers in binary system, meaning to something that the computer will already understand.

And so we created something called the ASCII standard, which stands for American Standard Code for Information Interchange. It's a kind of table that contains mapping of English alphabet characters, punctuation marks to numbers. Like a dictionary of sorts, which letter corresponds to which number.

For example, 'A' - capital A, was marked as 0100 0001, or 65. 'a' is 97. The newline character is 0000 1010, or 10. Why? Because that's how it is and that's it. That's what the American wise guys came up with and that's it. It's kind of a conventional matter.

A small note, we usually write numbers in binary system with the prefix `0b`, so it's clear that we're dealing with binary. Because how do you distinguish between 10 in decimal and 10 in binary? The notation is the same but the values are different. It's like in that joke that there are only 10 types of people in the world, those who understand binary language and everyone else. Hehehe, a programmer's dad joke, sorry.

B in ASCII encoding is 0b0100 0010, or 66. C will have the number 67. And so on. Guess what D will have?

Every letter you see here is translated in a similar way and stored on your computer's disk as a sequence of ones and zeros. Then, when reading, the computer, after interpreting what letter a given number is, displays the specific letter. However, it completely doesn't understand that this 'k' is some letter, and not a piece of binary code. Simply.

What distinguishes ASCII? ASCII is an encoding where each element can be expressed using 7 bits. What does it mean, using 7 bits? A bit is nothing else than a 'digit' in binary system.

At least that was the original assumption. It therefore allows for the translation of 128 characters, to numbers from 0 to 127, inclusive.

Many people, at least those who had some contact with programming or computer science, will be surprised. What do you mean - 7-bit ASCII? Everyone often learns that it's 8-bit. Well, no, originally ASCII was designed as a 7-bit system, having 128 characters.

The fact that today we often think about ASCII in the context of 8-bit comes from the fact that during the creation of ASCII, 8-bits wasn't yet such a standard. Many encoding systems were 7-bit, and they used the 8th bit for their own various strange needs.

What does it mean that 8 bits is a 'standard'? Well, nowadays we have something called a byte. A byte is a collection of 8 bits. Like 1111 0000 or 1000 0000.

It's such a very small unit of your computer's memory that you can use. Whether in RAM or disk memory.

It's clearly defined and simple. But... A byte wasn't always defined as 8 bits. There were systems where 1 byte, the basic unit, was defined completely differently - as 2 bits, 7 bits, 6 bits. Pick your poison. Free for all.

In general, many things in computer science or programming, like in mathematics, are conventional. Get used to the fact that sometimes we do something in a specific way just because that's how it is. Computer scientists basically come from mathematicians in a straight line, so we're strange people.

That's why today, despite ASCII originally being a 128-character 7-bit system, it's written using 8 bits, giving it 256 possible characters, officially, 8-bit ASCII we call extended ASCII, but colloquially, usually when we say ASCII, we mean this 256-character one - with several additional characters.

Selected fragment of the ASCII character table:

Decimal	Character	Binary	Decimal	Character	Binary
65	A	0b1000001	66	B	0b1000010
68	D	0b1000100	69	E	0b1000101
69	E	0b1000101	70	F	0b1000110
72	H	0b1001000	73	I	0b1001001
73	I	0b1001001	74	J	0b1001010
76	L	0b1001100	77	M	0b1001101
77	M	0b1001101	78	N	0b1001110
80	P	0b1010000	81	Q	0b1010001
81	Q	0b1010001	82	R	0b1010010
84	T	0b1010100	85	U	0b1010101
85	U	0b1010101	86	V	0b1010110
88	X	0b1011000	89	Y	0b1011001
89	Y	0b1011001	90	Z	0b1011010

Simply put, there was chaos before. And from chaos emerged order. And computer science was born. In this place, order and chaos coexisted in harmony so you could show me your wares. Okay, enough nerdy references to Gothic.

Well, anyway. Because I lost the thread.

We have this encoding, in this case ASCII, and it's elegant. Although not completely, because a problem appears. We use ASCII on 8 bits now, right? Yes. 8 bits, that's 8 zeros or ones, right? Yes. Using 8 zeros or ones, we can express 256 numbers.

Taking into account that 1 number = 1 letter, we'll quickly see that we have a maximum of 256 characters to use.

That's a bit small to accommodate all the world's alphabets, right? Indeed. Let's also consider that 'A' and 'a' are two different letters for a computer - lowercase and uppercase characters are not the same. Add periods, commas, and other punctuation marks.

It quickly turns out that not much space is left for just letters.

That's why ASCII encoding contains only Latin alphabet letters. You can't write in other languages in ASCII because there aren't enough characters. What now?

Well, you see, so that Poles and other nations, like Chinese or Japanese people, who have quite a lot of these characters because each word can be a different character, meaning like a different letter for us, wouldn't feel disadvantaged, new encodings started to appear. Many encodings. Way too many. We won't talk about them. We'll skip straight ahead to times of clarity.

6.5 And Then UNICODE And UTF-8 Enter All in White

Currently, however, the standard is something called UTF-8. It's a UNICODE character encoding system that uses 1 to 4 bytes for storage. Oh, important information. Up to 4 bytes. 4 bytes, how much was that? 1 byte, 8 bits, 4 bytes, 32 bits. And what does it mean that UTF-8 is a UNICODE character encoding system? UNICODE is our whole mapping of which letter/character corresponds to which number, and UTF-8 is a way of writing this, converting to bits, because when you have slightly more than 1 byte, the matter becomes less obvious, hence these characters/mappings can be presented differently. UTF-8 is one of the ways to do this. Going back to the bytes topic…

32 bits means 32 ones or zeros, so using them we can write many different numbers, therefore also different characters. Now this becomes quite a lot, because it gives us about 4,294,967,296 possible characters. Quite a lot, right? It's still less than the 70 million that Sasin wasted on elections that didn't happen, but that's nothing. Even if we put all those Asian characters in, there will still be plenty of space left. Beautiful, idyllic. A dream.

A dream, because in reality, since RFC 3629, UTF-8 handles at most 2,097,152 characters. This is due to various historical circumstances, implementation details, and other strange things that you don't need to worry about, neither you, dear reader, nor I, but rather big brains like Ken Thompson and company, somehow it turned out that some bits are reserved for special purposes, some bytes must have a specific format to know various useful things, and so on.

And what's this whole RFC about? Generally, these are standards that certain brains set. On what basis? On whatever they decide. Similar to ASCII - because that's how it is. In general, I'm simplifying, and essentially when making various decisions, decision-makers are guided by more rational arguments.

That many characters is enough for us on a daily basis. Currently in UNICODE we have defined standardly about 143,859 characters. So we even have some reserve in case we need to add new characters later.

Or rather `codepoints`, but let's simplify, not delve into it and just say characters. What is a codepoint? Usually when discussing various encodings, instead of 'character', the term codepoint is used. Small difference. From your perspective, it doesn't really matter much.

So: remember that when UTF-8, then generally also Unicode. Keep these two terms together in memory, but UNICODE is not the same as UTF-8 and that's very important.

Plus another note. There's also something called UTF-16. How does it differ from UTF-8? In word length. Meaning in UTF-8 one word has 8 bits, in UTF-16 it has 16 bits. And that's it. There's also UTF-32. Here there's always one word, such 32-bit.

Just to be clear - they have the same number of bytes in the upper limit, meaning maximum 4, and in the case of UTF-16, minimum 2 (well, because 16 bits). What is a word? It's not a word like from the dictionary. Word, meaning machine word, is kind of like a byte, but not quite. A way to group bits into X pieces simply.

In UTF-8 some characters can be expressed with 1 word, meaning 1 byte, in UTF-16 the smallest size is 2 bytes and in UTF-32 it's already 4 bytes. Meaning regardless of what character we use, e.g., `A`, meaning 65, meaning something that fits in 1 byte, with UTF-32, the computer will still write everything in 4, wasting quite a lot of memory. In UTF-16 it will write in 2, in UTF-8 in one. More economical. But UTF-32 is easier to find in memory because you know that each character is 4 bytes and that's it. And in UTF-8 there are different lengths, one character will have 1 byte, another 4, well and try to guess here, human, what is what, what ends when and where, but somehow we manage with it.

You don't need to worry about this too much, but somewhere there maybe remember.

UTF-8 is 100% compatible with ASCII - text in ASCII is valid UTF-8, but UTF-8 does NOT have to be valid ASCII. This is very important! Why? Because UTF-8 mainly contains characters that we can't fit in one byte, making them absent in ASCII. Therefore, ASCII is a subset of UNICODE.

By the way, interesting fact - when you write some messages on Facebook and send emojis, they are often encoded in UNICODE too, they have their specific numbers!

6.6 Summary

Let's summarize what we managed to do and learn.

We have in Python a function called **print**, which, attention, prints text on the screen. This text in reality is not text for the computer, but nothing else than a sequence of zeros and ones, because the computer doesn't understand anything else, due to how it's built - voltage or lack/low voltage - that's all it really understands.

Therefore, something called the binary system was created. It's a counting system, slightly different from decimal, where we express numbers using two digits and only that. It's somewhat more sprawling compared to decimal - writing the same number as in decimal takes up more space, you could say, but generally it's not some complicated concept, it can be grasped.

And now having something that the computer can understand, meaning the **binary system**, based on two values, we can build something on that.

We, as clever people, built something called **character encoding**. Well, we figured out that in certain cases, a given number in binary system would mean not a specific number but e.g., a character.

And thus was born one of the first more popular encodings, namely **ASCII**. ASCII was cool, but it had this flaw that the creators didn't anticipate many characters, let's say. You know, Uncle Sam, center of the world, didn't think about other nations, only about characters from their alphabet.

So something new came along, which I use to this day, which is somewhat better - **UTF-8**. UTF-8 is a way of encoding **UNICODE** characters. Unicode is a kind of 'mapping' of numbers to given characters. To avoid problems with backward compatibility, meaning so that old texts and programs would work on new computers, UTF-8 is **backward compatible** with ASCII, meaning text in ASCII is also valid UTF-8 text. In the other direction, not necessarily - not every UTF-8 is valid ASCII.

I'm probably boring you a bit, aren't I? I warned at the beginning - there will also be some theory and other things because this won't be just a book about Python. Although I admit honestly that I personally find all these topics very exciting, interesting.

It's amazing to me what we've created as humanity and how all these processes work. Beautiful thing. I hope that at least partly I'll manage to interest you, reader, during this reading, with such topics - not just Python itself, but computer science, science in general.

On the other hand, I think that such an approach that I present here - discussing a broader scope, some history and theory, and not just dry saying, "Oh here you have print and it prints text." is much better. It gives you insight into the fundamental theories that lie at the feet of what you'll use every day. You'll know the tool and its construction, application, you'll be aware. In my opinion, this is necessary to be a good programmer.

We'll move on to tasks/questions soon. Besides them, I'd like you to play a bit yourself after each chapter with what I write about - we're talking about print, do some printing yourself. I know it seems boring, but do it. Please. Plus, you can Google a bit more and delve into the topics discussed here. It will help.

6.7 Tasks and Questions

Some will be super basic, but answer them anyway. Well. If you don't completely know the answer, don't worry, read some piece again if necessary, try to think.

Best to take a piece of paper and write down your answers to questions that don't require programming on it. This will make you remember better. Formulate the answer based on the text. Later I'll give you the answers.

1. What function in Python is used to print text to the screen?
2. Does this function print anything besides the entered text, or not? Hint: what will happen if you use it again to print something new on the screen? Will the text be on the same line?
3. How does the computer know when to start printing on a new line?
4. What is the binary system?
5. What is a bit? And what is a byte?
6. What length is a byte currently? Were bytes always this length?
7. What values does a computer understand at its very foundations? Why?
8. What's the deal with ASCII? What is it?
9. How does a computer internally represent the text you type?
10. And this whole UTF-8 - what is it?
11. How many characters approximately (order of magnitude) can be represented using the two encodings we talked about in this chapter?
12. Convert the following numbers from decimal to binary: 5, 10, 32, 127, 256.
13. Now in the other direction, from binary to decimal: 0000 1101, 1000 0000, 0010 0100.
14. Are these two character systems we discussed backward compatible? In both directions? That is A with B and B with A? Or only in one direction?
15. What's the difference between UTF-8 and UNICODE? What is what?
16. Differences between UTF-8, UTF-16, and UTF-32. Which uses the least memory usually? Which uses the most? Why sometimes it's worth choosing the less memory-optimal variant?

You'll find the answers on the next page.

6.8 Answers

1. The print function.
2. Yes, it prints an additional newline character at the end of our text, which means that if we print something new, it will be on a new line.
3. The computer knows to print text on a new line if it encounters a special character, known as the newline character. It doesn't display it to us, but interprets it itself.
4. It's a number system using 2 digits - 1 and 0.
5. A bit is the basic and smallest unit of information used to write binary values - it's such a basic particle that takes one of two values. A byte is nothing else than 8 bits.

6. Currently, it’s generally accepted that a byte is 8 bits. It wasn’t always like this, before the spread of the 8-bit convention, you could find bytes of completely different lengths.
7. A computer, at its very foundations, understands only two values. 1 and 0. Nothing else. Everything above is human abstraction. This results from how it’s built - the binary system is based on the computer operating on two values: voltage and lack/low voltage.
8. ASCII is a character encoding system. A kind of translation where we assign appropriate numbers to specific characters. Something like the ciphers created in childhood. We simply agree that X means Y.
9. The same as everything else - using 1s and 0s, meaning numbers. Then it translates these numbers into specific characters.
10. UTF-8 is a UNICODE encoding system. So kind of like ASCII, but newer. Allows representing more characters and so on.
11. ASCII - 256, UTF-8 - 2,097,152
12. I don’t want to do this.
13. This either.
14. Yes, UTF-8 is backward compatible with ASCII. ASCII is not compatible with UTF-8, meaning any valid text encoded in ASCII will be valid in UTF-8/Unicode, but not every text in UTF-8 will be valid ASCII.
15. UTF-8 is a way of encoding UNICODE characters. UNICODE is the mapping of numbers to characters.
16. UTF-8 uses variable length encoding (1-4 bytes), UTF-16 uses 2-4 bytes, and UTF-32 always uses 4 bytes. UTF-8 usually uses the least memory for English text as most ASCII characters only need 1 byte. UTF-32 always uses the most memory but provides constant-time character access since every character is exactly 4 bytes.

7 Chapter 5: Variables

7.1 Let's Start with This Code

Let's begin with this code:

```
#include <stdio.h>

void main() {
    short a = 1;
    a += 1;
    short b = 4;
    char z = ##### SECRET #####;
    printf(z);
    int c = 4;
    int h = 5;
    printf(c);
}
```

Try to analyze this code on your own and think about what's happening here. It's not that difficult. Basically, we've declared several variables - `a, b, c, h, z` and printed `z` and `c`.

We'll discuss the rest below, but try to figure some things out on your own first!

Now, here's a little trick. We'll look at what code the compiler generates from this. In this case, we're using `x86-64 gcc 9.3`. Let's see what assembly instructions our compiler produced for the processor. Due to size, the code is on the next page.

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     WORD PTR [rbp-2], 1
        movzx   eax, WORD PTR [rbp-2]
        add     eax, 1
        mov     WORD PTR [rbp-2], ax
        mov     WORD PTR [rbp-4], 4
        mov     BYTE PTR [rbp-5], 102

        movsx   rax, BYTE PTR [rbp-5]
        mov     rdi, rax
        mov     eax, 0
        call    printf

        mov     DWORD PTR [rbp-12], 4
        mov     DWORD PTR [rbp-16], 5
        mov     eax, DWORD PTR [rbp-12]
        cdqe
        mov     rdi, rax
        mov     eax, 0
        call    printf
        nop
        leave
        ret
```

Whoa. What's going on here? Don't worry, let's break it down piece by piece.

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
```

Let's skip this part - it's not relevant to this specific example.

```
        mov     WORD PTR [rbp-2], 1
```

This is what interests us. This is equivalent to our: `short a = 1;` And what's happening here? Let's use some magical illustrations to help!

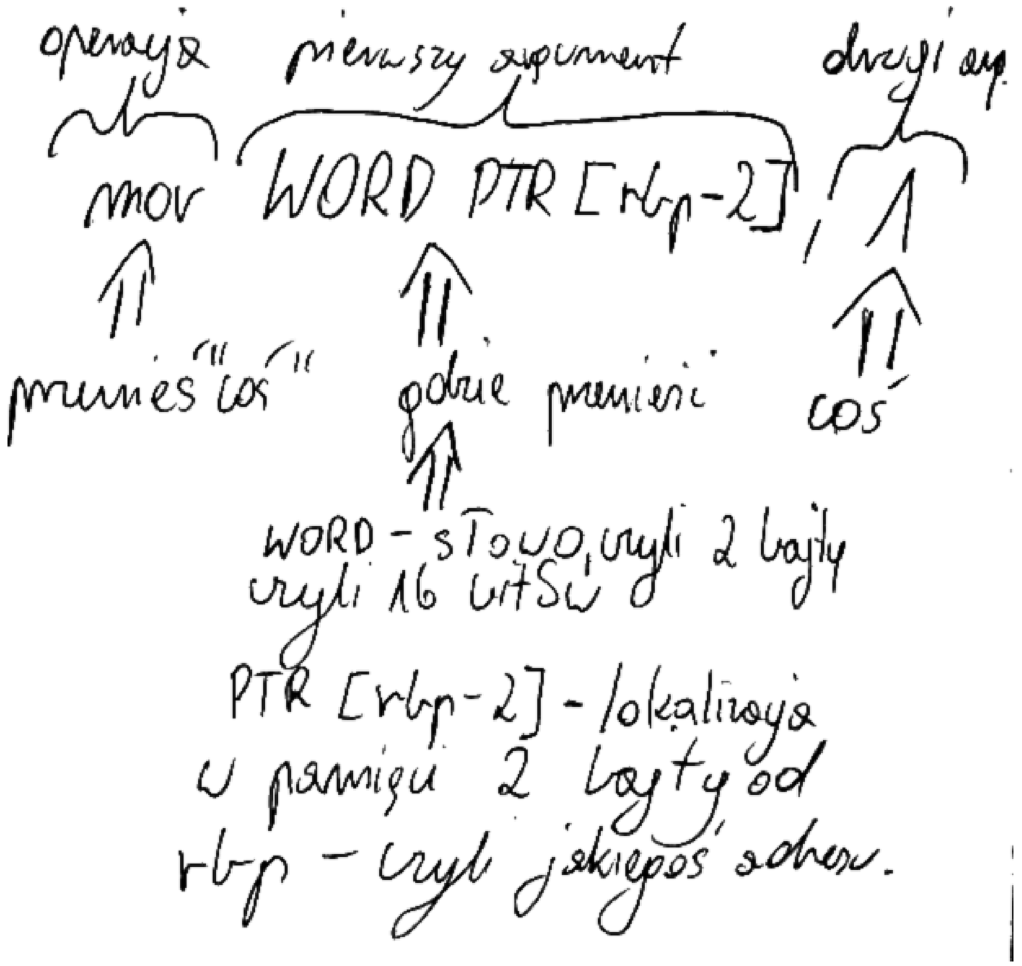

operacja
pierwszy argument
drugi arg.
mov
WORD PTR [rbp-2]
, 1
przenieś "coś"
gdzie przenieść
coś
WORD - słowo, czyli 2 bajty czyli 16 bitów
PTR [rbp-2] - lokalizacja w pamięci 2 bajty od rbp - czyli jakiegoś adresu.

Figure 1: Detailed breakdown of the above instruction

Okay, I've broken it down a bit in the illustration. Now let's go into deeper explanations.

Our declaration `short a = 1;` simply means creating a variable `a` of type `short` and assigning it the value `1`. `short` is a small integer, in this implementation it's 16 bits, or 2 bytes. In Python, a similar (at some level) notation would simply be `a = 1`. So we're creating a variable `a` with value 1. This is how it looks in C. Time to go one level lower. And what's one level lower? What language? Remember from previous chapters or Google it.

`mov` tells the processor to move the value of the second argument to the location specified in the first argument.

The first argument is the entire expression `WORD PTR [rbp-2]`. It means nothing more, nothing less than that at (PTR) the address, specifically address [rbp-2], which is `rbp` minus two bytes, we have a `WORD`, which is a word, and a word in this implementation is 2 bytes, or 16 bits. What `rbp` is - let's not worry about that now, it's not very important. Imagine it's some address, reference in memory, pointer to something, whatever.

And as the second argument, we have the value that needs to be placed there, which is `1`.

Take your time to analyze this, it's not that complicated. Pay attention to the fact that the type `short`, which has a size of 2 bytes in this implementation, or 16 bits, somehow sounds similar to the size the compiler used in the first argument, there's `WORD` - also 2 bytes and 16 bits. Coincidence?

Go back a bit and look at the rest of this code, especially the declarations of the next variables, their types, and the fragments with X PTR [rbp-XD].

After all this, a light bulb should go off in your head. Answering the question from the beginning of the chapter - how does the program know when to stop reading? Well, during compilation, something like `a` disappears. Its occurrences are replaced by something like `WORD PTR [rbp-2]`. Having this, the program knows exactly when to stop reading and when to start, because we have both the address and the number of bytes to read.

Spend some time on this, think about it. It doesn't have to click immediately. First analyze the entire code. Take it easy. Only then move forward.

7.2 Everything's Fine, But Python is Different

Now here's a very important note. Throughout the examples in this chapter, I've been using code from the C language and describing the process that happens there during compilation and so on.

In Python, however, things look a bit different, as I've already mentioned. This comes from the different natures of these two languages - statically compiled C and dynamically interpreted Python. The knowledge I'm sharing here is universal, because while at the higher level, which is just below the surface, Python doesn't look like this, at the lowest level it definitely does - after all, CPython is written in... C.

In Python, the mechanism of variable declaration looks a bit different (we'll talk about how another time) and so on, but the logic is, let's say, preserved somewhere. However, it's easier for me to explain using C and Assembly examples. Why? Because in Python, during interpretation, millions of different optimization tricks are applied and various strange things happen there. For example - during Python's startup alone, about 50,000 different objects, variables, and other things are initialized! Not bad, right? That's why for now we're operating on "simpler" C. Later we'll probably get into these Python optimization threads, but for now it's not necessary. So when you sometimes hear compiler instead of interpreter, you might think something got mixed up and so on, but don't worry.

Also, don't worry at all if the above isn't obvious to you, this fragment with the code. Take it easy. Spend some time on it, it's not that simple! Think about it, analyze it on your own, even Google it. The fact that you don't understand what's happening there after a second doesn't mean you're unintelligent or stupid. So keep going!

7.3 Another Difference

While we're talking about Python's differences, let's say a few words, literally two sentences, about how Python, as an interpreted language, is different from compiled languages, but only apparently.

In the traditional model, as we've already discussed, we have code, then this code is compiled into machine code, then run. In Python, we have an implemented interpreter that executes/interprets our code. That's how it works, at least apparently.

If we look a bit deeper, it turns out that... Python is interpreted, but compilation also happens here? But how, one might ask. What's going on?

Well, the Python code you write is also compiled, but not to machine code. Python compiles to `bytecode` that's understandable for the Python interpreter, something like a virtual machine in Java (JVM), but differently.

Then, our compiled bytecode is executed by the Python interpreter, and the Python interpreter is nothing else but another code compiled to machine code, that is, a regular program. In short, Python is a compiled interpreted language, so to speak.

Have you ever wondered what and why `.pyc` files are created after you run your code? This is a form of optimization and remembering by Python of the intermediate compilation step. Python looks at the source file, based on it calculates some sum from this file, or 'number', after all, every file underneath is nothing else but some very long binary string. Binary strings can be translated, calculated into a regular number, in simplification. So Python does this underneath, looks if a `.pyc` file exists, if not, it creates it. Then it checks if this number, which is unique for each source file, is the same. If it is, it doesn't perform the compilation step again, it immediately jumps to interpreting.

If we make any change in the code, Python will catch the change, because this 'number' will change and before interpreting, it will conduct the 'compilation' process again.

7.4 Memory Management

One might ask, how does Python know when to 'delete' variables from memory? Because we know when to allocate. How does Python know when a variable is no longer needed and can free the allocated memory? If Python didn't know and didn't do this manually, didn't conduct so-called `Garbage Collection`, then every run of our program would permanently clog, at least until restart, our computer's RAM. How does it happen that this doesn't occur in most cases?

Well, the Garbage Collector in Python, which cleans up variables we no longer need and frees memory for reuse, keeps a list. In this list, it keeps the reference count for each object. If this count reaches 0, it means that the object is no longer needed, because nothing refers to it, so the memory can be freed.

What about when one object refers to another, but nowhere else? Then we have a loop of dependencies, so to speak. That is, theoretically the reference count is greater than 0, but the object isn't being used.

Here comes the system for detecting reference loops. Python is able to detect this and then also free the memory. Linked list and such. Read up on it yourself.

It's also worth adding that Python is a clever beast. It has an optimization mechanism that most often checks newly created objects. If an object is fresh, there's a good chance it won't be needed soon. Old objects that have survived until now have a good chance of surviving further. Those who have, will have more added to them, those who have nothing, will have it taken away. Or something like that. It's like with fashion and trends. New ones pass, but the old core continues.

7.5 Summary

Due to the fact that quite a lot of information has flowed onto your heads in this chapter, my dear readers, I've decided to end it here, somewhat earlier than originally planned.

Let's remind ourselves of what I wrote about in this chapter.

Our computer is quite good at remembering things, much better than our brains, so it's worth taking advantage of this. Usually during programming, we use **RAM** to remember certain things for some time.

By the way, a note - the expression `RAM memory` is a linguistic pleonasm, because RAM means `Random Access Memory`, so writing RAM memory is like writing butter butter. But let's get back to the topic - we use RAM. When?

We do this, for example, by using **variables/constants**, which are nothing else but some **name, alias** that we create for what we want to remember, through which the computer knows where in its memory, at what address, to be more precise, to look for a given thing, and it's easier for us to type and remember `first_name` as a variable name/reference/alias instead of `0xA1FBA`.

When creating these `aliases` or **naming our variables, we must follow certain rules**, such as not starting them with digits. Variable names should be descriptive but short, similar to the entire code. This has fundamental significance when it comes to the readability of the code we create and its quality.

Besides the binary system, we also have something called the **hexadecimal system**, which we use to write more concisely what would take us much longer in binary. It all converts on a similar principle as from decimal to binary.

The computer associates an **address**, variable size depending on what's inside. Based on the Assembly/C code, more or less we learned how it looks underneath. But just, we examined the process in C more precisely, in **Python it all looks a bit different, because it's interpreted**, but the general mechanics remain similar somewhere, hence it's good to know it.

7.6 Exercises and Questions

Now it's time for questions. Remember, some of them will require searching for information on the Internet or general knowledge. That's not a bad thing!

1. What types of memory does your computer have that we discussed? List their characteristics and which is used to store what kind of data?
2. How do you declare a variable in Python? Declare several in the interpreter.
3. What rules do you know about names we can give to variables in Python?
4. Polish characters in variable names are a good idea. Do you agree with this statement? If yes, why? If no, why?
5. In the chapter, there's a piece of code with a function called `redirect_logged_in_user`. Translate this entire fragment line by line and describe in your own words what it does, what each piece of this code does, or what you think it does. In the chapter text, you already have some description, but do it now independently and go deeper into the details.
6. What does computer memory look like, figuratively speaking? How does the computer move through it, simplifying?
7. What is the hexadecimal system?
8. Convert several numbers from decimal to hexadecimal. Which ones? 2, 8, 19, 32, 111.
9. How much RAM does your computer have? How many bytes is that? Pay attention to whether we're talking about bits or bytes ;) How to express this number in hexadecimal? And in binary?
10. How does the computer know where to look for the value we saved for a given variable?
11. How does the computer know when to stop reading the value at a given address?
12. In the code from point `8.6`, there's a fragment shrouded in mystery - the value for variable `z` isn't expressed. By analyzing the assembly code below, try to guess what value was assigned there. I'll just hint that the `char` type is nothing else but some `character`, that is, a character. Hint: remember a bit about how we represent/encode characters/text.

7.7 Answers

1. I'm talking about RAM and hard disk memory. Their characteristics - look them up yourself. In short, RAM is usually (generalizing) faster but less persistent, disk the opposite.
2. `variable_name = "value"` for example.
3. So far we've only talked about not starting with digits but with letters or underscore.
4. It's not a good idea. Why? We shouldn't necessarily write code in Polish, except for really few exceptions.
5. I won't add details here, let this be a small challenge ;)
6. More or less you can think of it as a sequence of cells placed in a straight line, containing ones or 0s.
7. A number system based on 16 as its base.

8. Again - do it yourself.
9. See above.
10. By the address that it manages 'underneath' and reads from there - at this specific location in memory.
11. We talked about this in `8.6`.
12. If we analyze the line `BYTE PTR [rbp-5], 102` we can conclude that this `char` definition says something about the number 102. 102 in ASCII/UNICODE is nothing else but `f`.

Remember that you can post your answers on GH here - https://github.com/grski/junior-python-exercises, and I'll check your solutions and give feedback. More about this in the 'Interactive Part' subsection.

8 Data Types

In the previous chapter, we talked about variables. While on this topic, it's worth mentioning the types of data that our variables can store in Python. We'll discuss this specific language, but similar divisions exist in most languages.

As I wrote in previous chapters, Python doesn't require us to define types for our variables - it doesn't have static typing. Remember what this meant and answer the question - what does the lack of static typing mean? What is dynamic typing? What are its advantages/disadvantages? Look in the book, maybe in the answers to previous chapters. Do this now.

Despite this, it's good to know what types we typically divide variables into. Why? Because Python uses them too, it just somehow guesses what type we used. Depending on the type, different operations can be performed on the variable. When we think about it, this is logical, because even though underneath it's all the same - binary code, on certain fragments that we interpret as X, we want to perform only operations from set Z, and when we interpret G, only from F. Speaking more simply, if we mark something as text, we'll treat it differently, or apply different modifications than when something is a number. On numbers, we can perform arithmetic operations, while in text we can search for our name, for example. Different operations depending on the type. Logical, right?

So in Python, we distinguish the following basic data types:

1. Numbers
2. Strings
3. Bytes
4. Boolean type

Among the more complex ones, we have:

1. Lists
2. Tuples
3. Dictionaries
4. Sets

Let's discuss them all one by one. Let's start with numbers, as it will probably take the longest.

8.1 Numbers

8.1.1 Brief Characteristics

So, my dear readers, in Python we distinguish three main types of numbers: integers, floating-point numbers, and complex numbers, or in order: ints, floats, complex.

What does this mean?

8.1.2 Integers

This is probably simple, right? `1, -1, 5, 0, 938, -24861` are examples of integers, or simply `ordinary` numbers without any decimal points, peculiarities, or inventions. I won't elaborate too much on them because there's no need.

Or is there?

Note that in the examples above, we have a negative number, less than zero. In the case of text, it's simple to write - we put a minus in front and we're done. As humans, we're taught to interpret this as a negative number. But how does a computer do this?

8.1.3 Example Way of Representing Negative Numbers

Let's recall previous chapters and what we talked about there. I showed, among other things, how a computer stores numbers in memory and how it represents them, namely using the binary system, bits, bytes, those things. Now, the question arises: how then does the computer indicate that a given number is negative? After all, it can't put a `minus` in memory in some way.

Well, I'll show you how it looks, again, in C. There were many ways and ideas to solve this problem, and there still are, but let's discuss just one. Let's assume we're operating on some type that is exactly 1 byte in size. This means it's 8 bits long. How many values/numbers can it handle at most? Please answer this question, calculate.

Good, having 8 bits, we have 8 zeros and/or ones at our disposal. So we can represent a maximum of 256 values, right? That is, for example, numbers from 0 to 255. Well, not exactly!

In the default case, we'll have 256 values at our disposal, right, but from a different range: from **-128** to **127**. This can be determined by the formula: $(+/-)2^7 - 1$

Where does this change come from? Well, we take one byte to indicate whether a given number is positive or negative, in a nutshell. In C, if we know we're not interested in negative values, we can tell the compiler to shift the negative range to positive. Variables with sign vs variables without. Signed variables vs unsigned variables.

By the way, while we're at it, I'll add another interesting fact. Did you know that even the way of writing the order of bits in memory is conventional? What does this mean? Well, some people couldn't agree on what's better - writing the highest-value bit first or last. Hence we have two standards: big endian and little endian. What does this mean and how does it look in practice? Simple thing.

Let's assume we're talking about Big Endian. We want to write the value 0x4A3B2C1D at address 100. It would look like this.

100	101	102	103
4A	3B	2C	1D

And Little Endian?

100	101	102	103
1D	2C	3B	4A

So the opposite. It's generally about which to write where. This makes a difference when calculating/reading these values. Which is better? Big Endian will probably be easier to understand, as it's analogous to the notation we use daily in the decimal system.

Different processors have different conventions, fortunately you don't have to worry about this in your code - the Python interpreter will do it for you.

8.1.4 Floating-Point Numbers and the Inaccuracy of Their Representation

What are we dealing with here? Nothing other than numbers with a `decimal part`, speaking in simple terms. And that's basically it. If we see a number in Python where there's a dot - e.g., `1.0`, we need to know we're dealing with a floating-point number. Why is this important knowledge? Well...

Here, unlike with integers, there are peculiarities, and big ones, but underneath. It's a longer topic, but it basically comes down to the so-called inaccuracy of floating-point number representation in the binary system. Yes, yes. Clear, right? I'll just say mysteriously that you should always and everywhere remember that using regular floats/doubles for precise calculations or storing information about money is not a very good idea, because sometimes `0.1+0.2 != 0.3`. Why? Because try to represent, for example, 1/3 exactly using powers of two. Hard, right? But what am I getting at?

Let's consider a simple program in C (this issue concerns practically every language):

```
#include <stdio.h>

int main()
{
    float example_float = 0.1;
    if(example_float == 0.1)
    {
        printf("Equal");
    }
    return 0;
}
```

Simple code, right? I think everyone should understand it if they know at least the basics of programming. The expected result of running this code for a large part would be printing 'Equal' in the console, right? I thought so too at the beginning. Check for yourself what happens when you compile and run the code.

Surprisingly, "Equal" didn't display. Why? Did something go wrong? The numbers appear to be the same, because here 0.1 and here 0.1, what's going on? Hmm, maybe the variable was written incorrectly. Let's print it and see.

```
printf("%f", example_float);
```

Add this line of code after the finished if. Run the code... And what? Here's the result:

```
0.100000
```

Wait. So something is wrong in our program, right? Because `example_float` is equal to 0.1, right? Well, no.

We can't see it here because the precision is too low, but let's force the `printf` function to display our float with greater precision than default, because as you can see, printf by default displays only 6 decimal places.

```c
#include <stdio.h>

int main()
{
    float example_float = 0.1;
    if(example_float == 0.1)
    {
        printf("Equal");
    }
    printf("%.16f", example_float);
    return 0;
}
```

Gives us

```
0.1000000014901161
```

A slight modification of our code and everything is clear. Our `example_float` is not exactly equal to 0.1, but a bit more. Why?

Everything results from the fact that the computer 'operates' in binary language. This means that when creating numbers, only powers of two are available, multiplied appropriately by 1 or 0, which can be summed (in great simplification, we talked about this already). So it's no wonder our float looks like this. Try to build exactly 0.1 from these numbers `{..., 1/128, 1/64, 1/32, 1/16, 1/8, 1/4, 1/2, 0, 1, 2, 4, 8, 16, ...}`. It can't usually be done perfectly. Theoretically, in an imaginary world where we would have an infinite amount of memory at our disposal and an infinite amount of time, we could get infinitely close, sometimes even achieve, any number. But if you want to know more about this, read about limits or recall from high school mathematics.

Hence this inaccuracy - it results only from how floating-point numbers are represented in computer memory. While in most cases, satisfactory accuracy can be achieved using a finite amount of memory, there are cases where this accuracy won't be sufficient.

For such cases, we have special libraries or perhaps a special approach that deals with the topic differently, but it's worth knowing about this. Therefore, if we're writing a program that has anything to do with money, it's worth thinking twice before using a float or double. Maybe it's better to keep zlotys in a separate int, and grosze in a separate one? Who knows.

Solutions to many problems related to floating-point numbers can be found in the `Decimal` and `Fraction` modules.

To read:

1. https://docs.python.org/3/library/decimal.html
2. https://docs.python.org/3/library/fractions.html

8.1.5 Complex Numbers

Very rarely encountered, but sometimes when someone is doing some scientific calculations or other strange things, this knowledge might be useful - these are numbers consisting of a real part and an imaginary part. Mathematical topics. If you don't know what this is about, don't worry.

We define them like this: `test = 21 +3j` or `some_complex_number = complex(32, 3)`. And we can perform the same operations on them as on numbers - division, multiplication, addition, and so on. Sometimes useful.

And that's it. For now, that's all you need to know about things we qualify as numbers in Python. There are also decimals or rationals, but we'll talk about them another time!

8.1.6 Operations on Numbers

As I mentioned regarding complex numbers, we can perform various basic mathematical operations on numbers, and they will work more or less as we would expect. I'm talking about addition, subtraction, multiplication, division, integer division, modulo operation, and other basic arithmetic operations.

Operators that Python understands are:

Operator	Action
*	Multiplication
**	Exponentiation, so 2 ** 3 means two to the power of three.
/	Regular floating-point division
+	Addition
-	Subtraction
//	Division without remainder, so 5 // 2 equals 2, -11 // 3 equals -4 (note negative numbers)
%	Remainder of, so for example 5 % 2 is nothing other than the remainder of dividing 5 by 2, which is 1

It's also worth remembering that if we have several different types of numbers in one expression, Python will cast the result of the entire expression to the most complex type. What does this mean? The result of `1.0 + 1` will be not `2` but `2.0`, the result of `1 + 1.0 + 3 +0j` will be `5+0j`. The order of complexity is thus:

Integers -> Floating-point -> Complex.

It's worth remembering this, as it carries certain consequences - just watch out for those dots, because sometimes you can get caught out.

Additionally, I'll mention a syntactic sugar that allows us to shorten the notation of popular operations. It looks like this:

```python
foo = 2
foo = foo + 2 # this line will be equivalent to the one below
foo += 2
```

8.1.7 Numeric Conversions

Numbers can be 'converted' between each other, or we can perform a kind of `type casting` as we would say in other strongly typed static languages. How? Examples below. Analyze them a bit yourself.

```
>>> int(1.3)
1
>>> int(1.6)
1
>>> int(1.9)
1
>>> int(1.4444)
1
>>> int("1")
1
>>> int("5")
5
>>> int("52")
52
>>> int("-52")
-52
>>> float(1)
1.0
>>> float(3)
3.0
>>> float("2")
2.0
>>> complex(2)
(2+0j)
>>> int("1.3")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1.3'
```

Play around with the `int`, `float`, `complex` functions. Then describe what each of them does. What values they accept, what causes errors. The last example `int("1.3")` caused an error. Translate it and try to explain what happened here.

It's worth noting one important thing that Python provides out of the box, which not every language necessarily has. All these functions understand that `"1"` is 1. In other languages, it's often different. Why? Well, during type casting, an operation often occurs directly on values in memory. Let's recall that fragment about how a computer stores text in memory. ASCII, UTF-8, Unicode, and so on. Go back to previous chapters if you need to.

Exactly, so how? As numbers appropriately mapped later. Therefore, in other languages, instead of interpreting `"1"` as 1, it's often cast to the numeric value that hides behind the character 1 in a given character set/encoding. In our case, "1", that is, one in text is marked not as 0b1 but as 49 which is 0b110001 or 0x31.

By the way - a small note. For quick conversion, you might be interested in the functions visible in the snippet below.

```
>>> ord('1')
49
>>> bin(49)
'0b110001'
>>> hex(49)
'0x31'
>>> oct(49)
'061'
>>> chr(49)
'1'
```

So ord, bin, hex, oct, chr. Play around and read about them. Where? In the Python documentation. Best in English. Summarize your conclusions and the effects of playing by writing an article in which you describe what each function is about, briefly characterize each type. Give examples for which functions don't work and guesses why. Alternatively, use the help function, e.g., help(int).

Additionally, I'll show you a small trick:

```
>>> dir(float)  # alternatively: dir(1.0)
['__abs__', '__add__', '__class__', '__coerce__',
 '__delattr__', '__div__', '__divmod__', '__doc__',
 '__eq__', '__float__', '__floordiv__', '__format__',
 '__ge__', '__getattribute__', '__getformat__', '__getnewargs__',
 '__gt__', '__hash__', '__init__', '__int__', '__le__',
 '__long__', '__lt__', '__mod__', '__mul__', '__ne__',
 '__neg__', '__new__', '__nonzero__', '__pos__', '__pow__',
 '__radd__', '__rdiv__', '__rdivmod__', '__reduce__',
 '__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__', '__rmul__',
 '__rpow__', '__rsub__', '__rtruediv__', '__setattr__',
 '__setformat__', '__sizeof__', '__str__', '__sub__',
 '__subclasshook__', '__truediv__', '__trunc__',
 'as_integer_ratio', 'conjugate', 'fromhex', 'hex',
 'imag', 'is_integer', 'real']
>>> dir(str)  # alternatively: dir("text")
['__add__', '__class__', '__contains__', '__delattr__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__getnewargs__', '__getslice__', '__gt__',
 '__hash__', '__init__', '__le__', '__len__', '__lt__',
 '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '_formatter_field_name_split',
 '_formatter_parser', 'capitalize', 'center', 'count', 'decode',
 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index',
 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle',
 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition',
 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

So the `dir` function. The dir function is a function that returns all available methods/attributes of a given object.

For now, don't worry about those that start with `__` or `_` and focus on those that start with normal letters. What are they though? Methods starting with `__` are so-called Python Magic Methods/Dunder Methods/Magic Methods. This is something we'll talk about later, but these are special types of methods/functions of a given object that are supposed to fulfill specific roles. Those that start with a single underscore `_` are private methods.

Python doesn't have encapsulation, which means that generally when we add some attribute/method to a class/object, we can't very effectively prevent others from calling it, even if we want the user not to have the ability to do so, because, for example, a given method is only auxiliary, **private**. The convention thus says that we should put an underscore before private variables, methods, and we as programmers shouldn't use such unless it's inside the definition. We'll talk about this later. In the meantime, you can Google about this whole encapsulation.

To summarize: using `dir` you can check what you can do on a given object, what methods/functions it has, etc. Useful.

Play around with this on the types you know.

8.1.8 Examples of Basic Operations on Numbers

```
>>> integer = 4
>>> second_integer = 9
>>> first_float = 2.0
>>> second_float = 6.0
>>> help(integer.conjugate)
>>> help(integer.numerator)
>>> integer.numerator
4
>>> integer.numerator
4
>>> integer.as_integer_ratio
<built-in method as_integer_ratio of int object at 0x7f5443904150>
>>> integer.as_integer_ratio()
(4, 1)
>>> integer.bit_count()
1
>>> integer.bit_length()
```

```
3
>>> integer.imag
0
>>> integer.real
4
>>> 4 * 2
8
>>> 4 ** 2
16
>>> 4 // 3
1
>>> 4 / 3
1.3333333333333333
>>> 5 // 3
1
>>> 5 % 3
2
>>> first_float.is_integer()
True
>>> first_float.as_integer_ratio()
(2, 1)
```

8.2 Strings

8.2.1 Brief Characteristics

Strings/text, character sequences, so-called strings. It's simply text. Usually at least, because they can also be some kind of byte collections, but we'll talk about that another time.

In Python3, UNICODE and usually utf-8 reign for regular strings. Of course, you can use other encodings using appropriate methods, but for now, we probably don't need to worry about that. In Python 2, it was somewhat different, so I'll only mention this for historical knowledge and won't elaborate too much on how the process looked there, as this is already a somewhat archaic approach and all modern programming languages assume the use of UNICODE and utf-8.

Strings in Python are declared in the following way:

```
some_text = "foo"  # first way
another_text = 'bar'  # second way using ' instead of "
longer_text = """hahah
another line """
```

So it's simply text surrounded by `"` or `'`. Python allows the use of both double and single quotes. My practice is to prefer `"`. Technically speaking, the standard allows using either one or the other, as long as we don't mix them in one project. What does this mean?

If we already have some codebase/code/project, and we decide to use `'` instead of `"`, that's okay, let it be so, although it's not my preference, but let's not mix styles. The convention is to stick to one and have a unified style throughout the code. This means that if we start using single quotes somewhere, we should use them everywhere in that project. If double, then double. Did I mention that I prefer double quotes and consider them better? The creators of the code formatting tool - `black` - think the same, plus objectively double quotes have their advantages like better readability or easier use with English where we have a lot of `'` characters in text. No need to add an escape character.

Ah yes, the escape character. Let's talk about special characters. What if in our text, which we defined using `"`, we need to use that character too? Let's try.

```
quote = "chciałbym tutaj zacytować "paula coelho" ale nie wiem czy moge"
```

The given code won't work. Why? Python can't guess that you want to quote here and this character should be treated specially, not as usual. You need to tell it about this. How? Simple thing.

```
quote = "chciałbym tutaj zacytować \"paula coelho\" i wiem, że mogę"
```

Simply add \ before the character you want to treat in a special way. Now it might dawn on you why `"` > `'`. In English, single quotes often appear. If we use them to define strings, a problem arises in the form of having to often use the escape character. If we use double quotes, then less often. So the reason is laziness and code readability, which is also laziness. Yay!

Anyway, let's get back to the main topic.

Strings defined this way must fit in one line. If we want our text to be multiline in the code, we need to use triple quotes - so either `"""` or `'''` instead of one. This will cause Python to read not only to the end of the line but until it finds the closing character, which can be in another line.

It's also worth noting that for commenting in code, we use `#` for single-line comments and multiline strings as multiline comments.

```
class RedirectMixin:
    """
    Mixin that is used for the purpose of...
    """
```

Example above.

Remember to use `dir` on strings and review the basic methods that Python has in the standard library that allow us to manipulate strings.

8.2.2 Single Characters

Hmm, if strings are character sequences, what about single links in these chains? Well, imagine that you can iterate over text strings like over a list, plus we have access to their individual elements, to slicing, etc. NEAT!

8.2.3 Variables in Text

Besides simple strings that just contain hardcoded text, for example:

```
name = "Aryo"
```

There is the possibility of performing operations on strings that allow us to insert variables into text, add strings, etc. There are several ways to achieve this. Instead of elaborating, I'll just present them.

```
age = 23
name_and_age = f"Olaf {age}"
name_and_age = "Olaf {age}".format(age)
name_and_age = "Olaf " + str(age)
```

The first method is called f-strings. They are elegant. Beautiful. Awesome. Proper.

The second option is the format function.

The third is so-called concatenation.

Which one to use and when? F-strings ftw. Format is okay, and concatenation when you can't use anything else.

8.2.4 Using Variables in Text - Performance

I'm quite a big fan of f-strings in Python. I like them, they're elegant, readable, and simple to use. However, I was curious about how they perform under the hood, because, well, that elegance probably has some hidden cost. Nothing in life is free, right? I decided to check and compare different methods of string manipulation in Python in terms of performance.

In the competition were: f-strings, string concatenation (addition), the join() method, and the format() method. The % operator was not included in the comparison. Why? I don't really like it, to be honest. My personal preference. I believe it should be avoided for certain reasons. A relic of the past, we have better solutions today.

8.2.4.1 Testing Methodology I tested using Python's built-in timeit module, running commands from the terminal. All variables used in the modified string were defined and loaded before timing began. Each command was run in loops of 1,000,000 iterations, with each such loop run 3 times. From this loop, the shortest single iteration time was selected. Let's move on to the testing itself.

8.2.4.2 Comparison Let's begin then. Below is the code I used. Forgive the primitive variable names, but I wrote it completely on the fly.

```
python3 -m timeit -s "x = 'f'; y = 'z'" "f'{x} {y}'" # f-string
python3 -m timeit -s "x = 'f'; y = 'z'" "x + ' ' + y" # concatenation
python3 -m timeit -s "x = 'f'; y = 'z'" "' '.join((x,y))" # join
python3 -m timeit -s "x = 'f'; y = 'z'; t = ' '.join" "t((x,y))" # join2
python3 -m timeit -s "x = 'f'; y = 'z'" "'{} {}'.format(x,y)" # format
python3 -m timeit -s "x = 'f'; y = 'z'; t = '{} {}'.format" "t(x,y)" # format2
```

Everything is quite simple. I considered two options, one standard and another where the method lookup would occur in the setup, and in the measured time only its call would occur.

What do I mean when I say the method will be found using the . operator? Well, Python underneath stores class attributes/method names and so on in a hashed dictionary. So when we write `object.attribute`, underneath there's a dictionary lookup to see if such a thing exists in that class. This of course adds to the execution time because the lookup instructions take time, albeit very little, almost nothing, but still, plus the time needed to allocate memory and add elements to the dict underneath when constructing the instance. For certainty, I tested different cases. I'll note, however, that in production code, you should generally avoid this type of optimization, okay? At a junior level, it's rare that you'll be processing such large datasets and your code will require such performance that you need to do these things. Just a warning. Anyway.

I did the same with join and format. Here I considered two options - normal call with lookup and one without it.

And here are the results:

```
f-string: 10000000 loops, best of 3: 0.0791 usec per loop
concatenation: 10000000 loops, best of 3: 0.0985 usec per loop
join without lookup: 10000000 loops, best of 3: 0.112 usec per loop
join: 10000000 loops, best of 3: 0.144 usec per loop
format without lookup: 1000000 loops, best of 3: 0.232 usec per loop
format: 1000000 loops, best of 3: 0.264 usec per loop
```

8.2.4.3 Surprise I'll be honest, I didn't expect that f-strings would not only be an elegant solution but also the fastest! This makes me very happy. In second place was concatenation, join without lookup, join, format without lookup, format, and at the very end template string. Since the optimization I made is quite impractical and probably no one would create such monsters in code except for certain exceptions that perhaps should be written in C rather than Python, I'm not including the results without lookups in the ranking, which looks like this:

1. f-string
2. Concatenation
3. join()
4. format()

8.2.4.4 A Slightly More Complex Example I showed a simple example - inserting two variables separated by a space. What if we have more than 2 variables? Let's assume a case with 13 variables that we want to join with a space. Code:

```
a, b, c, d, e, f, g, h, i, j, k, l, m = [str(s) for s in range(13)]
# f-string
f"{a} {b} {c} {d} {e} {f} {g} {h} {i} {j} {k} {l} {m}"
# concatenation
"a + ' ' + b + ' ' + c + ' ' + d + ' ' + e + ' ' + f + \
' ' + g + ' ' + h + ' ' + i + ' ' + j + ' ' + k + ' ' + l + ' ' + m"
# format
"{} {} {} {} {} {} {} {} {} {} {} {} {}".format(
    a, b, c, d, e, f, g, h, i, j, k, l, m
)
# join
" ".join((a, b, c, d, e, f, g, h, i j, k, l, m))
```

I ran the code above analogously to the previous time.
I'm curious how the situation will look here.
Results:

```
join: 1000000 loops, best of 3: 0.352 usec per loop
f string: 1000000 loops, best of 3: 0.399 usec per loop
format: 1000000 loops, best of 3: 0.872 usec per loop
concat: 1000000 loops, best of 3: 1.13 usec per loop
```

Based on the previous results, they didn't surprise me much. Why? Let's start with what changed. Join jumped from 3rd place to 1st. Concatenation dropped from 2nd to second to last. Format to 3rd from fourth. Quite reasonable, why.

First place for join in such a situation is obvious - look at what we're doing - we're joining many strings with a common string, which is exactly what join was created for. I'm almost certain that underneath at the implementation level or even interpreter level, there are optimizations made for this, thanks to which join handles a large number of arguments very well. This makes me happy - again, the solution that looks most elegant in this case comes out on top.

Second place f-string. Here I wasn't surprised either. Why? Well, f-strings, originally they were slow, very slow - in the first implementation they were "compiled" to nothing else than a set of appropriate joins or formats, I don't remember. However, in the next implementation, f-strings got their own, optimized OPCODE in CPython, which allowed for significant savings and better adaptation of the C code that's underneath.

Why did format overtake concatenation? Well, I suspect. It seems to me that it's about evaluation. Perhaps Python, because strings are immutable in Python, each time it performed an addition operation on two strings, had to allocate a new piece of memory that would fit X characters, where X is the sum of the lengths of two strings, then copy them there to get the final value. Based on experience with how Python works, I'd bet that in our case, when we had code in the form a + ' ' + b + . . . , Python performed each addition operation separately. That is, probably the instructions underneath looked like this:

1. Allocate memory that will fit variable a and string ' '.
2. Copy value a
3. Copy value ' '
4. Add the obtained result to variable b.
5. Allocate memory that will fit the previous result and variable b.
6. . . .

And so on. And all this costs time - new allocations, copying. At least that's how I think it worked, I'm not sure if Python developers haven't made some optimizations for this case and maybe they do it differently? I don't know, I haven't looked that deep, but looking at the results, I don't think so.

8.2.4.5 Summary - About Performance In Python, mechanisms that seem elegant in a given situation are usually optimized and prepared for such, hence it's worth using them. This snake is simply beautiful. Elegant code.

So use f-strings wherever you can and enjoy life, where you have many strings to join in a predictable way, use join. This way your code will be prettier but also faster!

8.2.5 Examples of Basic String Operations

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__getnewargs__', '__getslice__', '__gt__',
 '__hash__', '__init__', '__le__', '__len__', '__lt__',
 '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__', '_formatter_field_name_split',
 '_formatter_parser', 'capitalize', 'center', 'count', 'decode',
 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index',
 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle',
 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition',
 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>> text = "some text"
>>> text.capitalize()
'Some text'
>>> text.count("t")
2
>>> text.replace("t", "f")
'some fexf'
>>> coolest_org = "NAFO"
>>> coolest_org.lower()
'nafo'
>>> coolest_org_lower = coolest_org.lower()
>>> coolest_org + coolest_org_lower
'NAFOnafo'
>>> "RuSSia" + "HATE" "CLUB"
'RuSSiaHATECLUB'
>>> 'test' * 10
'testtesttesttesttesttesttesttesttesttest'
>>> coolest_org = "nafo"
>>> coolest_org = coolest_org.upper()
>>> coolest_org
>>> 'NAFO'
>>> coolest_org.swapcase()
>>> 'nafo'
>>> coolest_org.isalpha()
>>> True
>>> coolest_org.find("F")
>>> 2
>>> "AaBbCcDd"[::2]
>>> 'ABCD'
```

Note that strings are immutable. This means that we cannot modify them and any modification results in a new string being created. We'll talk more about mutability later when we discuss lists and tuples.

8.3 Bytes

8.3.1 Brief Characteristics

Hmm, bytes. Yes, this is a basic and simple type in Python, it's simply a sequence of bytes as the name suggests. Useful for file operations. Read about it yourself.

8.4 Boolean Type

8.4.1 Brief Characteristics

Here things look simple - the boolean type is the so-called bool - either true or false. A type of length 1 bit. Either 0 or 1, either `True` or `False`.

8.4.2 True Values vs False Values

Usually used in conditions, setting flags, and so on. It's worth noting that in Python the boolean type is somewhat extended. That is, anything that can be evaluated to certain things will be treated as a boolean type. Simply put, False is zero or something empty. True is any number other than zero or something non-empty. An empty string is False, some text is True. An empty list is False. A non-empty one is the opposite.

```
>>> bool(1)
True
>>> bool(0)
False
>>> bool(0.1)
True
```

The code above when compiled and run will give us:

```
numbers in general: 0x7ffc9f728f20 -- 0x7ffc9f728f20
number no. 0: 0x7ffc9f728f20 -- 0x7ffc9f728f20 -- value: 1
number no. 1: 0x7ffc9f728f24 -- 0x7ffc9f728f24 -- value: 2
number no. 2: 0x7ffc9f728f28 -- 0x7ffc9f728f28 -- value: 3
number no. 3: 0x7ffc9f728f2c -- 0x7ffc9f728f2c -- value: 4
int size: 4
```

Let's analyze what's going on here.

Before we do that, I'll just note that if you ran this code on your machine, you might have gotten slightly different results. That's normal.

For most people unfamiliar with C/C++, this code might seem a bit cryptic, but it's actually quite simple.

Let's start with the line:

```
printf("numbers in general: %p -- %p\n", &numbers, numbers+0);
```

I assume the first part of the print is understandable to everyone, except maybe `%p` - this simply tells us that the argument to print will be a specific data type.

And what's this whole `&numbers` - the & operator says that I want to get the address of a given variable - that is, its location in memory. Because as we well know, variables are allocated in memory, in a certain place chosen by the computer. Again - we talked about this in chapters 4 and 5.

This place is usually described as an 'address' - that is, the number of bytes/bits from the 'beginning' of memory that the processor must 'jump' to reach a given variable.

Our array (which is kind of like a Python list, but not quite) is located at address: 0x7ffc9f728f20 (hexadecimal notation), and this is also the address of our first element.

However, the compiler needs to know at what address the next element of our array is located. How? I've already explained this and I'm counting on your answer. You'll find mine below.

We declared that the elements of our array will be of type `int`. The `int` type on the computer I'm using is 4 bytes, or 32 bits. This is basically standard (although officially the standard says that int should just be at least 16 bits, it doesn't specify its size exactly), but sometimes there are exceptions to the rule, depending on the architecture, hence the `sizeof(int)` in the code - it returns the size of a given type in the current environment.

Therefore, if 0x7ffc9f728f20 is the address of the first element, which occupies 4 bytes in memory at addresses:

- `0x7ffc9f728f20`,
- `0x7ffc9f728f21`,
- `0x7ffc9f728f22`,
- `0x7ffc9f728f23`,

then we can deduce that the next element of this array will be after it, at address `0x7ffc9f728f24`, that is, 4 bytes further. The next one again another 4 bytes and so on, until the last element. A simple formula can be extracted from this.

The address of a specific array element can be determined by the formula:
$first\_element\_address + (index * type\_size)$.

The computer uses this formula - every time you write `numbers[index]` the compiler internally translates it to

$(numbers + (index * type\_size))$.

What does `*` mean for the compiler? Nothing else than 'go to the given address and take the value at that address.'

So when we write numbers[0], our compiler will translate it to

$(0x7ffc9f728f20 + 0)$=$(0x7ffc9f728f20)$,

which in turn means: take the value from this address and insert it here.

In the case of numbers[1], for example, it will be

$0x7ffc9f728f20 + (1 * sizeof(int)) = 0x7ffc9f728f20 + (1 * 4)$

$0x7ffc9f728f20 + (1 * 4) = 0x7ffc9f728f20 + 4$

that is

`0x7ffc9f728f24`.

Clear? For me it is. If you have trouble understanding this concept, don't worry, many people don't fully understand pointers, addresses, and memory. I had trouble with it too. At least at the beginning.

You can help yourself with Gynvael's videos - Gynvael's Code: Pointers #1 or lectures from CS50 - Harvard course they, as people with much more knowledge, explain the whole issue much better than I do.

8.4.3 What Would It Look Like If We Indexed from 1?

Let's assume we index from 1. Then the formula would have to be modified - and it would look like this:

$first\_element\_address + ((index - 1) * type\_size)$

Another solution would be to shift the location of the first array element 4 bytes forward relative to the array's address itself, but then our array would occupy additional space in memory unnecessarily, as those first x bytes, where x is the size of the given data type, would simply be empty. That's one thing, and two, you would have to remember that the array's address is not the address of its first element.

Both these solutions are nonsensical, because while it's not much - a few bytes on each array, or one subtraction operation, when we multiply it by the number of such variables we have in memory, it adds up to quite a substantial sum of bytes/operations that are, in essence, unnecessary.

Additionally, how much code is already based on zero-based indexing. It would be impossible to change all of this.

Of course, there are also other arguments for indexing or counting elements from zero, such as those advocated by Dijkstra - Why numbering should start at zero]. That's quite a well-known and important gentleman, for those who don't know ;)

Overall, I used some simplifications and mental shortcuts here, but the general concept is conveyed.

8.4.4 Examples of Basic Operations on Lists

8.5 Tuples

8.5.1 Brief Characteristics

A tuple is really nothing more than an immutable list. What does this mean? More or less that after declaring a tuple, it can't be modified anymore. We declare once and that's it. This carries many consequences which I'll tell you about soon.

The only thing you can do with a tuple is read it, copy it, or redeclare a variable with a given name. Examples below.

Why do we need these tuples? Generally, data immutability often makes it harder to shoot yourself in the foot with certain things. Plus it's an approach more similar to functional programming, let's say. Immutability is quite cool.

Besides this, we have one more advantage here. Performance.

8.5.2 Efficient Beast

Well, if it's an immutable data structure then the Python interpreter knows exactly how much memory to allocate plus for certain reasons this process happens somewhat faster. So here allocation beyond needs doesn't take place plus the instruction executes somewhat faster, Python knows what types are used, knows the specific data we used etc.

As an anecdote, I'll mention a story where using tuples reduced our memory usage from 4 GB to ~2.1 GB in a certain small web app. In other cases, the reduction was even more drastic.

8.5.3 Examples of Basic Operations on Tuples

```
dir(tuple)
['__add__', '__class__', '__contains__', '__delattr__',
 '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__getitem__', '__getnewargs__',
 '__getslice__', '__gt__', '__hash__', '__init__', '__iter__',
 '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__rmul__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__',
 'count', 'index']
In [9]: some_tuple = ("f", 1, 2)
In [10]: some_tuple[1]
Out[10]: 1
In [11]: some_tuple[1] = 2
In [13]: some_tuple.count(1)
Out[13]: 1
In [14]: some_tuple.count(2)
Out[14]: 1
In [15]: some_tuple.count(3)
Out[15]: 0
```

8.6 Dictionaries

8.6.1 Brief Characteristics

What is this magically sounding dictionary also known as Dict/HashMap? Well, it's nothing other than a kind of dictionary/mapping. Just as in a regular dictionary we have some kind of mapping of **keys** to **values** in the form of a word and its meaning, similarly it is in Python's dictionary/hashmap.

To the point.

A dictionary in Python is a data structure that allows us to store any values under specified keys. Imagine a list, but instead of a numerical index, you have an index in the form of a specified key.

In practice, it looks like this:

```
>>> test_dict = {"test": "some_string", 1: "hehe", 2: 3}
>>> test_dict["test"]
'some_string'
>>> test_dict[1]
'hehe'
>>> test_dict[2]
3
```

Predictable. The rest works similarly to a list.

What's different from how a list works is that while in a list there is a guarantee that elements will always be in the order we put them into the list. A hashmap by definition doesn't provide such a thing. The current implementation of CPython, from version 3.8 or so, nevertheless provides something like this additionally, meaning that a regular `Dict` became an `OrderedDict`, however it's better not to count on this, as Python versions like 3.6 or 3.7 are quite new and there are lots of projects written in them. What does this mean? Well, that the code you will write will probably be run on a Python version that doesn't take into account and doesn't guarantee preservation of element insertion order, so it's better not to rely on this too much, because in most cases this order will be preserved anyway, but it's not guaranteed by implementation, meaning there will always be that 1% where something goes wrong. Then try getting such a bug for investigation.

Of course, if you are aware and know what you're doing, plus you have guarantees about which Python versions your code will run on, then go ahead. However, remember, in the newest version of Python -> okay, below 3.8 or 3.7 not necessarily. Check exactly in which version `OrderedDict` was introduced as default.

8.6.2 How Does the Process of Adding Elements to a Dict Work?

Well, generally just as in the case of a list we had a numerical index, using which Python calculated the offset in memory, in the case of a dict we have something called a hash function. This function takes as an argument the key we're using and based on it tries to generate a fairly unique hash. Then based on the hash, usually through modulo operation, we figure out the address/offset where the given value is held.

Logical? So yes, every time someone writes `some_dict["key"]`, underneath something happens where the Python interpreter, to get the address from which to read the value for a given key, takes this key, throws it into a hash function, I don't know, let's say `hash("key")`, this function then returns us some possibly unique hash generated/calculated based on the given key. From the hash we conjure up an address/offset. Something like that.

Why fairly unique?

8.6.3 Hash Collision

Hash collision is something that happens sometimes. Why? Well, the hash function can't be completely random. It must be stable and repeatable. This means that for a given argument it must always return the same thing, hash generation must happen in a predictable way. Why? Well, if it were otherwise, and for one key it would be possible to generate several hashes, then a problem would arise in the form that we could never, or sometimes we couldn't, hit the exact address where we originally assigned the value. What does this mean?

The lack of complete randomness means that hashing algorithms are limited to some extent. They are also limited by efficiency and the time that a computer can devote to hashing, which happens quite often however, without costs to the user. Therefore, a compromise had to be found between the complexity of the hash function and its resource consumption, execution time, and the uniqueness of provided hashes for different keys.

Currently, smart heads have figured out some golden mean, however in today's times it happens to operate on such large data sets that hash collision happens and the hash function generates the same hash for two different keys, causing one key to overwrite the other. A very, very rare case. However, if you have to process a million trillion records, then suddenly very rare cases have about 100% chance of appearing.

8.6.4 What Can Be a Key?

A key in a dictionary can be any value/variable/object that is hashable. What does this mean? Well, that in its definition it contains an implementation of the `__hash__` method. In a big shortcut. That is, those objects for which a hash function has been implemented can be keys. Quite logical, because if we don't have a method to hash a given object then we won't calculate the hash. We won't calculate the hash from the key then we won't figure out the address/offset. Without this, we don't know where in memory to store the value. And therefore we must have this method implemented. Logical.

8.6.5 Pass by Value & Pass by Reference

What's this about? About passing content by reference or by value. More precisely, it's about the fact that some objects Python will copy, as is the case with list slicing and getting its copy in a way that will share the internal elements of this object for both the copy and the original. I wrote about this a bit already when writing about lists.

I think it was explained fairly clearly there. Now this - why am I mentioning this in the context of dicts? Well, mutable data types are often passed by reference, meaning instead of the object itself, we get a reference to it. For this reason, for example, a list cannot be a key in a dictionary - it is mutable, passed by reference, and for this reason doesn't implement the `__hash__` method, making it unhashable, and therefore the dictionary implementation in Python, when trying to establish a new key that is a list, will throw an error.

Let's recall the code with which I illustrated passing by reference vs by value.

```
>>> pass_by_reference = [[1,2,3], 1, 2, 3]
>>> new_pass_by_reference = pass_by_reference[:]
>>> pass_by_reference
[[1, 2, 3], 1, 2, 3]
>>> new_pass_by_reference
[[1, 2, 3], 1, 2, 3]
>>> pass_by_reference is new_pass_by_reference  # two different lists
False
# first element of list is another list
# lists are mutable and we pass them by reference
>>> pass_by_reference[0]
[1, 2, 3]
# here we modify second element of this internal list from original
>>> pass_by_reference[0][1] = "test"
>>> pass_by_reference[0]
[1, 'test', 3]
# somehow list in copy also changed
>>> new_pass_by_reference
[[1, 'test', 3], 1, 2, 3]
>>> new_pass_by_reference[0]
[1, 'test', 3]
```

It's worth being careful about this, especially when choosing function arguments or establishing default values. Pass by reference -> we return the address and there the modification of the given object occurs. Pass by value -> we return the value itself and its 'copying' occurs fresh instead of modifying the variable at the given address, we get a new one. Another example:

```
student = {"Putler": 10}
def test(student):
    new = {"Volodia": 20, "MonkeyMan": 21}
    student.update(new)
    print("Inside function", student)
test(student)
print("Outside function:", student)
```

As seen above, dict is passed by reference, that is by address so to speak. Python therefore goes to the given address and modifies the object, causing changes to spread to places where an inexperienced programmer might not expect. In the case of passing by value, the matter is different. The original object is not modified, only its copy.

```
>>> student_name = "NAFO"
>>> def test(name_name):
```

```
...     student_name = name_name + " <3"
...     print("Inside: ", student_name)
>>> student_name
'NAFO'
>>> test(student_name)
('Inside: ', 'NAFO <3')
>>> student_name
'NAFO'
```

I don't know if this makes sense, maybe we'll come back to this. Analyze and search a bit on the net yourself about this topic to additionally clarify the situation.

8.6.6 Shallow Copy and Deep Copy and Dictionary Keys

We have all this passing by reference, value etc. Let's talk now about shallow and deep copies. Briefly but briefly, but worth mentioning.

When we used slicing as a method of copying a list, we got a so-called shallow copy of this list. What does shallow mean? Namely, only the initial object, the object from the very top was copied. Everything inside that was passed by reference was not duplicated. Only references were copied. This is a shallow copy.

A deep copy is a copy where the interpreter 'enters' the object we're copying and copies everything by value, not by reference. This means that we get an actual, independent and standalone duplicate of the given object and not just its 'top level' as in the case of a shallow copy.

Sometimes needed. Worth knowing, because in some cases we think we have two different objects after copying using a shallow copy, we modify one object and bam, changes in both. This can cause really ugly bugs to debug. Not recommended.

8.6.7 dict.values() keys() items()

Pay attention to these three methods. Play with them and summarize your conclusions. I'll just point out one thing.

What type of objects do the functions dict.keys(), dict.values(), dict.items() return? dict.items() - obvious. List of tuples. But not quite. Because if you look closer it's a dict_items class, which isn't quite a list – it's kind of an extended class, because it allows us to perform operations on it like on sets. Similarly with keys() and values(). So on objects returned by these functions, you can perform operations of sum, difference, or intersection from sets. Tldr – these functions return an iterable set-like object.

8.6.8 Examples of Basic Operations on Dictionaries

```
In [17]: dir(dict)
Out[17]:
[(...)
 'clear',
 'copy',
 'fromkeys',
 'get',
 'items',
 'keys',
 'pop',
 'popitem',
 'setdefault',
 'update',
 'values']
In [18]: some_dict = {"NAFO": "OK", "SS": "NOT OK"}
""" dict_values is a set-like object on which
you can perform operations like on sets"""
In [19]: some_dict.values()
Out[19]: dict_values(['OK', 'NOT OK'])
# dict_keys similarly
In [20]: some_dict.keys()
Out[20]: dict_keys(['NAFO', 'SS'])

In [21]: some_dict.items()
Out[21]: dict_items([('NAFO', 'OK'), ('SS', 'NOT OK')])
```

8.7 Sets

8.7.1 Brief Characteristics

What are sets? Analogously as in mathematics. It's kind of like a list, but without repetitions. At least apparently. Underneath it's somewhat different, because underneath sets are closer to a hash map. Actually, it is somewhat a hashmap. What for and why? Well, let's ask ourselves, what are the attributes of sets. Each element occurs only once. Not necessarily preserved insertion order. Starting to sound familiar? Yup. Sets are like hashmaps where values are also keys in a way.

What is the advantage of a set? First is element deduplication -> each occurs exactly once. We can extract 'statistics' from a given element, how many times it was added to the set, but in the set itself it will appear only once. The second is performance.

8.7.2 Search Faster than in Warsaw's Wola

Searching in a set has computational complexity at the level of O(1) - constant time. What does this mean? Regardless of the size of the set to check membership of a given element in the set, we perform an operation that is characterized by constant execution time independent of size. So even for very very large sets, if they fit in memory, we can determine super quickly whether they are in a given set.

In the case of a list, it's not so easy, especially if the data is unsorted.

Why is this? Well, because underneath it's somewhat a hashmap, to check if an element belongs to the set it's enough to just calculate the hash of this element and then check if everything matches. Hence O(1) regardless of set size.

This in turn forces restrictions on what we can throw into the set. What kind? The same as with keys in dictionaries.

Additionally, the Python set also handles similar operations as a mathematical set. Conjunction, alternative, difference. A regular list can't handle all of this.

8.7.3 Examples of Basic Operations on Sets

```
In [22]: some_set = {1,2,3,4}
In [23]: another_set = {3, 4, 5}
In [24]: some_set
Out[24]: {1, 2, 3, 4}
In [25]: some_set | another_set
Out[25]: {1, 2, 3, 4, 5}
In [26]: some_set & another_set
Out[26]: {3, 4}
In [27]: some_set - another_set
Out[27]: {1, 2}
In [30]: dir(set)
Out[30]:
[(...),
 'add', 'clear', 'copy',
 'difference', 'difference_update', 'discard', 'intersection',
 'intersection_update', 'isdisjoint', 'issubset', 'issuperset',
 'pop', 'remove', 'symmetric_difference',
 'symmetric_difference_update', 'union', 'update']
In [31]: some_set.add(7)
In [32]: some_set
Out[32]: {1, 2, 3, 4, 7}
In [33]: some_set.add(7)
In [34]: some_set
Out[34]: {1, 2, 3, 4, 7}
```

8.8 Summary

Phew, finally. Quite a chunk of text, right? And these are just selected Python types.

It's worth knowing them well, playing with them and getting familiar. Why? Well, in Python's standard library there are so many goodies, so many different things that make life easier, it's mind-boggling. It's a shame to reinvent the wheel and implement something yourself when the language provides its version.

Moreover, implementing from scratch is often also pointless for one very important reason. Well, your own implementation might be flawed because you check it, maybe people on Code Review, and that's it. However, when it comes to code that's in Python's standard library and contributors' implementations, the matter is that this code has been tested and reviewed by thousands of people. Mistakes happen, that's true. However, where are we more likely to find a bug? In code that has been reviewed by thousands of people, which is tested in millions of production applications and covered by many tests? Or in code that you and maybe your team reviewed? There's no comparison.

Additionally, I'll bet my hand, believing that thanks to the effort of thousands of contributors, code from Python's standard library will be better optimized. Use what has been built and don't reinvent the wheel by creating your own naive implementations of sorting algorithms or something. Sometimes there is such a need, true, but I doubt that you would have such as a junior wannabe.

That's why good knowledge of Python's standard library is necessary. Leave your own implementations for learning or fun purposes, to understand how something works. In production code, let's try to avoid this in favor of proven solutions from the standard library.

This not only makes the task easier but makes the code more solid, more optimized, and probably delivered faster. It's easier to put something together from ready-made blocks than to build a house yourself starting from extracting clay and firing bricks.

8.9 Questions and Exercises

1. Is there dynamic typing in Python? Or maybe static?
2. What does this mean? What are its disadvantages, what are the advantages.
3. List the basic data types in Python.
4. And the more complex ones?
5. What are the differences between basic numeric types?
6. What does it mean that a value is `truthy` or `falsy` in Python? Give examples distinguishing what is what.
7. What are the ways of declaring text in Python?
8. What is an escape character and why do we use it?
9. What functions facilitate converting characters to numbers, to other encodings, which I mentioned?
10. Prepare an article in which you describe what each function is about, briefly characterize each basic type. Give examples for which functions don't work and guesses why. The target audience is other beginners. Let it be an extensive article full of your personal notes and not just Copy Paste from documentation. Give live examples.
11. Prepare a summary list for the basic data types you know in which you will present and briefly describe what methods/attributes particular types have and what each method does. Skip those that start with `__`. What command do you need to use here?
12. The same as in 11., but for complex types. Describe very thoroughly. You can combine the articles. Focus on showing practical use. Comment on the code and show examples other than here.
13. Send articles from 12. and 11. to olafgorski@pm.me and I'll check them and give feedback :)
14. Prepare a comparison table: list vs tuple vs set.
15. Experiment with really large numbers. Describe your conclusions.
16. Create code that will get a number from the user and then say whether it's even or not. How to get something from the user? Google if you don't remember.

17. Create a string and then swap its last letter with the first.

Remember that you can put your answers on GH here - https://github.com/grski/junior-python-exercises, and I'll check your solutions and give feedback. More about this in the 'Interactive Part' subchapter.

9 Loops and Iteration

Let's talk about loops in Python and say a few words about iteration. What it is, what it's for, and who needs it. We'll discuss different types of loops: `for` loops (step-by-step loops), `while` loops (regular loops), and cover iteration and iterable objects.

But what does all this mean?

Loops are a programming concept used to execute some action, some piece of code, a specified number of times. Imagine you need to process 10 elements from an array. You need to perform some operations on each of them. Let's say these operations aren't just one line but complex processing. Actually, let's just take multiplication as an example. So what now? A naive solution would look like this:

```
>>> elements = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> elements[0] *= 2
>>> elements[1] *= 2
>>> elements[2] *= 3
(...)
>>> elements[9] *= 3
```

Nightmare. Now think about having more elements. Say a million, because that's how many users you have. You can't edit this manually, unless we hire a finite number of students or outsource it. Still, it would take an incredibly long time. And this is exactly where loops come in to save the day.

9.1 For Loop

9.1.1 Brief Characteristics

Let's start with what's called a for loop or step-by-step loop. This is a loop that allows us to 'walk through' the elements of a given object and enables us, step by step, or element by element, to process each element. This means that the `for` keyword allows us to iterate over elements of an iterable object.

9.1.2 Iterable Object

What is an iterable object? It's an object that, when passed to the `iter()` function, returns us an iterator. And an iterator is something we call `next()` on. You've already seen several examples of iterable objects: Lists, Dictionaries, Tuples.

These are objects that have implemented the `__iter__` method and subsequently `__next__`, which in short means the programmer told Python how to take subsequent elements from a given object/structure and which returns an iterator object.

In lists, dicts, tuples, we have this by default as part of the language. If we create our own specialized classes, we can also make them iterable by implementing these methods. So in short, it's something you can iterate over.

To iterate means to go step by step through the elements of a given structure. Clear? Hopefully. If not, Google it or look at the code below.

How does this look in practice? Let's modify our example from above.

```
>>> elements = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> for index, element in enumerate(elements):
...     elements[index] *= 2
...
>>> elements
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Instead of a million lines, we have two. Nice, right? But wait, what's this magical `enumerate` function? It's a built-in Python function that allows us to number the elements of a given object. In simpler terms, it's a wrapper, an overlay on our given object, like a list, which besides the given element from the object also returns its index.

The whole thing, translated into plain English instructions, looks like this:

1. Take the elements object. Elements is a list.
2. Pass the elements object as an argument to the enumerate function.
3. The enumerate function returns a new object. This object is something that returns us subsequent elements of the original object and attaches an index/number to them.

Then when we have this newly returned object, the for loop comes into play and does its thing.

1. Take the newly returned object
2. Under the hood call `__iter__()`, which will return us an iterator for this list, unless we already have an iterator passed by default
3. Call `next()` with the received iterator as an argument
4. Next will return the next element that the iterator has to pass
5. When there are no more elements, the iterator will raise a StopIteration exception, which is correct behavior for iterators that have no more elements to pass.

```
>>> l = elements.__iter__()
>>> next(l)
2
>>> next(l)
4
>>> next(l)
6
>>> next(l)
8
>>> next(l)
10
```

```
>>> next(l)
12
>>> next(l)
14
>>> next(l)
16
>>> next(l)
18
>>> next(l)
20
>>> next(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

This is how it looks in practice and this is more or less what happens under the hood when we use a for loop and iterate over a given object. In short, as I wrote, we simply step through all elements, receiving a single element at our disposal and being able to process it in some way.

This has many applications, and the for loop is often daily bread in Python. You'll see it many more times. Get familiar with it quite well, because it will probably be your friend.

9.2 While Loop

9.2.1 Brief Characteristics

The while loop, as I call it, is simply the `while` loop. While the for loop executes by going through elements of a given iterable object, the while loop executes as long as a given condition is true. The concept is somewhat similar to the for loop, but slightly different.

9.2.2 Examples of Loop Usage

```
counter = 0
while True:
    counter += 1
    if counter >= 10:
        break
```

And that's probably it. The rest is for independent analysis.

9.3 Comprehensions

Comprehensions are nothing more than a kind of syntactic sugar that makes certain typical behaviors describable more concisely in Python. These behaviors relate to for loops and using them to create new objects, lists, tuples, etc.

Example of how to use comprehensions:

```
list_comprehension = [x for x in range(10)]
dict_comprehension = {x: x**2 for x in range(10)}
set_comprehension = {x for x in range(10)}
set_comprehension_variation = set(x for x in range(10))
tuple_comprehension = tuple(x for x in range(10) if x % 2)
```

Read a bit more and experiment on your own. Remember that comprehensions can be nested, meaning you can have a list comprehension made up of a list comprehension. Book. Inception.

I'll just additionally mention that personally, I like to write longer list comprehensions in the following way:

```
list_comprehension = [
    value
    for value in range(10)
    if value % 2
]
```

That is, in three lines, with each line having successive elements. If the comprehension is short, I don't do this, but for more complex ones I do. In my opinion, this improves readability.

9.4 Generators

Since we're talking about loops and iteration, I'll mention generators. What are they?

Generally, these are functions that `yield` something instead of returning. What does this mean in practice and what's it all about, what's it for?

Speaking in simple terms, it's simply about the fact that sometimes we have data sets that are too large to load into RAM at once, as RAM is finite, large but finite and limited. What to do then? A generator is one of the strategies for dealing with such a situation. Generators allow us to process large data sets step by step.

Generators are actually just regular loops under the hood, wrapped in functions that `yield` some value. What does this mean? Instead of returning a given element and ending execution, a generator yields a given value, 'saves' or 'remembers' its current state and waits for a signal to return the next element. In the meantime, we can process and do something with the given values.

Generators can work on finite sets, they can be infinite. Many variations.

```
def infinite_sequence():
    num = 0
    while True:
        yield num
        num += 1
for i in infinite_sequence():
    print(i, end=" ")
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 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41 42
```

```
[...]
6157818 6157819 6157820 6157821 6157822 6157823 6157824 6157825 6157826 6157827
6157828 6157829 6157830 6157831 6157832 6157833 6157834 6157835 6157836 6157837
6157838 6157839 6157840 6157841 6157842
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

Generators also have the characteristic that they are one-time use. This means that once initialized, a generator can only be traversed once. This plus the fact that we don't load data into memory all at once is the main difference between generators and traditional lists or tuples.

9.5 Walrus Operator

Since Python 3.8, we have something called the Walrus operator or assignment expression. It allows us to assign variables not only in statements but also in expressions, using the operator `NAME := expr`.

Alright, but what does this mean in practice. Let's look at the code.

```
data = None
if our_function_getting_json(some_arg) is not None:
    data = our_function_getting_json(some_arg)
    data.do_stuff()
```

Rather simple to understand, reasonable, right? Example taken from some code. An ugly example. The above code sucks in this context. How could we improve it?

```
data = our_function_getting_json(some_arg)
if data is not None:
    data.do_stuff()
```

We can do something like this, but is it the shortest possible, the best possible? Currently probably yes, but...

It would be nice if we could declare that variable right there in the if statement - simply store the function's result where it's originally used. This is important when we want to perform some operations on the result of an expression that we executed, for example in a condition, but because we currently can't assign variables in expressions, we have to store it ourselves, earlier. Whether in loops, list comprehensions, lambda functions, or others.

Unless we use the walrus operator.

```
if (data := our_function_gettin_json(some_arg)) is not None:
    data.do_stuff()
```

Other examples of usage:

```
if (match := pattern.search(data)) is not None:
    match.do_stuff()
```

```
while (value := read_next_item()) is not None:
    ...

filtered_data = [y for x in data if (y := f(x)) is not None]
results = [(x, y, x/y) for x in input_data if (y := f(x)) > 0]
# or something like below
stuff = [[y := f(x), x/y] for x in range(5)]
y = y1 := f(x) # ERROR
bar(x = y := f(x)) # ERROR
bar(x = (y := f(x))) # OK
something := 'lalala' # ERROR
something2 = 'hey' # OK
```

9.6 Summary

Loops, generators, list comprehensions etc. are useful tools for every programmer and fundamental building blocks of any code. It's good to be very well acquainted with them.

9.7 Questions and Exercises

1. Write examples of for and while loops. 3 different ones each.
2. Write an article comparing the for loop with the while loop.
3. Add a comparative characterization of for loops and comprehensions.
4. Write code that finds the largest and smallest number in a list.
5. Write code that counts words in a given string.
6. Then count letters.
7. And their frequency of occurrence.
8. Check if in a given list, there are two numbers – a and b, whose sum equals a given number.
9. Write code that from any list (numbers only) will display only those that are less than 5.
10. Ask the user for a number and then print all divisors of that number.
11. Get two numbers from the user and then return the square of their sum.
12. Write code that will read a string from the user and then display the first three letters of that string, consecutively, without new lines. E.g., “MelonTusk” -> “MelMelMel”
13. Get a string from the user and then display it with letters in reverse order
14. Based on the f-string benchmarking code from the previous chapter, conduct an analysis of what's faster: list(), [] or list comprehensions. Each should contain natural numbers from 1 to 10.

Remember that you can post your answers on GH here - https://github.com/grski/junior-python-exercises, and I'll check your solutions and give feedback. More about this in the ‘Interactive Part’ subsection.

10 Functions

Functions are a fundamental element of programming that allows grouping code for reusable purposes. They are also used to divide a program into smaller, more understandable fragments, which makes it easier to create and maintain, improving readability. As I've mentioned before, this is critically important, right? After all, you write code once but read it dozens of times, sometimes. Both you and other people.

What are functions and why are they important in programming? We'll discuss this in more detail shortly. Let's start with regular functions.

10.1 Regular Functions/Methods

First, let's discuss traditional functions, or when we talk about functions defined in classes, methods.

10.1.0.1 Syntax for defining functions in Python. In Python, a function definition starts with the `def` keyword, followed by the function name and parentheses with arguments. The code inside the function is indented. Example of a function definition:

```python
def function_name(arg1, arg2):
    # function code
    return result
```

10.1.0.2 Passing arguments to functions To call a function, you need to provide its name and appropriate arguments. Depending on what arguments are required by the function, you need to provide the appropriate number of arguments.

10.1.0.3 Returning values from functions Functions in Python can return values using the `return` keyword. When a function is called, the code inside the function is executed, and then the value is returned and assigned to a variable or used in another way. If a function doesn't return any value, it returns `None` by default.

10.1.0.4 Using the `return` keyword to return values from functions To return a value from a function, you need to use the `return` keyword along with the expression to be returned. For example:

```python
def increment(x):
    return x + 1

a = 5
b = increment(a)
print(b)  # Output: 6
```

10.1.0.5 More values to return In Python, it's possible to return multiple values from a function using a tuple or dictionary. To return multiple values as a tuple, simply separate them with commas. Example:

```
def min_max(x):
    return min(x), max(x), x

a = [1, 2, 3]
min_a, max_a = min_max(a)
# alternatively:
result = min_max(a)
min_a = result[0]
max_a = result[1]
print(min_a)  # Output: 1
print(max_a)  # Output: 3
```

By the way, the line with the function call is an example of so-called tuple unpacking. A useful and important thing.

If we determine that the second argument returned by the function is unnecessary, or that in a function that returns three arguments, we only want the first one, or only the second one, or maybe the last one, that's also possible. How? Analyze the code below.

```
def min_max(x):
    return min(x), max(x), x

a = [1, 2, 3]
min_a, max_a, _ = min_max(a)
# the third argument will be ignored and not assigned to
# any variable _ is a placeholder that Python recognizes
*_, last_argument = min_max(a)
first, *all_the_rest = min_max()
first, _, the_third = min_max(a)
first, *_, the_third = min_max(a)
```

Play with the code above and see what conclusions you draw. Additionally, I recommend trying to create functions with more return arguments.

To return multiple values as a dictionary, you need to create a dictionary and return it using the `return` keyword. Nothing groundbreaking.

```
def min_max(x):
    return {'min': min(x), 'max': max(x)}

a = [1, 2, 3]
min_max_dict = min_max(a)
print(min_max_dict['min'])  # Output: 1
print(min_max_dict['max'])  # Output: 3
```

10.1.0.6 Default argument values In Python, you can define default values for function arguments, which means these arguments don't have to be passed to the function when it's called. The default value is used when the argument isn't passed to the function.

However, there's one important thing to remember here when defining default values, which we'll discuss shortly.

Here's an example of a function with default arguments:

```
def greet(name, greeting='Hello'):
  print(f'{greeting}, {name}!')


greet('Alice')  # will print "Hello, Alice!"
greet('Bob', 'Hi')  # will print "Hi, Bob!"
```

In the above example, the `name` argument doesn't have a default value, so it must be passed to the function when it's called. However, the `greeting` argument has a default value of `'Hello'`, so it can be omitted when calling the function. If you do pass a second argument to the function, its value will be used instead of the default value.

The second argument can be passed in two ways.

```
greet('Bob', 'Hi')  # will print "Hi, Bob!"
greet('Bob', greeting='Hi')  # will print "Hi, Bob!"
```

This second way is using keyword arguments - arguments with default values that are optional. Arguments without default values can also be passed as keyword arguments.

```
greet('Bob', 'Hi')  # will print "Hi, Bob!"
greet(name='Bob', greeting='Hi')  # will print "Hi, Bob!"
```

The argument name is used as a key. You can use keyword arguments to pass arguments to a function in any order, regardless of the order in which they were defined in the function definition, but it's better to stick to the conventional order. Just because you can doesn't mean you should.

Here's an example of using keyword arguments:

```
def greet(name, greeting='Hello'):
  print(f'{greeting}, {name}!')


greet(name='Alice')  # will print "Hello, Alice!"
greet(greeting='Hi', name='Bob')  # will print "Hi, Bob!"
```

In the above example, the `name` argument is passed to the function using the keyword argument `name`, and the `greeting` argument is passed using the keyword argument `greeting`. You can swap the order of these arguments when calling the function, and the function will still work correctly.

Note: keyword arguments must be placed after positional arguments (those not specified by name) when calling the function and in its definition.

```
def greet(name, greeting='Hello'):
  print(f'{greeting}, {name}!')


greet('Alice', greeting='Hi')  # correct
greet(greeting='Hi', 'Alice')  # SyntaxError
def greet(greeting='Hello', name):
  print(f'{greeting}, {name}!')  # error
```

10.1.1 One-time herring

Default argument values are like a one-time herring - the Python interpreter will only look at them once, take a sip, and won't deal with them anymore. What does this mean? Well, the interpreter works in such a way that if it can, it initializes default arguments only once. While this isn't a problem with strings and numbers, it becomes an issue with mutable objects. Can you guess what the problem is?

It turns out that each subsequent function call, contrary to what one might expect, won't cause the creation of a new instance of the default element but will simply use a reference to the object initialized at the beginning, e.g., a list. Consider the example below.

```
>>> def xd(default_list=[]):
        default_list.append(1)
        return default_list
>>> xd()
[1]
>>> xd
<function xd at 0x7fd52a695510>
>>> xd()
[1, 1]
>>> xd([1,2,3])
[1, 2, 3, 1]
```

Look at this code and think about it. This often appears in recruitment interviews as a trick :-).

10.1.1.1 Functions as objects In Python, it's possible to assign functions to variables and pass them as arguments to other functions.

Since functions are objects, they can be assigned to variables, just like a variable's value can be assigned. You can also pass a function as an argument to another function. Example:

```
def increment(x):
    return x + 1

def apply_to_list(function, l):
    return [function(x) for x in l]

a = [1, 2, 3]
b = apply_to_list(increment, a)
print(b)  # Output: [2, 3, 4]
```

10.1.1.2 Pure functional Eur—- network Pure functions are functions that always return the same value for given inputs and don't affect the program's state outside their scope. In other words, pure functions don't modify global state, don't cause any side effects (such as changing a global variable or displaying something on the screen), and always return the same value for a given set of inputs. It's a bit like mapping functions we know from mathematics. Mapping values for each argument.

Here's an example of a pure function:

```
def add(x, y):
  return x + y

print(add(1, 2))  # will print 3
print(add(1, 2))  # will print 3
print(add(1, 2))  # will print 3
```

The above `add` function is pure because it always returns the same value for a given set of inputs (here 1 and 2) and has no side effects.

Pure functions are often used in functional programming because they are easy to test and easy to understand. Additionally, pure functions are often more reliable than functions that cause side effects because there's no risk that their operation will depend on global state or other undefined variables.

I personally quite like them. Certain aspects of functional programming are quite close to me, and even when used in OOP, they improve code quality. When I try to write relatively pure functions, I often notice that the code turns out better. Of course, everything with a head, I don't recommend being a purist and fanatic. Know when to make an exception.

10.1.1.3 *args, **kwargs The keywords `*args` and `**kwargs` are used to pass any number of arguments to a function.

`*args` is used to pass any number of non-keyword arguments (so-called "positional arguments") to a function. These arguments are passed as a tuple.

Example:

```
def func(*args):
    print(args)

func(1, 2, 3)
# Output: (1, 2, 3)
```

`**kwargs` is used to pass any number of keyword arguments to a function. These arguments are passed to the function as a dictionary.

Example:

```
def func(**kwargs):
    print(kwargs)

func(a=1, b=2, c=3)
# Output: {'a': 1, 'b': 2, 'c': 3}
```

The keywords `*args` and `**kwargs` are particularly useful when we want to pass any number of arguments to a function without needing to know their exact number or names. You can also use both keywords simultaneously if you want to pass both positional and keyword arguments. You can use them together - `*args, **kwargs`.

10.1.1.4 Tips

To write readable and efficient code using functions, remember a few important tips:

- Divide code into smaller fragments using functions. This makes it easier to read and understand.
- Name functions to describe what they do. Names should be understandable to code readers.
- Try to avoid code repetition and the copy-paste method. Create atomic, small, and reusable functions.
- Specify exactly what arguments are required by the function and what values it returns. Type Hinting and Docstrings are your friends.
- Ideally, a function should have as few side effects as possible - it shouldn't depend on things outside of it.
- Try to limit the number of arguments passed to a function to a minimum. The fewer arguments, the easier it will be to understand.
- A function with one argument is ideal. Two is almost as good. Above that, think about it. Maybe it's worth wrapping those arguments in some class/object?
- Remember to test functions. Test each function to make sure it works correctly. Small, short, pure functions are easy to test.

10.2 Anonymous Functions/Lambda

This will be really brief. Anonymous functions/lambda are simply one-line functions that we don't give a name to, as we use them only in a specific local place or to pass a function as an argument to another function.

It's not good practice to overuse them, but they have their applications in certain situations. I usually prefer regular functions as they allow me to more extensively and descriptively mark what a given code does, give it a name. There are, of course, trivial examples and places where lambdas make sense. However, it's worth remembering not to create monsters that have a million nested lambdas or complex logic.

Below are examples.

```python
# Regular function
def add(x, y):
    return x + y

# Lambda function
add = lambda x, y: x + y
```

10.2.1 Where do we most often use lambdas?

Lambda functions are often used in combination with functions such as `map()`, `filter()`, and `reduce()`, which take functions as arguments, `sorted()`, etc.

10.2.2 Examples

Here's an example of using a lambda function with `map()` and a few other examples:

```
numbers = [1, 2, 3, 4]
doubled = map(lambda x: x * 2, numbers)
# doubled is now [2, 4, 6, 8]

# lambda function returning the square of a given number
square = lambda x: x**2
print(square(5))  # will print 25

# lambda function returning the larger of two numbers
max_number = lambda x, y: x if x > y else y
print(max_number(4, 5))  # will print 5

# lambda function taking a list and returning the sum of its elements
sum_list = lambda lst: sum(lst)
print(sum_list([1, 2, 3]))  # will print 6

# lambda function taking a list and returning a list composed of even elements
even_elements = lambda lst: [x for x in lst if x % 2 == 0]
print(even_elements([1, 2, 3, 4, 5]))  # will print [2, 4]
```

In the above code, lambda functions were used to create short functions returning the square of a given number, the larger of two numbers, the sum of list elements, and a list composed of even elements.

Remember that lambda functions are limited to one line and can't be used for more complex tasks. In such cases, it's better to use a regular function.

Example from slightly more real-life code:

```
def sort_timestamp(orders):
    return sorted(orders, key=lambda x: x.timestamp)
# another example
class Foo:
    def _create_transactions(
        self,
        book_side: dict[(int, list)],
        new_order: Order,
        start: int = None
    ) -> Order:
        """Here we take care of creating transaction and
        fulfilling orders if a match is found in the orderbook.
        When we process the book it's important to reverse the
        ordering based if it's a bid/ask side of orderbook.
        Orders that are fulfilled are removed from the book.
        If no orders are present for given price
        internally it's removed from the order book. Orders that
        are filled partly are added to the orderbook with
        the remaining quantity."""
        sorted_prices = sorted(book_side.keys(), reverse=new_order.is_ask)
        sorted_prices = sorted_prices if start is None else sorted_prices[start:]
        for price in sorted_prices:
            if new_order.is_fulfilled or new_order.price_doesnt_match(book_side_price
                break

            orders_at_price: list[Order] = book_side[price]
            sorted_orders_at_price = sorted(
                orders_at_price,
                key=lambda order: order.timestamp
            )
            (...)
```

The above code sorts objects by one of their attributes.
For reading:

1. https://analityk.edu.pl/python-lambda-wszystko-co-trzeba-wiedziec/
2. https://realpython.com/python-lambda/
3. https://www.geeksforgeeks.org/python-lambda-anonymous-functions-filter-map-reduce/
4. https://www.geeksforgeeks.org/intersection-two-arrays-python-lambda-expression-filter-function/?ref=lbp

11 Classes and OOP

11.1 Classes

You can think of classes, in simplified terms, as collections of functions. Functions created inside a class are suddenly called `methods`.

Classes can be 'connected', which is called inheritance. When one class inherits from another, it takes on its methods unless we override them. Example of a class:

```python
from collections import defaultdict
from queue import Queue

from orderbook.transaction import Transaction

class BaseOrderBook:
    pass


class OrderBook(BaseOrderBook):
    """
    This very simple order book implementation works
    within certain constrictions provided in the requirements.
    These constrictions in some aspects assume the 'happy path'
    hence the implementation will do the same and
    cover just these scenarios. Stubs may be provided in a place or
    two just for interface's sake or my sanity.
    We only implement two types of orders: Limit Order & Iceberg Order.
    """

    def __init__(self) -> None:
        self.asks = defaultdict(list)
        self.bids = defaultdict(list)
        self.order_ids = set()
        self.transactions: Queue[Transaction] = Queue()
        self.last_accessed_transaction_index = self.transactions.qsize() - 1

    def show_new_transactions(self):
        while not self.transactions.empty():
            if transaction := self.transactions.get():
                print(transaction)

    @property
    def maximum_bid(self) -> int:
        return min(self.bids.keys()) if self.bids else float("-inf")

    @property
    def minimum_ask(self) -> int:
        return min(self.asks.keys()) if self.asks else float("inf")
```

For reading: 1. https://realpython.com/inheritance-composition-python/

11.1.1 super() and MRO

`super()` is nothing more than a way to call a method from the class we inherit from. That's all there is to it. It's like when a mother shouts "call the old man".

If we inherit from multiple classes, which is allowed and quite common in Python, and they implement the same method, which method will be used is decided by MRO. Method Resolution Order.

Method Resolution Order (MRO) is the way Python maps multiple inheritance in classes. MRO determines the order in which Python searches for methods in classes when calling a method on an object and from which class it fires the method.

For example, if we have the following three classes:

```
class A:
    def method(self):
        print("This method is from class A")

class B(A):
    def method(self):
        print("This method is from class B")

class C(A):
    def method(self):
        print("This method is from class C")
```

and then we create an object of class `D`, which inherits from both class `B` and `C`:

```
class D(B, C):
    pass
```

If we now call the `method` method on object `D`, Python will use MRO to determine which version of the method will be called. In this case, since class `D` inherits from class `B` first, Python will call the version of the method from class `B`. If class `B` didn't have this method, Python would search class `C`, and then class `A`.

MRO works according to the C3 algorithm, which ensures consistent and predictable results. You can see the MRO for a given class using the `__mro__` function:

```
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.
```

MRO is important because it allows control over how multiple inheritance is mapped in code. This can be particularly useful in the case of classes that inherit from multiple base classes and want to ensure that methods are called appropriately.

The C3 algorithm works as follows:

1. All classes are placed on a list, in the order they are given as inheritance arguments. For example, in class `D` defined as `class D(B, C)`, class `B` comes before class `C`.
2. For each class on the list, add its base class to the end of the same list.

So first from top to bottom, then left to right.

I'll skip the rest.

The C3 algorithm has been used in Python since version 2.3. It is considered more elegant and simpler than the previous algorithm used in Python (algorithmic depth-first search). The C3 algorithm ensures consistent and predictable results for MRO, which enables better control over multiple inheritance in classes and fewer surprises in production :)

For reading: https://www.educative.io/answers/what-is-mro-in-python

11.2 Classmethods, staticmethods

A concept worth knowing to create nice interfaces and sensible classes is class method and static method.

What does this mean? A class method/classmethod is a method that doesn't need an instance of a given class, only the class itself. This means we won't have access to the initialized object and its attributes that we define in `__init__`, but only to variables at the class level, i.e., in its scope.

A static method is a method that doesn't even need variables from the class and doesn't refer to them, nor does it refer to other methods from the given class.

So in short: if we need the object's state or the object itself, regular methods. If only things from the given class, then a class method, if none of the above then just a static method. Examples below.

```python
from datetime import date


class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    # regular method using instance attributes
    def print_name(self):
        print(self.name)

    # class method creating instances of the given class
    # based on year
    @classmethod
    def from_birth_year(cls, name, year):
        return cls(name, date.today().year - year)

    # static method for checking adulthood
    @staticmethod
    def is_adult(age):
        return age > 18


person1 = Person('hejto', 21)
person2 = Person.from_birth_year('Sasin', 1996)

print(person1.age)
print(person2.age)
print(Person.is_adult(22))
```

For reading: https://www.geeksforgeeks.org/class-method-vs-static-method-python/?ref=lbp

11.3 Context Managers

Context managers are classes that define `__enter__` and `__exit__`. These are the things we use together with the `with` clause. In short, these classes simply define magic methods that are triggered when entering the code block with with and after completing the processing of this block and exiting it. They allow us to, well, set some specific context and then clean up after it.

A good example here are file operations. First we want to open the file, set the cursor appropriately, etc., and only then work on it. When we finish working on the file, we'd like to close it so nothing hangs in memory. Instead of doing this manually each time, we use a context manager that comes in all white.

```
class File(object):

    def __init__(self, file_name, method):
        self.file_obj = open(file_name, method)

    def __enter__(self):
        return self.file_obj

    def __exit__(self, type, value, traceback):
        self.file_obj.close()

with File('demo.txt', 'w') as opened_file:
    opened_file.write('Hola!')


# context manager as a generator
from contextlib import contextmanager

@contextmanager
def open_file(name):
    f = open(name, 'w')
    try:
        yield f
    finally:
        f.close()

with open_file('some_file') as f:
    f.write('hola!')
```

For reading: https://realpython.com/python-with-statement/

11.4 Typehints

Type hinting is a mechanism in Python that allows us to “hint” to the programmer what type of data we expect in a given place in the program. In Python, there is no need to declare variable types, so type hinting is an optional tool that can be used to facilitate coding or document code.

Type hinting can be used in several different places in the code, such as function declarations, variables and methods, and in comments.

Examples of type hinting usage:

```
def greet(name: str) -> str:
    return "Hello, " + name

def sum_numbers(a: int, b: int) -> int:
    return a + b

class Point:
    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y
```

The advantage of using type hinting is that it can help document code and make it easier to read for other programmers. Type hinting can also help detect errors at compile time, as the interpreter can report errors if data of the expected type is not passed to a function or method.

The disadvantage of using type hinting is that it can be cumbersome to implement, especially in large projects where it is necessary to manually add type hinting to many places in the code. Additionally, since type hinting is not mandatory in Python, some programmers may not use it in their projects, which can make collaboration and code reading by others more difficult, but forget them. Type hinting is great, I recommend it.

For reading:

1. https://towardsdatascience.com/12-beginner-concepts-about-type-hints-to-improve-your-python-code-90f1ba0ac49
2. https://docs.python.org/3/library/typing.html

11.5 Docstrings

Docstrings (short for "documentation strings") are strings placed in Python code that serve as documentation for the code. They are placed directly after the declaration of a function, method, class, etc., and are usually placed in triple quotes.

Docstrings are often used to describe what a given function, method, or class does, what arguments it takes, and what values it returns. Docstrings are later used by documentation tools (e.g., Sphinx) to automatically generate code documentation.

Here's an example of using docstrings in Python:

```
def add(x, y):
  """
  Function that adds two numbers.

  Args:
    x (int): first number to add
    y (int): second number to add

  Returns:
    int: sum of two numbers
  """
  return x + y
```

In the above example, the docstring describes what the `add` function does, what arguments it takes, and what values it returns.

Note: remember that docstrings must be placed directly after the function declaration and must be enclosed in triple quotes. Docstrings cannot contain any code instructions or be used as regular comments.

Docstrings and type hints are devilishly useful things that I really highly recommend using. Even when you're working independently on an amateur project. Why? Because you also forget what a given code did. It's much easier to return to your own project even then, when it's well documented with type hints or docstrings and concise comments.

Nonsense like "Good code doesn't need comments" let's leave between fairy tales.

11.6 The is Operator

The `is` operator in Python is used to check if two variables point to the same memory location. Whereas the `==` operator is used to check if two variables contain the same value.

Here's an example of using the `is` and `==` operators:

```
x = [1, 2, 3]
y = x
z = [1, 2, 3]

print(x is y)  # will print True
print(x is z)  # will print False
print(x == y)  # will print True
print(x == z)  # will print True
```

In the above example, variables `x` and `y` point to the same memory location (both variables point to the same list), so the `is` operator returns True. Variable `z` contains the same value as variable `x`, but points to a different memory location, so the `is` operator returns False. Whereas the `==` operator only checks the values of both variables, so it returns True for both `x == y` and `x == z`.

Note: remember that the `is` operator is faster than the `==` operator because it doesn't have to compare variable values, but only checks if they point to the same memory location. Therefore, the `is` operator is often used in places where speed is important, and comparison accuracy is not necessary.

Additionally, `is` is a language element, 'immutable' let's say. Whereas the use of the `==` operator depends on how the magic (dunder) method `__eq__` or `__equals__`? I don't remember, check. What does this mean? Well, we can define ourselves how Python will compare objects with `==`. Read about it.

12 Python Tools Ecosystem

A bit about cool tools that are worth knowing. I assume you'll install each of these tools and play around with them.

12.1 README File - What It Should Contain and How It Should Look

There's one thing that every proper project should have - a README file. A well-written, readable, and comprehensive README file is the basic form of documentation for every project, containing elementary information about the project itself, but in my opinion, not only that. A well-written README should contain a bit more. What exactly? I'll explain.

12.1.1 Technology

Usually, when we work with README files, they have a `.md` extension, meaning we write them in Markdown according to convention. Markdown is something that allows us to format text. Using it in READMEs is not a requirement, but it's a widely accepted standard that makes our lives a bit easier and gives us more possibilities than a plain text file.

12.1.2 What Sections Should a Good README Contain?

Below we'll discuss what sections, describing what, a good README file should contain.

12.1.3 Title

We start the README with a title, of course. To do this, simply type the title in a line and add `#` before it, which will mark that line as a Markdown header.

12.1.4 Project Description

Here we place elementary information about the project.

1. What kind of system component is it? E.g., API, Worker, frontend, library.
2. What functionalities does it handle? E.g., This is an API for a system that generates invoices.
3. What is the context? A bit of additional context, you can never have too much. E.g., These invoices are then sent to clients, (...). We use the standard template from the company's design book here.
4. Who are the stakeholders - who are the recipients of this project?
5. What business problem does this project solve?
6. Who are its end users?

The summary should be written in the domain language of the given problem. What does this mean? Well, its description, the description of the project's functionality should contain vocabulary from the field we're working in. If we're doing a project about tractors, let's use tractor terminology. When among crows, caw like them. This is the preferred approach compared to using pure, dry technical jargon.

12.1.5 Technology Stack

In this section, we describe the most important technologies used in creating the project. Dependencies, external applications used, services, etc.

This allows the reader to quickly familiarize themselves with the technological choices made in the project, used to solve the given problem.

Technologies should be briefly described, with appropriate materials linked for the reader's convenience.

12.1.6 Local Environment Setup Instructions

Here we place the steps needed to set up the environment locally. Additionally, this is where we place instructions for frequently used operations, what commands are most commonly used, such as clearing the database or creating it, migrations, etc. This is also where knowledge that is very specific to the given project lands, such as how localization, internationalization was solved using, for example, PhraseApp or OneSky.

I recommend describing this section particularly well, keeping in mind less technical users who might need to set up the environment locally for testing purposes. Sometimes these are non-technical people, testers, stakeholders, product owners, etc. They also deserve the possibility of having a local environment. Additionally, the entire environment setup process should be as automated as possible.

12.1.7 Deployment

In a few sentences, concisely and briefly, describe how the application is deployed, in what environment it lives, where to look for a more detailed description of the architecture, and so on. Add a few words about CI/CD.

12.1.8 Authors

List of main people in the project. Useful when jumping to a new project and quickly wanting to establish who holds the context and who is best to ask about things because they have the most knowledge about the project.

12.1.9 Summary

The README file is an important and integral part of the system. Now you know how it should look.

12.2 pdoc3

Automatic generation of the most important thing in the world - documentation. This tool plus extensive and sensible docstrings == happy developer.

For reading: https://pdoc3.github.io/pdoc/

12.3 PyCharm/Visual Studio Code

The best IDE/editor for Python. I'm personally #teamPyCharm, but Visual Studio Code also does the job.

12.4 Note Taking

There are a million tools for this. I recommend Notion. Other options?

For reading: https://bootcamp.uxdesign.cc/i-tried-40-project-note-taking-apps-what-i-chose-and-my-top-10-list-1d39d41852e4

12.5 Pyenv, Poetry, and Other Rascals

Pyenv, Poetry, and other rascals - modern dependency and Python version management, or about contemporary Python versions, environments, and dependency management.

12.5.0.1 PIP Pip is a tool that most of you should already know. It's used to install packages used in Python development and has been included by default with Python for several versions. But what exactly does it mean to install packages?

In short, pip provides tools for downloading packages from the Python Package Index - PyPI. This is the default Python package index where almost anyone can add packages. Default is a good word because pip allows us to use different indices. So, for example, your company might have its own hosted version of packages and then use it as a private version of PyPI. This allows for better package verification, only private network connections during CI/CD/development processes. This is quite an interesting option, especially considering recent malicious attacks on popular Python open-source packages.

12.5.0.2 Package Index What exactly is a package index?

Actually, it's nothing complicated. It's simply an HTTP server, let's say, that provides a list of Python packages/packages - packages and certain metadata about them. Nothing more.

An interesting homework task to experiment with something new: try implementing your own version of PyPI and add some features to it, such as token-protected access to packages or even more - tokens with granular permissions functionality.

12.5.0.3 Default Package Installation Usually, we have one, maximum two backward-incompatible Python versions installed on our computer. In the past, this was Python2 & Python3, currently most of the time only Python3 is installed as Python2 has died, perished, disappeared into the depths of the past.

Anyway. This means that in the Dark Ages or by default, packages were installed globally, for the entire system. This is bad for many reasons. As for what package installation means, in a very big nutshell, it's nothing more than downloading a Python code package with a specific structure, which gets downloaded and placed in a given Python installation directory, with additional possible steps in between.

What if project A requires package Z in version 1.0.0, but project B requires package Z in version 2.0.0? Would you reinstall this package every time you switch between different projects?

12.5.0.4 virtualenv To combat the problem described in the previous paragraph -> packages installing globally, virtualenv appeared. In short, it's something that allows us to "create" another, "instance" of Python installation. For example, just for a given project instead of system-wide.

This way we can have different versions of Python packages for different projects.

A subset of virtualenv's functionality is integrated into the default CPython installation since version 3.3 as the venv module.

12.5.0.5 Poetry What if pip and virtualenv had a child that was also on steroids? Well, we'd get Poetry.

The problem with pip is usually dependency version management. Even if we know that our project A requires package Z in version 1.0.0, usually at first glance pip doesn't tell us about the dependencies of that package Z. This introduces the possibility of problems when your project reaches a point where it has a few more packages installed. Because these packages also have dependencies, and their dependencies also have dependencies. Dependencyception.

Usually, it's not dependency hell like in JS worlds, but at some point, it can also become a bit sneaky if you only pin dependencies at the top level. And at some point, it's almost guaranteed that there will be problems with this. And besides - if versions of these dependencies aren't guaranteed by default, what about debugging? I mean, one build might have version 1.2.3 of some dependency, but another build, done 10 minutes earlier, might have 1.2.2, if versions aren't resolved in a deterministic, guaranteed way, this enables nasty bugs to appear.

It's also a security threat because if you don't know exactly what version of a dependency you have, a malicious version might find its way without our explicit knowledge, which is quite an easy way to shoot yourself in the foot. How to remedy this?

We have something called dependency resolving and dependency locking. Basically, it's just the process of making sure we know the dependencies of our dependencies and their dependencies, and we have a clear list of their versions, usually signed with a hash. This allows for something called deterministic builds, which is one of the key elements of modern CI/CD and applications that follow the Twelve-factor app pattern.

This is exactly what Poetry does. And it does it pre-elegantly.

Besides, while we're at it, Poetry also makes project management easier, handles creating and managing virtualenvs for you, and enables easier, more centralized project configuration through the introduction of pyproject.toml. pyproject.toml is now usually the new standard configuration file for Python projects. And it also makes building packages easier because it can bundle your Python code and publish it in the package index of your choice, e.g., in PyPI.

Generally speaking, Poetry is pretty cool.

12.5.0.6 Pyenv Python is a peculiar little animal that sheds its skin from time to time. This means that Python itself, just like our dependencies, also has its versions. Each version contains new features, various improvements. Some of them are sometimes even backward-incompatible. By default, it's not trivial to install different Python versions and manage them so they don't conflict with each other.

But why do this at all? Well, just like with dependencies. One project might depend on Python 3.10, another on 2.7, and yet another on 3.12. We need something like virtualenv that would provide isolation, but instead of at the project level, at the system level and not for packages but for Python versions.

How to do this? With pyenv. Pyenv has been enriched with a neat plugin that allows us to create such virtualenvs, but with different Python versions/interpretations. Pyenv + pyenv-virtualenv integrates beautifully with Poetry.

Anyway. We have pyenv-virtualenv, which is a wrapper for pyenv, which in turn is a wrapper around Python version management, working with Poetry, which is a wrapper for pip and pip-tools, integrated with virtualenv, which is also a kind of wrapper.

So we have a wrapper of a wrapper working on a wrapper of a wrapper. Wrapperception, dependencyception, wild, martens, raccoons.

12.5.1 Piptools

If your project is simple enough or you don't want to bother with all this, you can use pip-tools to pin your dependencies and be done with it. pip-tools is good enough for some projects, while I, for convenience and other things that Poetry offers, use it practically everywhere I can. Convention over configuration.

12.6 Python Tooling

In Python, sometimes you need to hit it with tooling. Since when do I sound like MMA bros on the scene? Something went wrong with writing this paragraph. Let's start over.

Formatting and static analysis of Python code, or its tooling, is an important element of the lazy person's approach to ensuring code quality. These are things that do and take care of quality for us, to make work easier. Let's talk a bit about them, but first about Pipelines.

12.6.1 Pipelines

What are they and why do we need them? Automation of things? Pipelines to the rescue. How strange that sounds in Polish. When we want to take care of our Python code quality, we usually want to take care of things like formatting, consistent import patterns, security, and keeping our standards up to date. If we want to do this in our repo/in the cloud automatically, we can use pipelines.

Pipelines are simply a set of steps that make up our CI/CD process. It's more or less just a piece of code that performs certain steps for us. Usually, pipelines are defined as a YAML file that specifies what steps/actions we want to take as part of our CI/CD process, i.e., analyzing, checking quality, formatting our code, and building/deploying it.

Among the most commonly known tools for this in the cloud are: GitHub Actions, GitLab CI/CD, Bitbucket Pipelines, CircleCI, Azure DevOps. Usually, these are things that run when, for example, we create a merge/pull request, push some code to the repo, merge one branch into another. They run various checks, building, tests, and so on.

The flow looks like this:

Trigger is received (e.g., Branch is pushed to repo) -> pipeline is run -> various checks/actions are performed -> based on this the pipeline may succeed or fail.

Besides pipelines being in the cloud, I think some of their parts are also an integral part of local development. Mainly parts related to quality control/formatting/etc.

12.6.2 What Makes Code Good?

Currently, the trend in Python is to take care of certain things that, while not crucial, over time contribute to the quality, readability, and maintainability of the project. At a high level, in my opinion, every piece of Python code would gain something by having:

1. Consistent formatting
2. Ordered imports that are divided into sections
3. Absolute imports instead of relative ones
4. Using modern standards that comply with new conventions
5. No unused imports and unused variables
6. Security/vulnerability scanning

12.6.3 The Dark Side of the Force - Black

Time to go to the dark side of the force.

12.6.3.1 A Few Words About Formatting and Black More often than in projects that aren't as automated as they could and should be, you can find people in pull requests arguing about which formatting is better. How to change formatting? Which is better? Which is more compliant with PEP8?

This can be a nightmare that is so counterproductive that it doesn't even fit in corporate.

To get rid of such problems and have it taken care of for us, in Python we have something called Black. Black is a code formatter that, well, just formats code for you. You can make Black automatically format your code before committing it. This way you can avoid all kinds of disputes about PEP8 and code formatting preferences by reviewers/authors, which makes the entire project have a consistent formatting pattern, making it easier to read and so on. The easier the code is to read, the better. The lazy person's approach. If you know what to expect, you won't be surprised. The less on your mind, the better.

12.6.3.2 ' vs " One thing worth noting is the fact that Python as a language allows the use of both ' and " to mark strings. Black by default prefers double quotes over single ones. Why? Readability, the use of single quotes in English and the need to escape it every time we use it inside our strings isn't very nice. That plus it's harder to make a mistake.

And so on. You can argue here, I stand on the side of double quotes because IMO it's a better approach. Readability is king.

12.6.4 Isort

Have you heard about sorting imports? It makes sense despite appearances and isn't a whim at all. Why should you sort your imports appropriately? The bigger the project we're working on, usually the more things we import from other parts of the code.

Over time these imports can become indecently large/numerous. Often they do. Isort is something that helps us with this, optimizing our imports, sorting them appropriately, alphabetically, grouping them in sections, and so on. I know it might look like a trifle, but it's these small things that add to the overall code quality. Google yourself an example of code changes after isort and see how it looks.

12.6.5 Imports Like Vodka - Absolute

The new standard in Python is using absolute imports. Why this is the case you can read on your own. There have been many debates on this topic, the result of which is: we prefer absolute imports whenever possible. They make for less ambiguity and provide clearer distinction of what we're actually using, from which package. We also have a tool for this purpose, which is absolufy-imports. This tool is particularly useful in older projects where you might need to fix imports in many files to match the new convention. This tool does it for you.

This:

```python
from .notifications.some_important_file import SomeClass
from .another_important_file import AnotherClass
```

Gets changed to this:

```python
from em.jobs.notifications.some_important_file import SomeClass
from em.jobs.notifications.another_important_file import AnotherClass
```

12.6.6 Bandit

Static analysis of our code for potential security vulnerabilities.

12.6.6.1 Why You Sometimes Need Bandits in Your Life When writing our code we should keep security in mind. Unless sometimes you want to expose your company to potential loss of millions. I'm exaggerating with this example, but still. Security is important. Somehow we can make simple mistakes due to forgetfulness and neglect that could have been prevented in another way. To remind us of this and warn us, there are various tools that can be used.

Among them is Bandit. Bandit is a static analysis tool that scans your code looking for potentially dangerous code fragments and warns you about them. When you run Bandit on your code, you'll get a report and a list of places in the code where there are potential problems.

12.6.7 autoflake

The less you have...

12.6.7.1 Slimming Down Code Like Putin Slims Down Citizens Sometimes it happens that in our code we can have unused import declarations or unused variables. It happens to the best of us. To automatically take care of this, we might want to include autoflake in our projects. This is a tool that simply takes care of this - removes unused imports and variables. There's no magic here.

12.6.8 pyupgrade

This piece of software automatically updates some old syntax patterns to new ones. That's all.

12.6.9 bumpversion

There's something we call semantic versioning or semver. It's a convention that tells us to version our code according to the following pattern: MAJOR.MINOR.PATCH

For example: v0.2.12

The major part is increased when we move to major rollouts that change many things.

Minor is increased when we make normal releases, e.g., with larger features.

Patch is something we use for smaller features, patches, fixes, etc. This element grows the fastest.

So we don't have to manage this manually, we have a tool called bumpversion. It updates the version, creates a commit with changes, creates a git tag, and so on, all automatically. It's a neat tool that you can have in your CI/CD. This makes version management, creating changelogs, filtering commits and noticing changes, bugs, versioning packages/api, etc., easier. You can see an example commit history and bumpversion usage in the commit history of my project - braindead. Take a look.

Do we run all this manually? No. We want to be lazy.

12.6.10 Git Hooks & pre-commit

Automate boring tasks, live long and prosper.

12.6.10.1 Git Hooks and pre-commit If you want all this to happen automatically, you can create git-hooks that run, for example, during commit or before commit. One way is to simply create a .pre-commit file and place it in the .git folder and use, for example, Makefile or use something like the pre-commit tool. It's a nice, handy tool that takes care of this for you. You need to install it and create a config for it to tell it which things to do before commit. There's no magic here.

13 Databases

A bit about what databases are and what's worth reading about them.

What are databases anyway? Simply put, they are places where we want to store data in a relatively permanent way.

Databases come in various forms and are divided into different categories. We'll focus on Relational databases and the query language that has dominated database systems for decades - SQL. There are databases that don't use SQL, but that's beyond the scope of this book.

Let me point out right away that because there are different types of databases, sometimes there are heated debates about which one is better, etc. Something like Linux vs Windows, which Linux distro is best, Apples or Plums, etc.

The truth is that it's foolish to praise only one solution in every situation. Each database has its own applications and certain niches where it works best. Well, maybe except for OracleDB, which should have died long ago and stopped torturing people with its existence. Anyway.

13.1 SQL

SQL is simply a database language that allows you to query data. That's all you need to know. Additionally, read about simple queries: SELECT, UPDATE, INSERT, WHERE clause, and others.

Examples to analyze if you don't feel like Googling yourself:

```
SELECT column1, column2, ... FROM table_name;
SELECT * FROM table_name;
SELECT DISTINCT column1, column2, ... FROM table_name;
DELETE FROM table_name WHERE condition;
DELETE FROM table_name WHERE condition;
SELECT column1, column2, ... FROM table_name WHERE condition;
SELECT column1, column2, ... FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

13.2 Relational Databases

These are generally databases where we have some formal structure, say, through defined tables (most commonly) and where we can define relationships between tables. To make things easier, I'll add a small general summary:

SQL (Structured Query Language) is a language designed for communicating with relational databases. It allows operations such as creating, modifying, and querying data stored in the database.

The advantages of SQL are:

1. Simplicity and ease of use - SQL is simple and intuitive, and its syntax is easy to understand.

2. Versatility - SQL can be used with many different types of databases, such as MySQL, Oracle, Microsoft SQL Server, and many others.
3. Performance - SQL is optimized for performing fast queries on large datasets.
4. Standardization - SQL is fairly standardized, meaning different databases will have similar syntax and set of functions.

SQL is often used in web applications to manage data stored in databases, such as user registration, storing products in an online store, and creating reports.

13.2.1 Tables

Tables are simply where we store data. Tables have columns and records/rows.

A table is a data structure in a database that stores data in the form of rows and columns. Each row in a table represents a single data record, and the table's columns define different fields that make up that record.

For example, if we're creating an employee database, we could create a table containing information about employees, such as their first name, last name, position, salary, and hire date. The table might look something like this:

First Name	Last Name	Position	Salary	Hire Date
John	Smith	Director	10000	2020-01-01
Anna	Johnson	Manager	8000	2020-03-01
Mark	Williams	Engineer	6000	2020-05-01
Agatha	Brown	Assistant	4000	2020-07-01

A table can contain any number of records and fields, and the data stored in it can be of various types, such as numbers, text, dates, etc. Unlike in Python, in SQL we define data types and this is mandatory.

A table is the basic data structure in a database and is used to store and provide access to data in applications.

For reading: 1. https://www.plukasiewicz.net/SQL/Introduction (a bit formal, but might be helpful)

13.2.2 Indexes

What are indexes? Indexes are special lookup tables that allow us to speed up data retrieval. They're like pointers that tell SQL where to jump and from where to read a given value. Like a table of contents in a book.

Thanks to them, SELECTs and WHERE clauses are much faster. However, operations for adding and updating data will be slower - when adding data, besides the table itself, the index also needs to be updated, which causes additional overhead.

Indexes can be created on one or more columns, and they can be combined.

When should you think twice before creating an index?

- In the case of small tables
- In very frequently updated tables where data changes often
- When we have a lot of NULLs in some column

13.2.3 Relationships

Relationships describe connections between pairs of tables. They exist when two tables are connected by a primary or foreign key.

Each relationship is described by a specific type of relationship, the constraints between these two tables. The first example will be **one-to-many** (one to many/foreign key). An example of this could be the relationship buyer <-> invoice. An invoice has only one buyer by definition, while one buyer can have many invoices.

Another example is **many-to-many**. Here, let's use the example of a fan of an artist. One artist can have many fans. One fan can like many artists.

For reading: 1. https://developeronthego.pl/sql-schemat-bazy-danych/ 2. https://analityk.edu.pl/relacyjna-baza-danych-o-co-chodzi-z-tymi-relacjami-sql/

13.2.4 Normalization

Normalization is the process where we get rid of unnecessary data and establish relationships between tables.

This helps avoid data duplication and mess in the data.

Data normalization is the process of optimizing the database structure by separating data into smaller, more specialized tables, each containing only necessary data. The goal of normalization is to ensure that data is stored in the best possible way, i.e., to avoid redundancy and ensure data integrity.

Data normalization is performed in several steps, called normalization degrees. 1NF, 2NF, 3NF.

Data normalization has several benefits, such as:

1. Easier data management - with fewer tables and no redundancy, data is easier to manage, less mess.
2. Improved performance - smaller tables are easier to search and query, leading to better performance.

But more details later.

13.2.5 Transactions and Concurrency

With transactions, it's about sometimes wanting to group certain operations and perform them together. For example, imagine the example from the link below - a bank transfer.

What if during the execution of an operation that can be divided into the following steps:

1. Withdraw funds from account A and update account A's balance.
2. Add the withdrawn funds to account B and update account B's balance.

...there's a power outage, disk error, whatever? Well, client B wouldn't get the money and client A would lose it. Not cool. This is where transactions come in, allowing us to permanently record a group of operations before completing them. All or nothing.

If something fails along the way, the transaction is rolled back as if it never happened and everyone's happy.

Here it's also worth reading about SELECT FOR UPDATE and Race Condition. Important for concurrency.

For reading: https://mst.mimuw.edu.pl/lecture.php?lecture=bad&part=Ch7

13.2.6 Subscriptions/Notifications

Sometimes we want to inform interested parties about our changes, e.g., adding a record to the database. Imagine that we have notifications in our application and we inform users about promotions, app updates, or something similar.

There are advanced solutions, you can simply poll the API, but that's a mediocre solution.

There are appropriate protocols and 3rd party services that nicely solve this problem.

But the simplest is probably... LISTEN and NOTIFY from Postgres. Many people don't realize that Postgres has built-in notification support. But here it is. Postgres is a nice all-in-one solution. Anyway.

To check out: https://www.cybertec-postgresql.com/en/listen-notify-automatic-client-notification-in-postgresql/

13.2.7 Permissions and Security

Databases sometimes provide something like column/row-level security. What does this mean? Well, thanks to this we can determine which user can see which columns or even which records in which table in which database/schema.

Some DBs require us to install additional packages to achieve this, others have it by default, and some don't provide such granular control at all. Postgres, for example, does. Nice stuff.

To configure row-level security, you need to add access rules to the table that determine which rows are visible to specific users or groups of users. For example, you can create a rule that allows a user named "john" to see only those rows where the "department" field has the value "marketing".

Row-level security is a useful tool for limiting access to data in a database and can be used in combination with other security mechanisms, such as user permissions and roles.

13.2.8 Profiling

We can profile queries similarly to Python code. There are special tools for this. There's something called `EXPLAIN` and `ANALYZE`, there are statistics, there are profilers.

EXPLAIN allows us to examine in advance what a given database plans to do to execute a query. This is the so-called 'query plan'. It's worth remembering these two terms. In the case of Postgres, `EXPLAIN` generates a plan. `EXPLAIN ANALYZE` additionally executes it, providing actual data and statistics.

More to read: https://www.cybertec-postgresql.com/en/3-ways-to-detect-slow-queries-in-postgresql/

13.2.9 Column Order

Even the order of columns in, for example, Postgres matters and affects how quickly our query will execute. Huge impact even.

I'll summarize here the conclusions I would expect from a junior. I wouldn't require knowledge of the exact mechanism, but it would be nice to know that the number of columns in a table affects performance, which column we're processing also has an impact, a huge one. It's worth remembering that creating huge tables with hundreds of columns is not desirable. Sometimes you need to split them and find a compromise between convenience and performance.

To read: https://www.cybertec-postgresql.com/en/column-order-in-postgresql-does-matter/

13.2.10 Summary

Databases are an essential element of almost every system. It's worth knowing a bit about them. The dominant language in the database world is SQL. Knowledge of its basics probably won't hurt us and might help.

13.3 Tenants and What They Are

A bit about the tenant pattern in Django implemented using django-tenants and Postgres.

(Note: this piece was written several years ago by me and Dominik Szadego, who was my junior at the time <3)

13.3.1 Junior's Perspective

Hi! My name is Dominik, I'm a junior developer at thirty3.

The environment that will be demanding, but is probably the best for a beginner like me. A place where there is a mentor who will help me in difficult times, answer all my questions and show the way, tell me about the mistakes I make.

Does this make learning programming easier than before? Absolutely! Does it make it easy? No!

Today I would like to write a few words about my experience so far, new tasks, mentoring and so on, in a joint article with my mentor - Olaf, and most importantly - about the tenancy pattern in software architecture.

I would say that the learning process can be divided into two parts:

understanding a new topic (technology, tool, etc.), which ends with having a general idea of how things are done, which allows building things based on example, making small modifications to existing things and so on;

an endless process of improvement that leads to being able to create complex things from scratch;

For me, being a junior programmer means that I will often face problems that will require me to perform the first part - learning something new to solve them. This is exactly how I could describe my first month at thirty3 - being outside my programming comfort zone and doing things I had never done before. Which is AWESOME.

13.3.1.1 First Days The first days in a new job are always difficult and I swear that when I was setting up my work environment in previous companies, something always went wrong - I was missing some tools, packages, getting strange errors. Fortunately, running the project at thirty3 for the first time was completely the opposite.

In this case, the difference-maker was the combination of Docker and Makefile. All I had to do was download docker (and docker-compose) on my computer and follow the damn README.md to have everything ready and working. The application works. Documentation, practices are defined, code is clear and easy to understand, tests are there. It was a breeze.

At least until a certain point. It didn't take long before I was hit by the multi-tenant architecture. What is this. Imagine that you have an application used by many companies (tenants). You would like to be sure that there is no accidental data leakage between companies, while at the same time ensuring that the architecture is scalable and efficient.

13.3.1.2 Tenants to the Rescue At thirty3 we used django-tenants to solve this problem, at least in several cases. There is one database instance to store data for the application, but many schemas - one for each tenant (company). This creates a logical separation between data. Before I jump to examples, let me try to understand this concept and the beginnings, so, hopefully, you will immediately notice my mistakes and understand how it works. Let's assume that we have a very simple Django project that allows companies to create projects they will work on.

We need two Django apps:

```
companies
projects
```

And models declared in appropriate models.py files:

```python
# `companies/models.py`
class Company(TenantMixin):
    name = models.CharField(max_length=255)

# `projects/models.py`
class Project:
    name = models.CharField(max_length=255)
    is_paid = models.BooleanField()
```

The behavior we would like to have is that each Company has its own Projects, which are not available to other companies.

As I mentioned earlier, each tenant has its own schema in the database - like a database within a database. It is created along with the creation of a model that inherits from TenantMixin (Company). The thing that allows us to distinguish tenants is a unique schema_name attribute (TenantMixin attribute), which we must provide when creating each Company object. (Olaf's note: this doesn't have to be the schema name - it can be an ID or almost anything really, as long as it's unique. Generally schema_name is just metadata that allows us to know where to look in the db).

In addition, we need to create a specific schema called “public”, whose purpose (O: Actually it’s there in postgres by default, we just create a model in our tenants table and set up public as a shared schema) is to store global data not specific to a given tenant/company and all tenant schema_names.

13.3.1.3 Tenants in Django The question we should ask ourselves now is: how does Django know which data should be stored in the “public” schema and which in the schemas for specific tenants? This is handled in the settings file by setting the SHARED_APPS and TENANT_APPS variables. These are lists of Django apps (similar to INSTALLED_APPS). Placing an app in e.g. TENANT_APPS will mean that tables for Models from this app will be created in each tenant’s schema. On the other hand, if we add our app to the SHARED_APPS list, then as in the case of other apps where there are no tenants, tables will be created in the “public” schema.

```
SHARED_APPS = ["companies", ...]
TENANT_APPS = ["projects", ...]
```

Another question is, how does Django know on which tenant’s schema to perform actions? Tenants are identified by URL, e.g. a request to URL “tenant.something.com” will cause the hostname to be looked up in the appropriate table in the “public” schema. If a match is found, the schema context is updated, which means that queries will be performed in the matched tenant’s schema. Django-tenants provides several tools for setting schemas from the code perspective.

```
with schema_context(schema_name):
    # queries will be performed against the schema "schema_name"
```

or

```
with tenant_context(tenant_object):
    # queries will pe performed against the schema of tenant_object.
```

Knowing all this, let’s look at the following code fragments to identify some mistakes I made during the thought process.

```
class TenantsTestCase(BaseTenantTestCase):
    def test_tenants_example(self):
        companies = Company.objects.all()
        ...
```

The expected behavior for me would be to get all companies, and yet the result was an empty QuerySet. Ok, maybe I need to create some tenant before displaying, logical, let’s try.

```
class TenantsTestCase(BaseTenantTestCase):
    def setUp(self):
        Company.objects.create(name="Test company", schema_name="test_schema")
    def test_tenants_example(self):
        companies = Company.objects.all()
```

This time I got an error
`Cannot get list of tenants while in tenant context, enter public schema.`
I asked myself "What's happening?", but I managed to find somewhere the use of schema_context. So I tried:

```python
class TenantsTestCase(BaseTenantTestCase):
    def setUp(self):
        Company.objects.create(name="Test company", schema_name="test_schema")
    def test_tenants_example(self):
        with schema_context("public"):
            companies = Company.objects.all()
```

Great, this time there's no error. Anyway, the companies variable is still an empty QuerySet. Last attempt:

```python
class TenantsTestCase(BaseTenantTestCase):
    def setUp(self):
        with schema_context("public"):
            Company.objects.create(name="Test company", schema_name="test_schema")
    def test_tenants_example(self):
        with schema_context("public"):
            companies = Company.objects.all()
```

Finally, this time I got a QuerySet with two Company objects. But wait, I created one. Time to put it all together.

It turns out that when we run tests with Django-tenants, a new tenant is created with schema_name "test" and all queries are performed against this schema unless we switch it. (O: At least in our case, because we use FastTenant case -> otherwise tenants are created e.g. per TestCase, which takes time because migrations are applied per tenant etc.)

```python
class TenantsTestCase(BaseTenantTestCase):
    def setUp(self):
        # schema_context = "test"
        with schema_context("public"):
            # schema_context = "public"
            Company.objects.create(name="Test company", schema_name="test_schema")
    def test_tenants_example(self):
        # schema_context = "test"
        with schema_context("public"):
            # schema_context = "public"
            companies = Company.objects.all()
        # schema_context = "test"
```

Now remember that Company, which is our tenant object, is stored in the "public" schema, so the empty QuerySets we got earlier were correct because we were trying to find the Company object in schemas that don't contain them. Going further, creating a

Project object must take place in the context of a specific tenant's schema because that's where its tables are stored.

```
class TenantsTestCase(BaseTenantTestCase):
    def test_tenants_example(self):
       # Here we can create project, as we are in context of "test"
       # tenant
       Project.objects.create(name="Test project", is_paid=True)
       with schema_context("public"):
            # Here we can not create Project, we are in "public"
            # context,no tables for Project here
            companies = Company.objects.all()
```

For me, working with tenants resolves around tracking how the context changes to always know what queries I can perform and what effects to expect.

13.3.2 Old-timer's Perspective

Ok. Enough from Dominik's perspective. Now it's my turn. I'll give you a bit of insight from a broader perspective. What you've read so far is Dominik's understanding of the tenant concept and how we used it at thirty3. It's more or less correct, some things are oversimplified, but the general idea is somewhere there. I'm a bit proud of him, it took me much longer to understand certain things. Enough patting on the back. Now I'll try to give you more information about the decision-making process we had when we started using tenants, why we use them and why you might want to do it too.

Let's start.

13.3.2.1 What are tenants? First thing - tenants. What is this? It's a concept, most often used in e.g. SaaS products, that simplifying a bit, they are like your clients. At least that's how I like to think about it. Something like when instead of creating a custom solution for a given Company, which is used only by that company, you create a general solution where the company is just like a user.

Some things are defined globally in the DB and shared between users/companies, other things are defined and should be available only to that specific Company and so on. In a normal case, when only that company will use the application, you don't have to think about it much. The problem arises when you want to globalize this application and have many companies. All of them have some private data, some public. This data should be separated and unavailable to other companies using the given SaaS. You need another layer of abstraction that logically ties or encapsulates this company data. Aw shiet, here go tenants. What are other benefits?

13.3.2.2 SaaS Scaling As time passes and our applications develop, when you start getting clients who aren't your closest family or investors, things start to get complicated. Privacy suddenly becomes important. Data breaches/leaks are costly. Then the product starts gaining traction, your user base grows, optimization becomes a problem. This happens in almost every successful product.

It's good to think about these problems and prepare for them, but only as much as needed, so as not to over-engineer. In our case, in most cases, we decided to use the Tenant pattern for this purpose, using DB schemas to implement the plan. Thanks to this, it's harder for all clients' data to leak or for one client to get access to another client's sensitive data, easier to scale our application, without overburdening our working time.

13.3.2.3 Database Scaling Methods What is the factor that most often limits us, at least in most applications? DB. What are the ways to scale DB? Horizontal and vertical. Vertical means you have one DB, to which you simply throw more resources - better hardware. Such scaling has its limits. When you hit them, no matter how much money you have - that's it. Can you do something about it?

This is where horizontal scaling comes to the rescue, i.e. using more machines/DBs instead of one. However, this is quite tricky - just putting up a second DB next to it won't do anything. Suddenly things like master-slave patterns, data consistency, node networks and so on come into play. Quite a complex topic in my opinion.

Of course, this method also has limitations, but often they are much greater than the limitations of a vertically scaled system. So yes - managing tenants in a SaaS-like product can be done in many different ways. The first is DB per client. Here we would probably have one larger DB with globally shared things in the application, and then smaller (or larger) DBs with data unique to the client.

The second is Schema per client, i.e. one database with like 'mini databases' inside.

The third is custom permissions and relationships in tables, for example with all clients' data placed in one schema, one db.

The first is costly and troublesome to manage on a smaller scale.

The third often ends up with messy DB tables, privacy concerns and permission nightmare.

The second however... Well, it almost doesn't impose costs on the situation where you would have a regular single schema/DB architecture - it's not a SaaS architecture, but it facilitates scaling and separating client data through DB abstraction - instead of having many DBs which are troublesome to manage, it uses one DB as if there were "many", at least in some sense.

That's why we decided on this solution. And honestly, we're quite happy with the results.

A big advantage of tenants is also the fact that queries remain simple. Query for invoices only from a given company? Just set the appropriate tenant context and that's it.

13.3.2.4 Different Tenant Management Methods The so-called search path, which we set for Postgres queries using our db router, can be set in many ways. The traditional tenant pattern uses subdomains as a way to identify tenant- e.g. x.myproduct.com will look up tenant x. That's one way. Remember to prevent users from registering tenants with names of commonly used subdomains e.g.`ftp, mail, static` and so on, otherwise you might be in for an unpleasant surprise. Also remember to have certificates that have a wildcard for subdomains - otherwise you'll be without SSL for your tenants' subdomains, which is quite bad.

Another solution is for example to put it as part of the url, but not a subdomain. For example:`api.example.com/v1/tenant/someendpoint`. We used the latter.

13.3.2.5 What Tenants Gave Us We achieved:

1. zero additional costs of managing additional infra
2. zero additional infra
3. ease of scaling -> change the way of setting the search path for db router and done, horizontal scaling in three minutes
4. client data is logically separated from each other
5. queries remain simple, same as permissions and tables

Anyway. Plus minus that's how tenants look and work.
PS: minuses of having me as a mentor - you'll probably start writing.

13.4 ORM

What is ORM? Well, a few words about this useful tool.

13.4.1 What is ORM?

ORM, or Object-Relational Mapping. Such a tool that converts SQL values into Python objects that are easier to handle, using appropriate 'classes' or 'mappings'. It's a collection of useful methods, conveniences and abstractions.

13.4.2 ORM vs Pure SQL

The difference between ORM and the traditional approach where we write pure SQL ourselves is diametrical.

13.4.2.1 More Abstraction The first difference will be the fact that ORM hides many things from us, providing a layer of abstraction above SQL itself. It does this at the cost of performance and at the cost of assuming that the programmer knows how ORM works, as their creators make certain decisions for us. These decisions affect how ORM works and how it creates queries. Sometimes not knowing them can make us shoot ourselves in the foot, unknowingly. Ignorance can sometimes hurt. Plus that performance.

13.4.2.2 Lower Performance It's not some enormous overhead, but with complex queries you'll probably have to resort to writing pure SQL. It might also turn out that something that takes a minute in pure SQL, in ORM, due to e.g. the unusual nature of the problem, might take hours. And vice versa. It happens.

13.4.2.3 Still Worth It Nevertheless, for most, the vast majority, which includes you, because otherwise you wouldn't be reading this book, ORM will be a significant convenience. However, as I always do, I must point out that it's worth knowing and knowing at least the basics of SQL and knowing how ORM works under the hood, at least very very superficially.

13.4.2.4 Debugging/Profiling ORM

ORM is additionally harder to debug and profile - queries are generated by the engine, often they can be not very friendly when it comes to readability, for humans plus somewhat non-optimal when it comes to performance. Again, for 99% of cases it won't matter, because a query returning a list of users will be just as simple here and there.

13.4.2.5 Serialization from/to Python Objects

ORM also provides us with a layer of abstraction in the form of mapping data from the database to Python objects. Data serialization and also their validation. In the case of writing SQL queries manually, it often becomes your responsibility to check if the data is correct, their interpretation or conversion. If it's about ORM, it either provides a range of tools that help us with this or even takes care of it for us. Something nice.

13.4.3 Summary

ORM is simply a form of abstraction and simplification of database interaction from the level of a given language. It has its advantages and disadvantages, you need to know them to make a conscious choice. In 95% of cases it will make your life easier, but the remaining 5% is probably beyond the scope of this book intended for junior wannabes.

14 Internet

Here we'll discuss various issues related to networks, the Internet, and everything else. Even though as a junior-python wannabe we're most concerned with Python itself and programming, we need to remember that the code we write and then run doesn't operate in a vacuum.

All our web apps, programs, etc. are run in some specific environment. This environment affects how our programs are executed and how they work. Additionally, we often need to interact with it in some way, often bidirectionally. What does this mean? Well, besides Python itself, it's worth knowing about the whole environment where it runs and so on, because otherwise, you might sometimes shoot yourself in the foot. Plus, knowledge about the environment and related things that we use directly or indirectly, often without being aware of it, meaning simply other components of the 'System' that we design, create, or maintain, are its integral part and something that affects our work and our code.

By System here, I mean some arrangement, set of elements. In IT, this is usually, for example, our web app, the servers where it runs, the client, etc.

Let's talk about this whole system and environment.

14.1 Request Journey

Most web applications work in a client-server model. The client makes requests to the server, the server returns a response.

But how does it happen that we're able to send this request? What happens when you type `grski.pl` in the browser bar and then see my blog?

Well, the matter looks like this.

Assuming you're connected to the Internet, your request will be processed by your ISP (Internet Service Provider), which will hit something called DNS. DNS is the Domain Name Server, like a phone book that contains mappings of domains/addresses like grski.pl to locations (IP) on the network.

What does this mean? Every server has its more or less unique IP. The so-called IP address. It's like a residential address.

IP looks like this: **172.16.254.1**. It's a 32-bit number, at least in the case of IPv4 standard. Something like this is hard to remember, right? For me it is. But `grski.pl` isn't. Hence this mapping and domains came about. Domains make it easier to remember and make life easier for users.

I'll throw in an interesting fact. If IPv4 defines an IP address as a 32-bit number, then the question for you is, what problem might arise here in today's times? Well, a 32-bit number is small by today's standards. Think about how many devices are connected to the Internet, most of them have unique public addresses. Holy moly. Generally, we're approaching a point where there won't be free unique IP addresses. Sad situation.

Therefore, IPv6 was created, which solves this problem. Example of IPv6: **2001:0db8:85a3:0000:0000:8a2e:0370:7334**. Here the size is already 128 bits. 2 to the power of 128 gives a ton of options, this address space will last us for a while.

Alright, let's get back to the request though.

Currently we have: your browser (client) -> ISP -> DNS -> Server.

Then on the server there are often Load Balancers/Proxy, which are pieces of software that appropriately direct requests coming to them to matching local resources/services/APIs etc.

From LB/Proxy the request goes to the target service. Then we receive a response and done.

There might be many other services in between like cache, CDN etc., but I presented the shortened version here. And speaking of CDN. . .

14.2 CDN

What is CDN and why is it important? It's actually thanks to them that we can use mobile networks without problems, pay less for Internet, it's thanks to CDNs that your funny cat video on wykop doesn't stutter, and your operator can handle the current number of clients instead of, say, half. I know, I'm exaggerating a bit, but only a little bit. So what is it?

14.2.1 Introduction

First though, a handful of information to gain some perspective. We live in times where existence without the Internet is practically impossible, or at least very inconvenient. Most of today's luxuries somehow base on/use this invention. Often, however, we don't realize how huge this Internet is and how quickly it's growing, let me explain.

In 2018, we crossed another barrier - that's when Earth hit a new record of internet users - 4 billion people using the Internet, which is as much as 53% of the population. This is an amazing increase, considering that just 4 years earlier there were about 2.4 billion Internet users.

In 2016, 44 billion GB of data flew through the Internet per day. Taking into account that there were significantly fewer users then, considering users in 2018, this gives us about 51 billion GB per day. Of course, this is a heavily underestimated result, because not only are users increasing, but they're also consuming more and more data, but for the purposes of this article it's enough, as it gives us some picture. Currently it's 2022 and about 5 billion. 62.5% of the population.

The average smartphone user consumes 2.9 GB of mobile transfer during a month (data from January 2018). That's 50% more than a year earlier.. The network is growing at an alarming rate. Why alarming? Well, while for cable connection there isn't such a tragedy, because you can add a cable or two, although this is also all complicated and costly, the real problem arises with mobile data network, as it's heavily limited by physics, and considering continuous growth. . . Well, we have something to worry about a bit or our network might get a bit clogged.

14.2.2 And Then Enters CDN, All in White

Yes. The whole situation is mitigated by CDN. What is this? These are servers that cache the most popular content on the Internet to generally offload the network, reduce loading times and prevent certain problems, improve security. These servers are scattered across the Earth in geographically strategic locations for the network.

In the network, we have content providers. This content is various, text, pictures, videos, multimedia. Content providers place them on their websites, servers and it's cool. When you want to read or watch something on the Internet, your device connects through the Internet with the content provider's server and transmits specific content to you. All cool, right? Well, no.

The problem appears when this data and users increase worldwide. The problem stems from the Internet's architecture. When you watch an episode of your favorite series, your computer doesn't connect directly with the provider's server, no. Before that happens it must go through dozens of other devices that will direct it to the right place, same with the response from that server.

Imagine you're in an office and to handle a specific matter you need signatures from ten different officials and at the end also a signature from a supervisor in America, to whom there's a long queue. It takes a lot of time, energy and so on, right? Yes. Complicated matter overall. The role of CDN is to shorten this list of needed signatures to one official who is in the local office.

So like with servers - your request instead of beating its way to a server in Asia or America and tiring one server, will first ask a local guy who is in the next city. In 99% of cases he will be enough. In the remaining 1% we'll have to drag to America, but we'll handle the matter quickly on site, because thanks to help with handling petitioners locally, Uncle Sam has less to handle, making the queue much shorter.

14.2.3 Details

From the Content Provider's server to CDNs, certain data is transmitted and cached - what kind? Those that are most popular - it's very important to maintain mainly that data on CDNs that is most popular, because thanks to this CDNs take over most of the traffic, reducing network load achieving high hit rate.

It's a complicated process, because after all different content is popular in different regions, and how they will change is not trivial to predict.

The algorithms used here to predict what and where will be popular is really a very interesting matter and important - CDN space and resources are limited, so the choice of this content is difficult. An amazing example of such optimization and prediction of what will be popular right now can be observed on the example of Netflix and how they solve this.

14.2.4 What is hit rate, lifetime?

I used the term hit rate earlier. This is a term that determines what percentage of user requests can be processed by CDN and only CDN, and what needs help from the Content Provider's server. Currently, some can optimize their servers so well that the hitrate to cache can be even around 99%. Amazing results.

Additionally, there's also determining the time for which once uploaded content should be available - lifetime - after its expiration the cache is 'removed' from the server and new (or still the same if they're still popular)/updated data jumps in its place. It's completely different depending on the data, region, the service provider itself.

14.2.5 Is CDN one huge server?

No, it's often whole clusters of geographically distributed servers. Hundreds of thousands of machines that have a certain hierarchy and work according to it. How? More or less like this. The content provider's server is CP.

Then we have CD & LCF - that's like headquarters you could say.

Then there's CCF and under it CDPF. CCF is the local office, and CDPF is the official.

By default, when you make some request for content, it lands in CCF, CCF checks if what you need is somewhere in its resources, meaning on CDPF servers where cached content is kept. So in short it checks if what you're asking for is somewhere 'copied' on the local server.

If it's not on one, it goes to the next of the CDPFs under its control. What if it doesn't find it on any of its CDPFs? Then it reports the fact to CD & LCF, which asks the remaining CCFs in turn.

If each CCF states that this content isn't on CDPFs under their control? Then CD & LCF makes a request to the content creator's server, gets the data from there and keeps it locally. So the original server is bothered in a very small number of cases, thanks to which the server itself and its surrounding network is significantly offloaded, traffic gets scattered across local and distributed CCFs instead of being concentrated in one location.

It's partly thanks to such solutions (or similar) that GitHub with help from Akamai company, were able to handle the recent record DDOS attack directed against this popular platform, which in its peak phase reached the size of 1.35 Tbps - almost one and a half Tb per second. Amazing. This is why Wykop works somewhat. This is why Netflix doesn't clog the entire Internet.

14.2.6 Summary

There are many things thanks to which our days are easier, and we don't even know it. CDNs were probably something like that for most of you. Of course there are a lot of simplifications in the text, so bear with it.

14.3 Cache

What is cache? Cache is like a database, but with a somewhat different purpose. By default, cache keeps data for a specified amount of time, usually quite short, relative to databases which sometimes keep data permanently.

So cache is like a database with a short expiration date, in which we store remembered results of computations, those that are usually costly and only those that concern reading and not writing to the database for example.

So if we have some view/function, anything that takes an argument, in our case it will be some request, then for similar or the same ones, cache will return the result 'from memory' instead of calculating/fetching anew.

For reading:

1. https://www.techtarget.com/searchstorage/definition/cache
2. https://realpython.com/lru-cache-python/
3. https://realpython.com/python-memcache-efficient-caching/

14.4 Cloud

AWS, Azure, GCP are cloud service providers. What does this mean? What is the cloud? Cloud is simply like a server room, but at someone else's house. Someone else worries about certain things.

For an appropriate fee, cloud providers handle certain things for us, provide additional services, take care of most things.

These are often such conveniences where in exchange for some cost and the fact that cloud providers sometimes decide for us on certain issues, we shift the burden of taking care of some matters to an external company. In a nutshell.

AWS is the cloud from Amazon, Azure from Microsoft, and GCP from Google.

Which is better? The one your target employer uses. Basically though, they are similar to each other, only some service names change. Additionally however, one cloud will have a better toolset for specific tasks, another for others.

IMO if it's about corporations then probably mostly Azure/AWS.

Startups mainly AWS.

GCP is a mix.

Source: institute of data from ass.

I personally most often encountered AWS, but that's just me. I recommend getting familiar and playing a bit with setting up different services in the cloud on your own. Each of them offers programs where for registering we'll receive some amount of cash to burn on our games. Use it, because it's worth it. It will add independence to you and +10 to fame if you learn basic things from DevOps. And what is this whole devops thing? Link number 4. In short, it's a guy from infrastructure and handling servers, meaning the place where our applications are run.

Anyway.

For reading:

1. https://azure.microsoft.com/pl-pl/resources/cloud-computing-dictionary/what-is-the-cloud/
2. https://experience.dropbox.com/pl-pl/resources/what-is-the-cloud
3. https://devopsiarz.pl/kurs-ansible/ <- strongly additional. Ansible is software for automating certain tasks, so you don't have to click manually.
4. https://devopsiarz.pl/devops/kto-to-jest-devops-engineer/
5. https://en.wikipedia.org/wiki/Infrastructure_as_code
6. https://12factor.net/

14.5 Docker

What is this whole Docker thing? They write about it everywhere.

Well, docker is a tool for building and running containers, containerization. Like a virtual machine simulating a computer in a computer. The main difference lies however in that containers are muuuch more efficient when it comes to resources.

VMs set up the entire system and emulate everything from scratch. Docker uses ready components of your system which makes it much less resource-hungry.

Thanks to containers we can download ready 'images' that contain everything the application needs to run as well as the application itself.

Imagine that you have to manually install all dependencies, packages etc.

Now imagine the same on 50 servers, because you need to scale the application. Docker makes this easier for us. Once built complete image only requires running.

Currently it's standard that applications are containerized.

Containerizing applications with Docker has many advantages:

1. Reliability: Docker containers allow running applications independently of the environment, which means that the application works the same on different operating systems and infrastructures.
2. Flexibility: Docker containers are easy to move between different environments, which means you can easily extend the application to new platforms or move it to the cloud.
3. Resource savings: Docker containers are lighter than traditional virtual machines, which means you can run more applications on the same hardware.
4. Speed: Docker containers are faster than virtual machines because they don't require installing the operating system and all needed libraries.
5. Ease of use: Docker provides a tool for creating, sharing and running containers, which makes it easier to manage applications and their dependencies.

Generally speaking, containerizing applications with Docker helps increase reliability, flexibility, efficiency and ease of managing applications, which is particularly useful in a cloud environment.

For reading:

1. https://www.czarnaowca.it/2022/01/docker-tutorial-1-co-to-jest-docker-i-do-czego-jest-nam-potrzeby/
2. https://sii.pl/blog/docker-dla-programistow-co-to-jest/

14.6 Docker-compose

In telegraphic shortcut Docker Compose is like scripts on steroids that allow us to more easily manage several containers. Imagine that we have a database, broker and worker in docker in one application. Potentially a million Dockerfile files, lots of commands to type etc. Mess overall.

Here comes to help the docker-compose file and tool of the same name. It allows for easy management of all application containers as one complete system, instead of managing each container separately. Docker Compose also enables easy resolution of dependencies between containers, such as network connections and required data files.

14.7 Docker Hub

Docker Hub is a public platform for storing and sharing Docker images. It allows for easy sharing of images with other people and using ready-to-run images. Docker Hub is also integrated with the Docker Compose tool, which enables easy running of multiple containers simultaneously.

15 Recruitment

Everything related to recruitment.

15.1 How the Process Looks Like

In general, recruitments look quite different, but they usually consist of similar building blocks. Some are quite short, others long, but the most popular pattern will be recruitment in the following style…

15.1.1 Specifics

The first will be a phone/Skype conversation. Here, the first technical tasks/questions may already come up, aimed at filtering out a wide group of candidates who are completely unsuitable for the given job, but usually at this stage these are often simple things that a non-technical person asks you about. Most of the issues you might be asked about, at least from my experience, will concern your person and your professional profile, what you did, what you want to do, where you're heading, why you're changing companies.

Contrary to appearances, such things are also important, because it's important what kind of candidate you are, what your character is and whether it will be good to work with you, really. Although it should be noted that your personality type matters more in small companies. What I mean is that in a large company there are many more different teams with different characters, where there is a much greater chance that you'll fit somewhere. In a small company it's harder - you have to fit better.

Does this mean you'll only be hired based on character? No. It's not THAT important. Remember one thing - don't be a jerk. Simple as that.

Does this mean you should pretend to be someone you're not? Definitely not. If you pretend during the interview, your true self will come out during work anyway. You'll suffer, the employer will suffer. What's the point?

Then there might be an in-person meeting or a take-home assignment in the form of some project or a Codility-type test.

In the in-person meeting, of course, you should expect more specific questions. Here most often a strictly technical person will join or lead the conversation. What kind of questions can you expect? Well, it depends.

Obviously, no one will ask you about things that aren't in your CV or about things completely unrelated to the position. That's rather clear. If it's different, it's a bit unprofessional from the interviewer's side.

As for those things that are in it, you can safely assume that someone might quiz you on every word placed in your CV. Besides that, there are also certain questions from general computer science, algorithms, or database basics that often come up, regardless of the position and company.

Naturally, it's worth taking into account the specific offer we're applying for and the company's profile. If it's a company largely dealing with database topics, it's simple that you can expect questions from that field.

A small note before we move on, some tasks are designed so that you won't be able to solve them, sometimes it will be necessary to ask for clarification, sometimes it won't be possible at all. Some recruiters use such a technique, or another one, introducing some stressful element during the task execution, to check how the candidate handles situations under pressure.

An example of something like this might be trying to convince you that a solution that is actually good is incorrect, or some interference during solving that is meant to distract you. Rather very rarely something like this is used, but I've heard about such practices, so be aware, although I doubt someone would use something like this towards a Junior.

If you're not sure about something, ask, don't be ashamed. Make sure you understand the task correctly, give some example and ask if that's what they mean. It's not like you get some negative points for every question you ask or something like that. It shows that you're not afraid to ask and communicate. For me that's a plus.

I'll give you the list of recruitment questions I encountered a bit later.

Then there might be more conversations or not. In any case, after some time, if the company has a normal approach to candidates, you should receive some feedback about whether you succeeded or not, if not, then what would be worth working on.

In short: a typical Polish recruitment can be divided into: - chat about you, your experiences - technical conversation - optionally homework assignment

And I'll reveal to you that I currently become rather picky - if a recruiter sends me an assignment that will take me several hours, then… Thanks for the recruitment. And I'm not saying this from a position of arrogance, I don't consider myself God knows who, absolutely not.

I just value my time, because those few hours I'll spend on solving will pass irretrievably. Besides, in my free time I prefer to program what I want, not some boring recruitment project.

However, you, at the beginning, when you're going to or looking for your first job or simply have little experience - you can't be picky.

The exception to this rule are assignments from IT giants. In such I haven't happened to work, that's first, secondly I haven't heard that they use such practices.

15.1.2 What Questions Will Appear?

How do recruiters determine questions? Is there some list that everyone chooses from? From my experience and from conversations with other people, it turns out that… no.

Such lists rather don't exist. Recruiters simply ask questions that… Someone once asked them. They have several of their favorite questions, sometimes they exchange them among themselves and that's it. Of course, there are also some standard example questions that appear everywhere, but you'll find such in the list of questions that I included in this book somewhere below.

There are rather no internal company secret lists that contain questions for potential employees. So here don't strain yourself unnecessarily looking for them. Simply be prepared and that's it. However, it's worth noting that if a company often deals with writing REST API then they'll probably ask about that. Generally, a good tip is to review the company's project profile and see what they deal with daily and prepare for that. If a company works in Python, they won't really ask about Java-related technologies.

What does it mean to be prepared? It's not just knowing some specific issues, no. Recruitment interviews are difficult, they're often stressful, especially if we care about the position. All this hugely affects how you'll work on the task given to you. That's one thing. Two, a common problem is also that someone knows a given issue but can't use it in practice.

Just reading, I don't know, Cormen, is not enough. These are not some scientific research or deliberations, but rather specific problems for which you'll have to find a solution. Besides, are you surprised? Most likely it's the case that your recruiter last had contact with such deep theory when they were finishing studies themselves and today they rather don't remember it, so naturally they won't ask about it, they'll ask instead about specific use, specific problem.

So once again I'll repeat, to be prepared, it's not enough to know or be familiar with some issue, you need to be able to really use it well in practice, because during the conversation, where you'll certainly be stressed, it will be harder for you to come up with new ideas, so it's really good advice that certain things should be just natural for you just like the multiplication table is. At least I hope it is.

15.1.3 Not Just Knowledge

Remember also that a programmer is not just some encyclopedia or a code typing machine. A programmer is also, and basically primarily, a human, and in people's lives communication and other soft skills are important. Very. Especially when you're starting as a Junior. Watch out for this during the conversation and try to be communicative. I repeat the appeal, don't be a jerk. In short NBJ.

You can be a really super programmer, but if it's hard to work with you, you're not like Peszko and don't create atmosphere, then the employer probably won't be happy, and especially your team.

This will translate into lower efficiency, worse atmosphere in the team, and a decrease in coworkers' satisfaction. It's better to hire someone who's super to work with, who has a good attitude, knows how to work in a team and has nice soft skills, because they'll learn the technical ones and quickly if they're somewhat capable. Simple.

It's much harder to teach someone those skills mentioned earlier, while technical ones are no problem. So know that your skills are not everything, your attitude to problems and solving them is also very important, fitting into the team, you also need to simply be human, avoid being an arrogant jerk and be nice and kind, but in a way that doesn't let others walk all over you. Just like that.

Don't think I'm exaggerating. I myself witnessed when a person, being the most competent in a given field in a specific team, was fired because team work was going rather poorly.

Communication is a must, especially in remote work, but... There's no need to overdo it in the other direction. Don't talk too much. Each additional sentence is a potential chance for you to weave some complete nonsense into it that might cross you off. I know because I speak from experience. You need to have some common sense and know when it's better to say a bit less.

Don't overdo it and be professional. At this stage let's say there's not always room for too much familiarity, so don't come out with such initiative, currently, as a candidate, you're in a position of, let's say, lower status, so give your interviewer due respect and behave professionally.

It's different if your recruiter proposes from the start that the atmosphere should be casual, that there's no need for formal addressing and so on. Usually that's exactly how it is, but it looks good if it's the older person who proposes switching to informal communication, rather than when you, the candidate, somehow force it. That's some advice.

While we're at it, I might mention a few words about attire. Well, I like to follow one rule, which is better to dress too elegantly than come in flip-flops and really embarrass yourself. Attire also speaks about you, similarly your overall appearance. Even if you don't like it, unfortunately, or fortunately, our appearance affects how people perceive us, regardless of whether they are aware of this fact or not.

Here again, common sense will help you again, because what you should wear often depends on the place itself where you're applying, it's known that you can come dressed somewhat differently to an interview at a startup or new software house, and differently to a huge corporation.

However, I usually dress as follows: black pants, simple shirt in subdued colors, if it's a more formal meeting, I add a jacket to that. Such a rather quite safe set that will be appropriate practically everywhere. So remember also that despite the fact that the meat and the most important thing are skills, knowledge and what you're like, presentation also counts. How they see you is how they write you. There are many studies where it was shown that we more willingly attribute positive traits to people who look nice, while to those who look sloppy, negative traits.

So look like something, preferably like a normal person, not a stereotypical programmer, a Mirek who was forcibly pulled out of the basement after two weeks of maniacally playing The Witcher 3. Straighten your back, pull your shoulder blades back, speak clearly and slowly. When you give a hand, give a hand like a human, not a wilted twig. Speak normally, not almost in a whisper so that you can't be heard. Come on! Be someone with dignity!

15.1.4 Result

What if you don't get in? Well, keep trying, improve your skills. There's no point in giving up. Note, sometimes you have to wait a bit for an answer. If the recruiter doesn't get back to you with information the next day, don't worry, it doesn't mean you didn't get in. Personally, once when I was recruiting, I had the interview somewhere at the beginning of June, and we only arranged the actual job in mid-August, so.

Sometimes you need to wait a week, two, and sometimes there won't be any information at all. I consider this a bit boorish, but unfortunately this is the practice and it's quite common unfortunately. Why is something like this disrespectful?

Well, if a candidate devoted their time, prepared, performed some tasks, came for an interview, they should at least get some information about the state of their knowledge, about what they did wrong.

It doesn't cost much work, just simply sending information, a simple email or two minutes on the phone, and, especially for someone who's just starting, it can help.

If someone doesn't even want to do something like this then... Well, it says something about the company and you should be happy that you don't have to work with such boors.

On the other hand - if it's you who shows up for an interview without any preparation, ignoring the interviewer, then congratulations, you ignored NBJ.

You got in, there's an offer? Great. Only...

What about that terrible question that always comes up somewhere along the way, which is how much would you like to earn? How to deal with this? About that in another chapter.

Same with example questions you might encounter.

15.1.5 Practice

Recruitment is also a skill. At the beginning it might go mediocrely for you. Despite the fact that in other conditions you'll solve the task, live during recruitment suddenly stress eats you up, or you stand still. It happens. That's why it's worth practicing and before recruitment ask someone (e.g. grski) to conduct a test conversation with you. Seriously.

Recruitment is also a test of how well you can sell yourself, without lying of course, and a test of your soft skills, negotiation skills. More than once I happened to earn more than my much more experienced colleagues just because I wasn't afraid to ask for raises or large amounts plus I performed quite tolerably in recruitments because I often participated in them even just for practice, conducted them myself, I was good at them. Thanks to this I could negotiate nice conditions surprisingly often.

15.1.6 Negotiations

People generally don't like to negotiate, and that's a shame. It's a shame because it simply pays off.

My record is a situation where the hiring bonus I got just by politely asking for it and jumping on a call for 5 minutes was 24,000 PLN. 5 minutes of additional conversation and 24k in the pocket. Damn, the hourly rate comes out nice here.

It's similar with salary. If half an hour of negotiation can bring you 5% higher salary, let's say even 300 PLN monthly, that gives 3600 PLN annually. Already quite a lot. And I remind you that it's only a few minutes usually.

15.2 How I Looked for My First Job

A bit about how these searches looked like for me.

15.2.1 First Class Spammer

Or first grade. Generally my approach, from the perspective of time not necessarily optimal, was as follows: go through all kinds of announcements. Anything related to programming and somehow at least with backend? Write to them. This was five years ago, year 2017, so there wasn't such an abundance of announcements as now. The bubble wasn't so big.

I wrote to various companies, sent emails, messages. Even when they weren't looking or when I didn't match in terms of candidate profile. I spammed en masse. This went on for weeks. I sent at least 80 applications to various companies. I got maybe 8 or 9 responses. 90% didn't even throw a simple "We'll call you back". I don't blame them.

15.2.2 Some Results

Nevertheless from those 8 or 9 responses I managed to get 3 emails where I was invited for an interview. In one case it was about a regular position(XD), in the second about a junior, and in the third about an internship.

15.2.3 First Interviews

I failed the first one at the algorithmic test stage, Codilime. I was missing 10%. Tough luck, moving on.

The second one, which was where the vacancy was for an internship, went like butter. I already saw myself in this job with my mind's eye.

Before the date of the arranged third interview came, I got a phone call. The CEO of one of the companies I spammed found my message in his SPAM folder, but read it anyway. Somehow it happened that I had something in my CV about knowing these whole pythons or that I used them sometime, and they just happened to have something open up, they need a workman.

So I said then that I would be whatever he wants me to be, give me that interview man. So he gave it.

That interview didn't go too great for me. I expected questions about Python, but suddenly surprise because the guy mainly questioned me about programming basics, data structures and such, which weren't my forte then. Fortunately I remembered some basics because I did a course from harvard - CS50, so I managed to piece together a few sentences. However in my head there was still chill utopia, because in my reception, that second interview went perfectly for me.

15.2.4 Surprise One

Well, it turned out that that internship interview didn't necessarily go as great as I thought. They didn't call back. Here I got a bit sad that nothing came of it and I'll have to spam again, apply that is.

15.2.5 Surprise Two

Well someone else called back. The CEO from the message in SPAM. He called with information that I made a cool impression, there's still a lot to improve of course, but I'm quite a nice guy so we can work something out.

They offered me remote work, which for me, still being in high school, was ideal. Money, for those times, also okay because I think something like 20 PLN per hour, where on construction you earned 11 PLN, and I gave tutoring for 30 PLN. So yeah.

Taking all this into account I didn't even think for a moment, immediately agreed and by the way informed that last company with which I had an arranged interview that I'll pass.

15.2.6 Trial Period

Generally the agreement was that first two months I have a trial period, if there's progress there will be extension. So what, I rolled up my sleeves and pushed with the topic.

After two months I got a raise and extension. It worked out.

I was jumping with joy overall. Here again, besides those thanks at the beginning, I want to thank Jakub Gąsiorski and Maciej Mondrzycki. These are the two people who largely are responsible for where I am and who I am. They allowed me to catch first experience, invested and took some risk, hiring an unknown guy, a kid from high school who barely turned 18, to a serious company. Kudos.

15.2.7 Summary

That's how my search for first job looked like. It wasn't ideal, now I would do it somewhat differently, primarily focusing on a smaller number of companies, writing more personalized messages, I would put more effort into portfolio projects on github, into making a nice CV and better prepare before recruitment interviews. I recommend this to you. Choose several offers or companies that you like. Write proper messages/cover letter even and send them out. Look for offers and companies on justjoinit, probably the best job board.

And speaking of CV...

15.3 CV

A readable, nicely polished CV is half the success. Let's talk about the art of creating a readable CV. I'm not an expert in this field, but it seems to me that I've seen enough CVs to know what is more or less okay and what isn't.

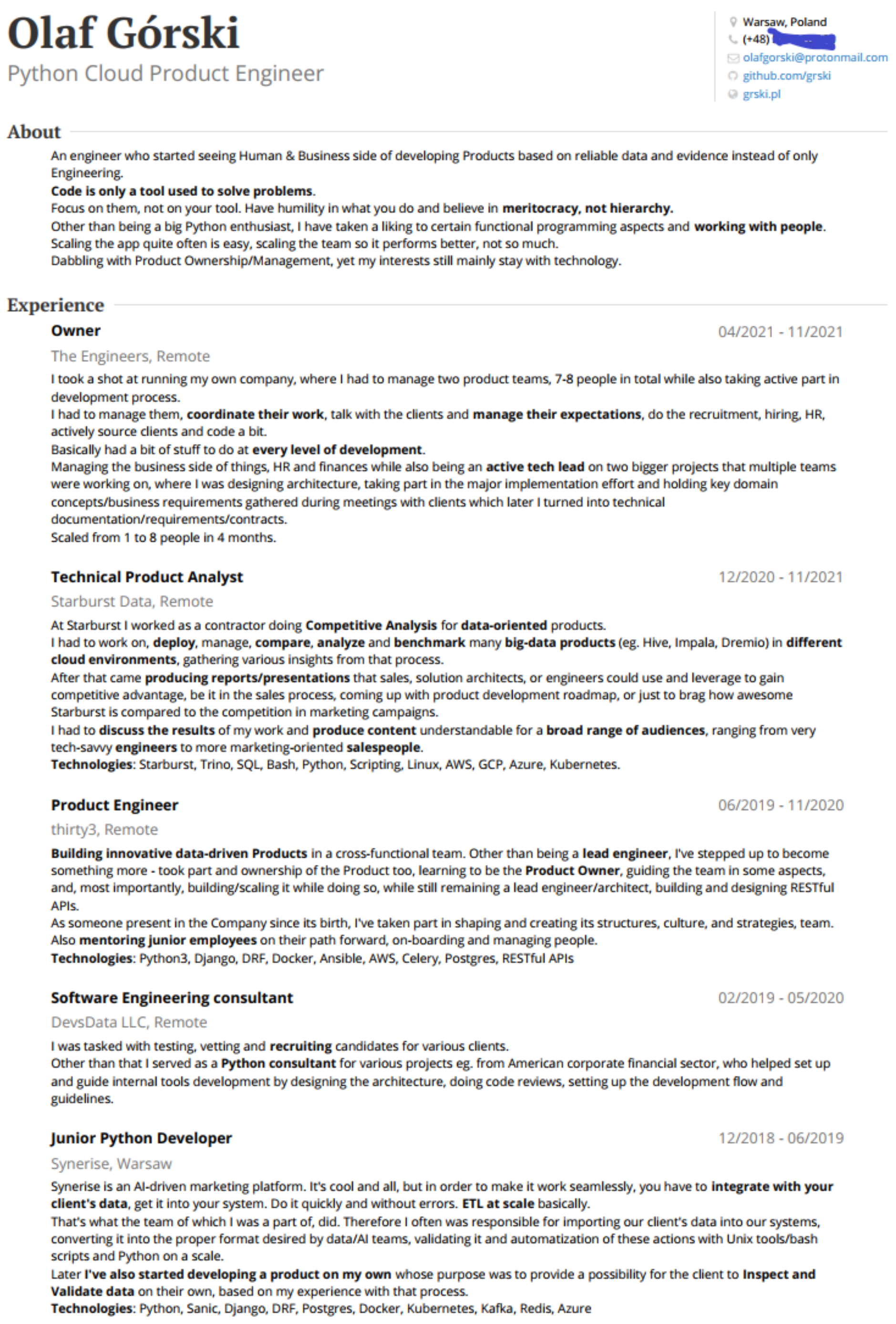
15.3.1 My Latest CV

To not analyze dry theory, I'll show you an example CV, in this case mine, which I consider quite tolerable.

15.3.2 Analysis

What does the CV above consist of?

1. Contact Information Things like phone number, email address or city of residence. There's no need to provide exact address, but sometimes it's worth it for the recruiter to know more or less where we work from.
2. GitHub Probably obvious. A nice portfolio on github helps a lot.
3. Blog Here of course not everyone has one, but I actually do. I think I have several nice articles, hence I linked it. This also gives some sense that I show initiative and do something 'from myself'. When I recruited I always gave some small plus for something like this. Additionally having a blog allows the recruiter to read a piece of your text, get to know you in some way. I also had interviews where recruiters openly admitted that they weren't necessarily planning to invite me for an interview, but after looking at the blog they changed their mind. Just remember, no content is better than shit content put there just to be there. While you sit quietly people don't know you're stupid - this saying applies here. So think twice about what you put on your blog and whether you need it. Nothing by force.



Olaf Górski

Python Cloud Product Engineer

Warsaw, Poland
(+48) [redacted]
olafgorski@protonmail.com
github.com/grski
grski.pl

About

An engineer who started seeing Human & Business side of developing Products based on reliable data and evidence instead of only Engineering.
Code is only a tool used to solve problems.
Focus on them, not on your tool. Have humility in what you do and believe in **meritocracy, not hierarchy.**
Other than being a big Python enthusiast, I have taken a liking to certain functional programming aspects and **working with people**.
Scaling the app quite often is easy, scaling the team so it performs better, not so much.
Dabbling with Product Ownership/Management, yet my interests still mainly stay with technology.

Experience

Owner 04/2021 - 11/2021

The Engineers, Remote

I took a shot at running my own company, where I had to manage two product teams, 7-8 people in total while also taking active part in development process.
I had to manage them, **coordinate their work**, talk with the clients and **manage their expectations**, do the recruitment, hiring, HR, actively source clients and code a bit.
Basically had a bit of stuff to do at **every level of development**.
Managing the business side of things, HR and finances while also being an **active tech lead** on two bigger projects that multiple teams were working on, where I was designing architecture, taking part in the major implementation effort and holding key domain concepts/business requirements gathered during meetings with clients which later I turned into technical documentation/requirements/contracts.
Scaled from 1 to 8 people in 4 months.

Technical Product Analyst 12/2020 - 11/2021

Starburst Data, Remote

At Starburst I worked as a contractor doing **Competitive Analysis** for **data-oriented** products.
I had to work on, **deploy**, manage, **compare**, **analyze** and **benchmark** many **big-data products** (eg. Hive, Impala, Dremio) in **different cloud environments**, gathering various insights from that process.
After that came **producing reports/presentations** that sales, solution architects, or engineers could use and leverage to gain competitive advantage, be it in the sales process, coming up with product development roadmap, or just to brag how awesome Starburst is compared to the competition in marketing campaigns.
I had to **discuss the results** of my work and **produce content** understandable for a **broad range of audiences**, ranging from very tech-savvy **engineers** to more marketing-oriented **salespeople**.
Technologies: Starburst, Trino, SQL, Bash, Python, Scripting, Linux, AWS, GCP, Azure, Kubernetes.

Product Engineer 06/2019 - 11/2020

thirty3, Remote

Building innovative data-driven Products in a cross-functional team. Other than being a **lead engineer**, I've stepped up to become something more - took part and ownership of the Product too, learning to be the **Product Owner**, guiding the team in some aspects, and, most importantly, building/scaling it while doing so, while still remaining a lead engineer/architect, building and designing RESTful APIs.
As someone present in the Company since its birth, I've taken part in shaping and creating its structures, culture, and strategies, team.
Also **mentoring junior employees** on their path forward, on-boarding and managing people.
Technologies: Python3, Django, DRF, Docker, Ansible, AWS, Celery, Postgres, RESTful APIs

Software Engineering consultant 02/2019 - 05/2020

DevsData LLC, Remote

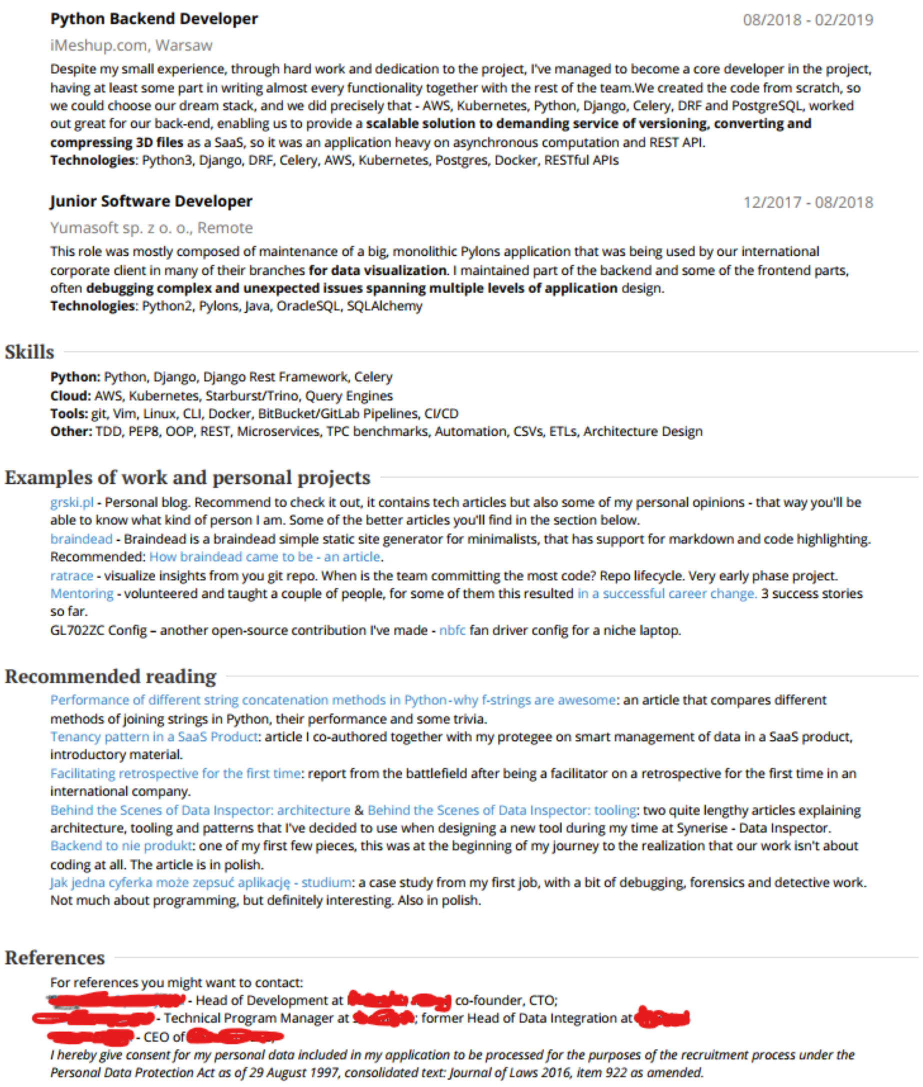
I was tasked with testing, vetting and **recruiting** candidates for various clients.
Other than that I served as a **Python consultant** for various projects eg. from American corporate financial sector, who helped set up and guide internal tools development by designing the architecture, doing code reviews, setting up the development flow and guidelines.

Junior Python Developer 12/2018 - 06/2019

Synerise, Warsaw

Synerise is an AI-driven marketing platform. It's cool and all, but in order to make it work seamlessly, you have to **integrate with your client's data**, get it into your system. Do it quickly and without errors. **ETL at scale** basically.
That's what the team of which I was a part of, did. Therefore I often was responsible for importing our client's data into our systems, converting it into the proper format desired by data/AI teams, validating it and automatization of these actions with Unix tools/bash scripts and Python on a scale.
Later **I've also started developing a product on my own** whose purpose was to provide a possibility for the client to **Inspect and Validate data** on their own, based on my experience with that process.
Technologies: Python, Sanic, Django, DRF, Postgres, Docker, Kubernetes, Kafka, Redis, Azure

Figure 2: First page of cv



Python Backend Developer 08/2018 - 02/2019

iMeshup.com, Warsaw

Despite my small experience, through hard work and dedication to the project, I've managed to become a core developer in the project, having at least some part in writing almost every functionality together with the rest of the team.We created the code from scratch, so we could choose our dream stack, and we did precisely that - AWS, Kubernetes, Python, Django, Celery, DRF and PostgreSQL, worked out great for our back-end, enabling us to provide a **scalable solution to demanding service of versioning, converting and compressing 3D files** as a SaaS, so it was an application heavy on asynchronous computation and REST API.
Technologies: Python3, Django, DRF, Celery, AWS, Kubernetes, Postgres, Docker, RESTful APIs

Junior Software Developer 12/2017 - 08/2018

Yumasoft sp. z o. o., Remote

This role was mostly composed of maintenance of a big, monolithic Pylons application that was being used by our international corporate client in many of their branches **for data visualization**. I maintained part of the backend and some of the frontend parts, often **debugging complex and unexpected issues spanning multiple levels of application** design.
Technologies: Python2, Pylons, Java, OracleSQL, SQLAlchemy

Skills

Python: Python, Django, Django Rest Framework, Celery
Cloud: AWS, Kubernetes, Starburst/Trino, Query Engines
Tools: git, Vim, Linux, CLI, Docker, BitBucket/GitLab Pipelines, CI/CD
Other: TDD, PEP8, OOP, REST, Microservices, TPC benchmarks, Automation, CSVs, ETLs, Architecture Design

Examples of work and personal projects

grski.pl - Personal blog. Recommend to check it out, it contains tech articles but also some of my personal opinions - that way you'll be able to know what kind of person I am. Some of the better articles you'll find in the section below.
braindead - Braindead is a braindead simple static site generator for minimalists, that has support for markdown and code highlighting. Recommended: How braindead came to be - an article.
ratrace - visualize insights from you git repo. When is the team committing the most code? Repo lifecycle. Very early phase project.
Mentoring - volunteered and taught a couple of people, for some of them this resulted in a successful career change. 3 success stories so far.
GL702ZC Config – another open-source contribution I've made - nbfc fan driver config for a niche laptop.

Recommended reading

Performance of different string concatenation methods in Python-why f-strings are awesome: an article that compares different methods of joining strings in Python, their performance and some trivia.
Tenancy pattern in a SaaS Product: article I co-authored together with my protegee on smart management of data in a SaaS product, introductory material.
Facilitating retrospective for the first time: report from the battlefield after being a facilitator on a retrospective for the first time in an international company.
Behind the Scenes of Data Inspector: architecture & Behind the Scenes of Data Inspector: tooling: two quite lengthy articles explaining architecture, tooling and patterns that I've decided to use when designing a new tool during my time at Synerise - Data Inspector.
Backend to nie produkt: one of my first few pieces, this was at the beginning of my journey to the realization that our work isn't about coding at all. The article is in polish.
Jak jedna cyferka może zepsuć aplikację - studium: a case study from my first job, with a bit of debugging, forensics and detective work. Not much about programming, but definitely interesting. Also in polish.

References

For references you might want to contact:
[redacted] - Head of Development at [redacted] co-founder, CTO;
[redacted] - Technical Program Manager at [redacted]; former Head of Data Integration at [redacted]
[redacted] - CEO of [redacted]

I hereby give consent for my personal data included in my application to be processed for the purposes of the recruitment process under the Personal Data Protection Act as of 29 August 1997, consolidated text: Journal of Laws 2016, item 922 as amended.

Figure 3: Second page of cv

4. Brief Summary Here in short I presented myself, my values and previous experiences. Short and concise.
5. Experience List of subsequent jobs I took. Here can also be internships, participation in various projects etc. Newest at the top, oldest at the bottom. Sorting order. Describe example technologies, what were the challenges, what actions were taken, what was the result and what distinguished this work from others. Such advice.
6. Skills That is keywords for recruiters. Recruiters are often non-technical. From technical jargon they might not understand, and the only thing they do is scan CVs looking for specific keywords. This is where to place them. However it's important not to throw everything in here as it comes. If you wrote hello world in Java don't add it to your skills. Assume that you'll be questioned about every skill written here. If they catch you lying it's Game Over.
7. Project Examples Also probably clear.
8. Article Recommendations If you don't have a blog then skip this. I added such a section because I'm proud of some texts and wanted to expose them.
9. References Also optional.
10. Hobbies(?) In my case actually lack of this, because I had little space due to large number of entries in Experience, however, as often happens with Juniors, when the CV is somewhat empty, throw in something about yourself about your hobbies, interests, generally what kind of person you are. For junior/intern you're hired more as a person and not specific skills anyway, because one needs to invest, so let yourself be known from a good side.
11. Data Processing Agreement At the very bottom in tiny text is the clause that unfortunately needs to be added to every CV, because otherwise the company can't process such a document.

15.3.3 Photo

For the love of god, don't. This isn't recruitment for president's assistant to throw in photos. No. There's no need. This is some weird Polish invention with these photos in CV. Abroad they don't know this.

15.3.4 Summary

My CV looks like it looks. I described its elements. This isn't some absolute list, but I think quite okay guidelines. CV necessarily in English. This is IT, here we communicate in English. Plus clean and neat format. As for font there is one rule, anything but Comic Sans. I think that's it.

15.4 Studies and Recruitment

When I was looking for my first job I was a freshly minted eighteen-year-old, being in the last year of high school. Zero education, experience, nothing. And yet it worked out. As you can see it's possible. That's what I note at the beginning, if you doubt yourself, to give you some encouragement. If grski managed then you can too as long as you put some work into it and have some luck. Before we move on to details two words about studies/education.

15.4.1 Is the So-called Paper Worth Nothing?

Generally one might get the impression that I have a negative attitude and talk about studies in a pejorative way. Nothing could be further from the truth. It seems to me that studies can be a wonderful place where you'll learn a lot and meet wonderful people. However, if you're going there just for the paper because mom told you to, you have no idea what you're doing there, you're not learning anything, well then what's the point.

Then yes, then this whole undertaking makes no sense. Better go work any job because you'll get zero benefit from these studies just suffering only.

If however you sensibly approach the topic of your studies then they will be only an asset and not something that might harm you. Then it's a nice opportunity.

15.4.2 Do I Regret Not Going to Studies?

Life somewhat chose for me, I was kind of forced to take up work quickly, that's a fact. So one might ask oneself, do I regret it? No. Having 3 years of experience, when my peers somewhere there were just starting to crawl, were interns, I was already managing a team, met wonderful people and developed connections in the industry.

I wouldn't give that up for those additional few years of carefree life.

However, I note that this is just my case. Not everyone will have so much luck. Studies are therefore something that's really worth considering, however it's not a necessary element.

Studies are a nice time to develop connections, because it's easier to talk with Manek, with whom you went to the stairs for beer, who is currently somewhere a Team Leader, and differently with freshly met Tadeusz who is a stranger. There's no one recipe.

15.4.3 Does Lack of Studies Cross Us Out?

A frequently asked question. Does lack of studies cross out a candidate? Well my dears, if someone judges you only through the prism of your education and not You, as a human, then the question is whether you want to work with such a company? For me that's a red flag right away.

In my entire career and currently one might say dozens of recruitments, only once did it happen that the CEO barged into recruitment and the question fell, I quote:

> Why should we hire you since you don't have studies?

After the technical conversation he already knew why and ultimately they offered me the job. I politely refused. I don't work with boors, however class must be maintained and respect for another human. As if that wasn't enough a year later they wrote again whether maybe something changed :-)

This fate is perverse, sometimes wants to play poker with me.

So no, lack of studies doesn't determine anything.

15.5 What I Knew Going to First Job - Case Study

Today about what specifically you need to know to get your first job, or rather about what I knew when I was looking for my first job. Catchy title, because in different companies there are different requirements for juniors and in one such a guy like me is an intern, and in another already a junior, it varies. Nevertheless this is just my perspective, not some absolute determinant, but these things below allowed me to become a junior python dev, so yeah.

Besides that I'll also describe some of my projects that I included in my CV.

Is what I'll write about here necessary or actually sufficient? I don't know. For me in interviews it was enough, but I didn't have that many of them, so my experience is limited, but let's say that for such an internship you'll manage to get in knowing what's below rather without problem.

The order I chose is rather random, if something is in my opinion important, I clearly mark it.

15.5.1 Googling

This is the first and probably most important skill. After all, broadly understood computer science is not a science about programming, but about... information processing. Part of this process is searching, analyzing and using given information needed to solve some problem, or implement something new. In short - googling. What do I specifically mean by this?

The ability to concisely rewrite errors, search for information on appropriate sites. Best in English - in this language we'll find of course the most materials.

This is also primarily about not throwing up your hands and stopping in case of standstill, problem, or error, but to do something, look for help, information. That's what it's about. 99% of problems are already solved somewhere, you just need to search. If you hit that remaining 1%, you're rather at such an advanced level that we have nothing to talk about.

15.5.2 English

This is a necessity. We talked about this in one of the first chapters. The multitude of materials available in English, the fact that it's an international language, popularized and ubiquitous, clearly speaks to the fact that it's necessary for every programmer. Soon probably there will be people who will state that they don't know any foreign languages and they work. Cool, but where? What kind of company is that? Rather I don't know respectable companies employing people without at least basic English.

How without it to cooperate with foreign clients, coworkers who often also aren't from Poland and don't speak Polish? How to read documentation, code?

Plus abroad usually pays better.

> Say what you want, but at the end of the day money must add up. 2nd Letter of Prophet Stonogiasz to Poles

What level is needed? Minimally such to be able to read technical documentation of some language/framework. For start that's enough.

15.5.3 Algorithms and Data Structures

Yes, contrary to appearances you need to know these too. Supposedly it's 'boring theory' and so on, but... It's a necessity. Not to implement them from scratch yourself, but for two purposes. First is passing recruitment, because you'll often be questioned about them, second is awareness of how certain things work under the hood. That is what I always mention, until nausea even.

15.5.3.1 Basic Algorithms

1. Insertion Sort - I probably don't need to explain, simple searching through data structure. Basics of basics.
2. Selection Sort - similar as above. Very basic, inefficient.
3. Bubble sort
4. Merge sort b) searching
5. Linear Search - complete basics
6. Binary Search

And now note - do I know each of these algorithms exactly by heart, implemented several times and used my own, modified versions? No. I simply know how they work - I know their work scheme, their advantages and disadvantages, when to use which. When not to use given algorithm, bad cases and so on. Because that's what's important. Sometimes it dramatically changes the speed of our program's work.

While we're at algorithms, it's worth mentioning something like **recursion**, which in turn is something you need to know to learn Merge sort or Binary search.

The next important thing is **computational complexity**, or so-called **big O notation**. What's it about? So in layman's terms it's the amount of some 'moves' that our program will need to execute given instruction for an n-element data set. We distinguish different cases for different situations - in best case, in worst and in average. Usually the worst is given, because it allows us to state that 'well, it won't get worse'.

So speaking even more simply, it's the number of steps needed to do something and how it grows when there's more of that something to do. There are different complexities - log n, n, n*n, ln n, 1 and so on. About details I won't write, waste of time. Generally the smaller this complexity, the faster the program.

Besides notation about `steps` there's also notation regarding e.g. memory/disk space. Analogically.

15.5.3.2 Data Structures Here the matter is simple. Data structures are things that can often significantly make our life easier, they are simply some data arranged in a specific way. It's like some specific way of thinking about data. Here it's worth to dig a bit into details.

What specific data structures did I know? I'll tell you.

1. Array - probably everyone knows, right?

2. Dictionary - also?
3. LinkedList - majority?
4. Stack
5. Heap
6. Set
7. Queue
8. Trees
9. HashTable
10. Trie

With those last three being rather extra, above level I would say. What's worth knowing here? What are advantages/disadvantages of given structure, when to use which. Computational or memory complexities for them and basic operations on them like searching, insertion, sorting. Similar as with algorithms.

Again I emphasize that I practically didn't implement any of these data structures myself - I only know approximately how most of them work and look. The only thing I implemented myself was Linked List.

Anyway. This seriously comes in handy. Believe it.

15.5.4 Tools

You need to know how to operate tools that are needed for work, period. What was this in my case?

IDE - PyCharm and Vim. Currently I use practically only PyCharm and now also sublime. Earlier it was vim with plugins, but… with larger project for me PyCharm is simply more convenient.

Python - what is virtualenv, pip and these things? What for, for what? And pipenv? Pypi?

Virtualbox/WSL - again, worth to learn a bit what it's about, because useful tools, especially when we work on several larger projects.

15.5.5 Glossary

And now I'll throw in also some various general terms from broadly understood industry. Do you need to know each of them and sometime use given tool and so on? Well no, but it's nice to at least recognize, it gives such orientation and when we talk with other people, maybe more experienced, then certain terms will be familiar to us, at least part. Then, instead of some incomprehensible jargon you hear already minimally familiar babble :D

1. VPS - in short virtual private server, like your own piece of some server you could say
2. VPN - virtual private network, like a private network you could say
3. Docker, Vagrant - that first one I only know the name, and that it's something with servers and app deployment, and that second one read about it, because nice tool
4. HTTP, HTTPS, SSL, SSH, IMAP, POP3 protocols (!) - just superficially which goes to what and how they differ
5. IDE - that is integrated development environment

6. MVC - model, view, controller - data processing model used in considerable amount of modern frameworks
7. API - application programming interface (!)
8. Cookie, Session, Cache - cookies, session, cache storage
9. MySql, Postgres, mongodb, oracle, sqlite, sql, nosql, sql injection - (!) databases, necessity in majority of applications
10. Http codes: 3xx, 4xx, 5xx, but in practice actually enough: 300, 500, 200, 404
11. AJAX, JS, ecmascript, typescript, react, vue - javascript, these things, you know
12. Class, Method, Object-orientation, object-oriented programming, functional, OOP (!!!) Generally object-orientation important matter
13. CMS - content management system
14. DDOS, DOS - such attacks by bad hackers
15. VCS - version control system, e.g. GIT
16. Devops - what we do here
17. DOM - document object model - for JS'ers necessity
18. DRY, KISS, YAGNI (!!) - such 3 principles worth following during code creation
19. linux, bash - linux is operating system, bash is scripting language in simplification
20. Framework, library - differences between them, is it the same thing?
21. Unit test, functional test, (!!) - unit tests and functional tests, about them below
22. Hash, crypto - what is this hash, cryptography - what does it deal with at all
23. Machine code, binary, hex - machine code, binary, hex. Such various number systems useful because computer uses them, not decimal
24. Production, dev, qa, localhost - production, that is server where real client's site/app stands, dev - development environment, local, localhost, qa/testing - such for testing you could say, before it goes to production it often wanders there, although as they say best tests are only on production. Seriously no, don't try this at home
25. Segfault - segmentation fault, that is what happens when you don't handle a bit in low-level language, or rarely, in high-level
26. Bluescreen - probably you know :-)
27. Memory leak - google it
28. Integer overflow - also
29. Server - there we throw in and run various apps/sites (in huge simplification) and they live there
30. Spaghetti code - simply code that is tragic
31. TDD - test driven development, DDD - domain driven development, just trivia
32. Firewall - well probably don't need to explain
33. Compiler, linker, interpreter, assembly
34. Design pattern, the big4/gang of 4
35. PNG, JPG, GIF, image sprite - file formats in big simplification
36. IP, TCP, PORT, Apache, Nginx, gunicorn
37. Bootstrap, jQuery, Material Design, gulp
38. Asynchronicity

A lot? Actually no, because majority should have rung a bell sometime. Again - it's about you simply recognizing these terms and knowing what's five.

Most of them are explained here: https://www.hackterms.com/ and if not there?

Google. Or DuckDuckGo, whoever prefers what.

15.5.6 Programming Languages

Okay, now time to talk about languages themselves.

In my case it's Python. What specifically did I know?

First let's start with knowledge of Python's standard library. I was at least superficially familiar with it. What does this mean? Well, that when I wanted to sort something, I didn't write function myself, just bam sort(), or sorted(), depending on case and done. This is simply about such general orientation - here you have actually a list: https://docs.python.org/3/library/index.html Everything? Absolutely not, simply need to recognize something there, no one expects you know it by heart. Browse documentation, official documentation of Python or Django is super.

What to focus on? Basic data types, structures in Python, most commonly used functions, how OOP looks in Python, what are these generators, iterators, list/dict comprehensions, f-strings, reading, writing to files, opening some link using python, what is GIL and multithreading in python, datetime, collections.

How? Simply by using in practice - make several diverse projects, read other people's code, see how someone solves given problem, compare with your idea.

Besides that I had contact with such libraries in python as BeautifulSoup, Selenium.

Additionally there's still C - but only basics, which I grasped at CS50. That only as part of learning about how certain things look level lower.

Besides that I know html5/css3 at level allowing me to independently create simple templates/modify those more complex.

JavaScript - here actually I google everything, but nice if sometime for test you make yourself, I don't know, script changing div content on site, or something, to see how it works. Remember also that often just JS is enough, don't need gazillion libraries.

With jQuery same as JavaScript - not much I grasp, everything I google as needed. Main thing is to know how to include jquery itself to html, rest google. In 2022 rather I wouldn't bother with it however. JS's API has grown and developed compared to 2017, when I was starting.

15.5.7 Frameworks

This bit about which ones I had contact with.

Django, flask, long long ago web2py, wordpress. Am I master in each of them? Absolutely - in none, but when I take documentation, then something I'll already create independently there, at least in those first two.

In your place I would aim for two - Django and FastAPI. But that's personal note, I'm not fan of Flask.

15.5.8 Design Patterns

Actually probably you use them but don't even know about it and that's normal. It's about you recognizing that something like this exists. My knowledge more or less was limited to this then. Maybe few I would name by name, you can additionally read. Singleton. Factory. Such things.

15.5.9 Testing

Well there is something like this. How does it look? Is it about clicking through app? Well not always. Nice to know what are these unit tests, functional tests, integration tests and such. Do you need to be proficient in them and know how to write them? No, actually majority of programmers not really grasp them so actually, but always plus for this and quite big one.

15.5.10 Operating Systems

If you heard that there is something like Linux, then cool. If not, well then also not worst. You surely already heard because I wrote about it earlier. Honestly it's heavily dependent on your work whether it will be needed for you, but in my opinion it's good to know at least some basics at start. I'm not telling you to install Gentoo right away or something, but if you'll be able to move around system using console, create, move or change file name, delete it, using linux (or windows) console, that will be great. So we are safe after reading this book.

If it's about me, then I simply used Linux daily, but it's not necessary. From what specifically? Manjaro (mainly), Solus, Ubuntu. For that second I even created package with pycharm-community edition because in repo there wasn't due to it being new project. Do you also need to? Well no. As I wrote above, enough to know how to move around, because who knows pitchfork might come to you to do something using SSH, even on VPS and what? Bummer. And like this bam and you can.

Although now for example I don't use linux at all, because I work on mac, previously it was windows with WSL - that's how it happened at work. I'm just saying, to once again mark, that knowledge of linux isn't necessary, but veeeery useful.

But let's say you're ambitious. What then? Commands: **cd, ls, touch, mv, cp, ps, kill, rm, man, echo, grep*, cat, chown, cat, sed** Additionally how to install packages in given distro, how to update.

That's all.

15.5.11 Speaking of Systems

Let's talk about things bit lower level. Yes, these horrible, archaic things. Yup. Well imagine that my understanding of computer science, programming and general concept, grew essentially after I grasped a bit these exactly basics.

How computer represents everything in memory, how this memory looks, stack and heap, differences between them, when we use which. How data structures look, why we index arrays from 0, not 1, how this linked list looks from inside, counting in binary system, hex.

What is this assembler, how compilation process looks, what difference is between compiled and interpreted languages, static and dynamic typing.

Sounds boring, but these basics will allow you to become simply better programmers. You can not know this and also find job, but question. Do we want to be broadly understood specialists or maybe simply code typers from templates?

This knowledge, about how it works from low-level side, came in handy for me in recruitment to current job actually.

Worth also to know how to do certain things in console - besides IDE. You better then get to know mechanisms of operation of some tools.

15.5.12 Debugging

When something goes wrong, then need to fix it.

Make friends at least minimally with debugger, get familiar with debugging methods.

Because besides debugger you can still do bit differently - rubber duck methods, or also 'XD' method. Nice thing. What do they consist in?

That first one in that you explain to someone, something, by default rubber duck, how your program is supposed to work step by step. At some point it might click to you that AHA! After all I wrote it differently or you compare it also with code and look whether actually it is so.

And 'XD' method? It's nothing else than usual method of printing out - you add to code lines printing out some data, some characteristic message, to easily find in logs and then you look where it breaks. For me usually this characteristic message is 'XD' because I'm supporter of this philosophy, but not important.

Last option is going through your code step by step and getting into role of computer - you execute instructions exactly as written, not thinking too much and e.g. on piece of paper substituting values.

So yeah. We grasp debugger. So called debugger. No, no one uses that second term.

15.5.13 Good Practices

This is mainly about code organization, naming variables and so on. Variables should be named concisely, but accurately, descriptively, specifically. What does variable x do? Well no one knows that, you'll also forget, but if variable is called number_of_clients then already something this tells us.

When it comes to names we talk about, then we have different methodologies of naming variables, methods, classes. Camel case, underscore and so on. Which? No matter, as long as you stick to one consistently. Read PEP8.

Functions that you write shouldn't be too long either - don't stuff everything into one bag. Divide problems into smaller elements, break them down and program, create in way allowing for their multiple use without repeating code.

Think about cases when user won't necessarily behave as you expect. In general before you take up programming then think. Actually programming is mainly thinking and only then writing. Analyze problem, consider different cases and only then take up implementation. It will go faster for you and you'll create better code than if without thought you simply took up writing.

Create code rather simple and flat, nicely divided. Complicated inheritance and millions of abstractions won't be your friend.

Care for code readability, file names, consistency.

15.5.14 VCS - about git

What VCS is and what for you need it I won't dig into, but one thing is obvious.

You need to know Git. And if you don't know? You can get somewhere there, but rather to such company from which you need to run. It's not difficult, learn it. Git commit, merge, branch, checkout, push, pull, rebase. Probably more for start not needed, for me this suffices.

16 Example Questions and Tasks

The questions you'll encounter depend strictly on what you include in your CV. Here I'll provide a list of questions I've encountered during recruitment interviews. I'll also add some of my own. I'll discuss all of them, at least briefly.

16.1 Django

1. Distinct in Django model
2. Django middleware - what is it, where is it used? Answer: It's a piece of code that processes our request on input and output in some way. At this level we handle things like authentication.
3. Django request path Answer: Middleware -> router -> views -> models/code/whatever/ -> middleware again (optionally) -> client
4. Django select_related
5. How do migrations work in Django
6. Does makemigrations need a database connection?
   Answer: No, it doesn't. Django compares models with the last saved state (using hashes, I think) and if it detects changes, it gets to work.
7. Does migrate need a database connection?
   Answer: Yes - it makes changes to the database, so absolutely.
8. Is it possible to implement login/authentication without sessions?
   Answer: Absolutely. JWT or Cookies for example.
9. Difference between Flask and Django
   Answer: Google it, or better yet try it yourself and compare, because it's basic stuff.
10. Difference between Django and FastAPI

Answer: Same as above.

1. Setting unique_together on a model attribute that has `null=True`. Does Django allow this? What's the risk? Answer: In the database null == null is False, so (null, 1) and (null, 1) will be unique for the database. Same for Python.

16.2 Python

1. 3 favorite Python 3 features compared to Python 2 Answer: Here the answers are obviously very individual, so I won't provide a template, but I, if I remember correctly, mentioned: f-strings, unicode as default encoding, assignment operator in expressions (walrus).
2. What would you change in Python - provide an example of a new PEP Answer: Very individual matter. I mentioned something about removing GIL. And speaking of GIL, time for the classic question in every Python dev recruitment.

3. What is GIL? Answer: Global Interpreter Lock, and what it is, I've already described, so you should know. In short, it causes a kind of lock that makes only one thread execute at a time in a given interpreter instance.

4. List types of comprehensions in Python. Answer: Like List Comprehensions, Set Comprehensions, Dict Comprehensions

5. Provide examples of the above, one for each type. Answer: Examples could be...

```
list_comprehension = [x for x in range(10)]
dict_comprehension = {x: x**2 for x in range(10)}
set_comprehension = {x for x in range(10)}
set_comprehension_variation = set(x for x in range 10)
tuple_comprehension = tuple(x for x in range(10) if x % 2)
```

6. What are the differences between a tuple and a list? Answer: The main difference is that one can be mutated, the other cannot - a list can be changed after creation, a tuple cannot - after initialization you can't change it. Besides that, there's a big difference in performance and implementation 'under the hood'. Because a tuple is immutable, the interpreter at initialization time knows exactly how much space it will take, so it allocates only as much memory as it needs, not a gram more. A list, on the other hand, during initialization, is created in such a way that it always allocates more memory than needed for the actual data in the user's list, so that later operations of adding something to the list work faster. If you create a list of, say, 50 elements, it turns out that your list has allocated space not for 50 elements, but for, say, 90. You can find out the exact numbers yourself.

7. What are generators and how do they differ from lists? Answer: Generators are something that makes it easier for us to work with large amounts of data, because they are a kind of 'functions' that instead of loading some entire data set into memory, process values one by one and yield subsequent values. They differ from a list in that a list loads everything into memory, a generator doesn't, greatly simplifying things.

8. Can you program functionally in Python? Answer: Yes. It's possible - Python is a language that enables programming in many paradigms such as functional, object-oriented, imperative, procedural. One could argue whether this programming would be 'pure' (probably not) functional and so on, but it's possible in itself. Not a very popular option and not very promoted. Generally some people look askance at this combination.

9. Encapsulation in Python - is it possible? Answer: Yes and no. On one hand, Python lacks access modifiers like private or public, but we have conventional designations that cause, for example, variables or functions starting with underscore '_' to be rather private and shouldn't be used outside a given class. However, this is only a convention, not a language element, nothing will prevent you from using a method with an underscore everywhere.

10. How to solve the GIL problem?
    Answer: Stackless Python or other implementations or using processes instead of threads in our concurrency model.
11. What's under the hood of a dict, what theoretical data structure? Answer: Hash map.
12. How does access to dict elements proceed? Answer: Based on the key, a hash is calculated using a specific algorithm. From this hash, an offset pointer to the reference is obtained.
13. Behavior of order of items added to dictionary in Python, how? Answer: In Python < 3.7: OrderedDict() from collections. In >= 3.7 dict() by default preserves order.
14. Differences between list and set. Answer: In a list elements can repeat. In a set they cannot. On sets we can perform certain operations that we can't on lists - similar to normal sets in mathematics - sum, intersection, difference. In the simplest case, you can think of a set as a list without repetitions. It's also worth remembering that data in a set is unordered by definition, so a set is completely unsuitable for ordering elements.
15. What type of objects do the functions dict.keys(), dict.values(), dict.items() return? Answer: Dict.items() - obvious. List of tuples. But not quite. Because if you look closer, it's a dict_items class, which isn't quite a list - it's a bit of an extended class, because it enables us to perform operations on it like on sets. Similarly with keys() and values(). So on objects returned by these functions you can perform operations like sum, difference or intersection from sets. Tldr - these functions return an iterable set-like object.
16. Bitwise operators in Python. Which do you know, what are they for? Answer: « and », bitwise shift left and right & - bitwise conjunction, i.e. 'and' | - bitwise alternative, i.e. or ~ - bitwise negation, i.e. ~x = -x − 1 ˆ - bitwise exclusive alternative
17. When to use list comprehension, and when a normal for loop? Answer: Where we can, we should use list comprehensions, because they are more readable and more efficient in terms of execution time, because they are optimized by the interpreter and run at a speed somewhat close to C speed, while normal for loops are executed normally by the interpreter at normal Python code speed. However, if we want to do something that has side effects or complex logic, sometimes it's worth sticking to normal loops.
18. What are the differences between for and while loops in Python? Answer: Mainly syntactic and performance-related. A while loop will always be executed by the Python interpreter, while a for loop used in list comprehension, for example, can be optimized and run much faster.
19. What is an iterator and an iterable? Difference between iterator and iterable.
20. `float("NaN") == float("NaN")` what's the result?

21. `True is True` and here?
22. `1 is 1` and here?
23. Write a program that checks if a string provided by the user is a palindrome (a sequence that reads the same forwards and backwards).
24. Write a program that counts how many times a given digit appears in a sequence of digits provided by the user.
25. Write a function that takes a list of numbers and returns their sum and arithmetic mean.
26. Write a program that creates a dictionary containing statistics about letters (how many times each letter appears in the text).
27. Write a program that reads a CSV file and displays its contents in table form.
28. Write a program that implements the merge sort algorithm.
29. Write a program that implements the quicksort algorithm.
30. Write a program that implements Dijkstra's algorithm for finding shortest paths in a graph.

16.3 Git

1. What's the difference between merge and rebase? Answer: Merge creates a 'merge commit', while 'rebase' kind of pastes commits from the merged branch. Besides that, using rebase, you can really mess up git logs. Which is better? Depends who you ask, there are different schools.
2. Differences between Git and GitHub. Answer: Git is a tool - version control system, while GitHub is a service where we can host our Git repositories and generally collaborate with others.
3. What does git stash do? And git stash pop?

16.4 Http/Rest

1. What is REST? Answer: google it
2. What determines what type of content can be sent to/from the server? Answer: Headers, specifically content-type.
3. Where do parameters go in a request? Answer: In the URL. ? and &.
4. 403 vs 401 Answer: Forbidden vs unauthorized. Difference between when user is not logged in (no access) vs is logged in but doesn't have permission.
5. HTTP response codes 1xx 2xx 3xx 4xx 5xx Answer: Check wiki.
6. What is JWT? Answer: JWT is simply an authentication method where we have a specific three-part token generated using some secret key/certificate, which contains a specific payload.
7. Is it possible to have authentication at the load balancer level so that the application doesn't have to do it additionally later? Answer: Absolutely. JWT ftw.

8. Ways to authenticate a request: basic auth and tokens - describe and characterize how they differ. Answer: Google it
9. Where does the server get username and password in basic auth? Answer: Header. Which one?
10. How does the browser know that a given response should be displayed as json and not as plaintext? Answer: Again - CONTENT-TYPE
11. Difference between PUT and PATCH. Answer: One requires providing all serializer fields, the other doesn't. Why? Which method requires all fields, which doesn't?
12. Available HTTP methods/verbs. Answer: Google it.

16.5 Databases

1. What is a join in SQL, how does it work, example
2. Database normalization - what is it, what's it about.
3. Types of databases. Relational and non-relational, a bit about them.

16.6 General Programming Concepts

1. What is OOP.
2. What is SOLID - name at least one
3. Let's say I type something in the browser window and click enter to go to a given page. What happens under the hood then?

16.7 Data Structures

1. What is a tree?
2. Balanced binary tree?
3. How does a hashmap/dictionary work?
4. And a set?

16.8 Rock Paper Scissors

1. Write a rock_paper_scissors() function that simulates a random game of rock paper scissors and then displays which player, p1 or p2, won the round. Does someone always win? Can there be a tie? Try to make your solution as short as possible. Try not to use ifs at all. Yes, it can be done without them. I managed to write the function simulating a single random game in 5 lines of code total, without an if.

2. Now add code that will call such a simulation 10 times and print its result to output.

Example of my solution:

```
import random as ra

def rock_paper_scissors():
    gestures = ['rock', 'paper', 'scissors'] # order here is important
    p1, p2 = ra.randint(0, 2), ra.randint(0, 2)
    return "P1: {0}, x P2: {1}, Result: {2}".format(
        gestures[p1],
        gestures[p2],
        {0: 'tie', 1: 'p1', 2: 'p2'}[(p1-p2)%3]
    )

for i in range(10):
    print(rock_paper_scissors())
```

16.9 Number Operations

Here are two exercises to do.

1. How can you implement multiplication by 2 in Python without using the multiplication operator, exponentiation, pow function, sqrt?
2. What about division by two considering similar restrictions as above?
3. Write an is_prime(number) function that checks if a given number is prime.
4. Same as above, but in one line of code.

16.10 Statistics from Logs

Here we'll go in English. The task description and potential solution below. The whole thing can be further simplified and refactored, but I'm providing here a solution that I quickly scribbled during a live recruitment. 30 minutes.

1. You are given a large log file which stores user interactions with an application. The entries in the log file follow the following schema: `{userId, timestamp, actionType}`. Calculate the average session time across all users. Assume that data yet get is valid and not sorted in any way. Assume that number of records that are `opens` will be equal to `closes`.
2. What if we want to modify the code to also calculate average session time per user?

```
from collections import defaultdict
from enum import Enum


# User level 1 -> 1512, 2-> 17
events = [
    [1, 1435459573, "Close"], [1, 1435456566, "Open"],
    [1, 1435462567, "Open"], [1, 1435462584, "Close"],
    [1, 1435462567, "Open"], [1, 1435462584, "Close"],
    [2, 1435462567, "Open"], [2, 1435462584, "Close"]
]


class ActionTypeEnum(Enum):
    OPEN = "Open"
    CLOSE = "Close"
    @classmethod
    def opposite(cls, action_type: str) -> str:
        if action_type == cls.OPEN.value:
            return cls.CLOSE.value
        return cls.OPEN.value

class Event:
    def __init__(self, user_id, timestamp, action_type):
        self.user_id = user_id
        self.timestamp = timestamp
        self.action_type = action_type

    def calculate_difference(self, second_event):
        if self.action_type == ActionTypeEnum.CLOSE.value:
            return self.timestamp - second_event.timestamp
        return second_event.timestamp - self.timestamp


def calculate_avg_session_time(events: list) -> tuple:
    time_sum, number_of_sessions = 0, 0
    user_level_sum = defaultdict(int)
    user_level_matches = defaultdict(int)

    user_time_mapping: dict = defaultdict(lambda: defaultdict(list))
    for event in events:
        wrapped_event: Event = Event(*event)
        opposite_action = ActionTypeEnum.opposite(wrapped_event.action_type)
        user_id: int = wrapped_event.user_id
        if first_user_record := user_time_mapping[user_id][opposite_action]:
            first_event = first_user_record.pop()
            user_session_time = wrapped_event.calculate_difference(first_event)
            time_sum += user_session_time
            user_level_sum[user_id] += user_session_time
            user_level_matches[user_id] += 1
            number_of_sessions += 1
            continue
        user_time_mapping[user_id][wrapped_event.action_type].append(wrapped_event)
    user_level_average = {
        user_id: user_level_sum[user_id] / user_level_matches[user_id]
        for user_id in user_level_sum.keys()
    }
    return time_sum / number_of_sessions, user_level_average
print(calculate_avg_session_time(events=events))
# (1213.0, {1: 1512.0, 2: 17.0})
```

Task shamelessly borrowed from a recruitment process.

16.11 Query Statistics

Create a class that will store the total execution time of a given query. As input data, you will receive records containing the query id and duration. Records can be partial, meaning one id can have multiple records. Such values should be summed together.

Additionally, implement a method `get_top_k_records` in this class that will return the specified number of queries with the longest execution time.

```python
from collections import defaultdict
from typing import Iterable

events = [
  {"id": 2, "partial_execution_time": 10},
  {"id": 1, "partial_execution_time": 15},
  {"id": 1, "partial_execution_time": 12},
  {"id": 3, "partial_execution_time": 25},
  {"id": 3, "partial_execution_time": 10},
  {"id": 4, "partial_execution_time": 15},
]

class QueryStats:
  def __init__(self):
    self.query_execution_times: dict = defaultdict(int)

  def add(self, event: dict) -> dict:
    self.query_execution_times[event["id"]] += event["partial_execution_time"]
    return event

  def get_top_k_records(self, k: int) -> Iterable:
    return sorted(self.query_execution_times.items(), key=lambda record: record[1], r


query_stats = QueryStats()
for event in events:
  query_stats.add(event)

print(query_stats.get_top_k_records(5))
```

Task shamelessly borrowed from a recruitment process.

16.12 Order Book

Implement an Order Book that handles regular orders with Price Limit (Limit Order).

Then add support for Iceberg Orders. What this is and how it works you can read in technical documentation of exchanges. For example: SETSmm and Iceberg Orders, SERVICE &TECHNICAL DESCRIPTION from London Stock Exchange. Or just google it.

We read data from standard input as subsequent orders and update the book in real time. If a transaction occurs, we print it to standard output. After adding each order, we print the updated book to standard output.

16.12.1 Input Data Format

JSON. Each order has a “type” and “order” field. Type contains the order type - Iceberg or Limit. Order contains the order data.

- “direction” order type (“Buy” or “Sell”),
- “id” unique order identifier (positive integer),
- “price” price (positive integer)
- “quantity” order size (positive integer).

Iceberg orders have an additional field - `peak`, which denotes the peak of this order.

We assume that the input data is correct.

```
{"type": "Limit", "order":
{"direction": "Buy", "id": 1, "price": 14, "quantity": 20}}
{"type": "Iceberg", "order":
{"direction": "Buy", "id": 2, "price": 15, "quantity": 50, "peak": 20}}
{"type": "Limit", "order":
 {"direction": "Sell", "id": 3, "price": 16, "quantity": 15}}
{"type": "Limit", "order":
 {"direction": "Sell", "id": 4, "price": 13, "quantity": 60}}
```

16.12.2 Output Data Format

When we read information about a new order and want to print the updated state of the order book, the JSON object we should print should have the following properties/attributes: `buyOrders` and `sellOrders`. Each order should contain “id”, “price” and “quantity” fields. Sorted by price. Buy orders non-increasing. Sell orders non-decreasing. If the price is the same, priority goes to the time the order was added. Example:

```
{"buyOrders": [{"id": 2, "price": 15, "quantity": 20},
               {"id": 1, "price": 14, "quantity": 20}],
"sellOrders": [{"id": 3, "price": 16, "quantity": 15}]}
```

Example session (sorry for ugly formatting, copy and fix it yourself):

```
{"type": "Limit", "order":
 {"direction": "Buy", "id": 1, "price": 14, "quantity": 20}}
{"buyOrders": [
    {"id": 1, "price": 14, "quantity": 20}], "sellOrders": []
}
{"type": "Iceberg",
 "order": {
     "direction": "Buy", "id": 2, "price": 15,
     "quantity": 50, "peak": 20
 }}
{"buyOrders": [
    {"id": 2, "price": 15, "quantity": 20},
    {"id": 1, "price": 14, "quantity": 20}
],
 "sellOrders": []}
{"type": "Limit",
 "order": {"direction": "Sell", "id": 3, "price": 16, "quantity": 15}}
{"buyOrders": [
    {"id": 2, "price": 15, "quantity": 20},
    {"id": 1, "price": 14, "quantity": 20}
],
"sellOrders": [
    {"id": 3, "price": 16, "quantity": 15}]
}
{"type": "Limit", "order":
 {"direction": "Sell", "id": 4, "price": 13, "quantity": 60}}
{"buyOrders": [{"id": 1, "price": 14, "quantity": 10}],
 "sellOrders": [{"id": 3, "price": 16,"quantity": 15}]}
{"buyOrderId": 2, "sellOrderId": 4, "price": 15, "quantity": 20}
{"buyOrderId": 2, "sellOrderId": 4, "price": 15, "quantity": 20}
{"buyOrderId": 2, "sellOrderId": 4, "price": 15, "quantity": 10}
{"buyOrderId": 1, "sellOrderId": 4, "price": 14, "quantity": 10}
```

Another example:

```
{"type": "Iceberg", "order":
 {"direction": "Sell", "id": 1, "price": 100, "quantity": 200,
"peak": 100}}
{"type": "Iceberg", "order":
{"direction": "Sell", "id": 2, "price": 100, "quantity": 300,
"peak": 100}}
{"type": "Iceberg", "order":
{"direction": "Sell", "id": 3, "price": 100, "quantity": 200,
"peak": 100}}
{"type": "Iceberg", "order":
{"direction": "Buy", "id": 4, "price": 100, "quantity": 500,
"peak": 100}}
gives us
{"sellOrders": [{"id": 3, "price": 100, quantity: 100},
                {"id": 2, "price": 100, quantity:
100}], "buyOrders": []}
```

Example solution on my github - github.com/grski

Task shamelessly borrowed from a recruitment process.
I'm not including the solution here as it's too long.

16.13 URL Shortener

In this task, please create a URL shortening service. Use Python and any framework. For example, django or fastapi.

So a service that shortens URLs, e.g., from https://codesubmit.io/library/react to http://short.est/GeAi9

Similarly, the reverse operation also works.

- Language: **Python**
- Framework: **any**
- Endpoints:
  - /encode - encodes full URL to shortened
  - /decode - decodes shortened to full URL.
- Both return JSON
- How the encoding will be done is an implementation detail. Choose as you see fit. As long as it can be encoded and then decoded. You don't need to attach a database - we can keep it in memory.
- Create documentation on how to run and use.
- Write tests.
- Use GIT and show your reasoning through commit history
- Write code as if it were going to production. Clean, elegant, beautiful.

Example solution on my github - github.com/grski
Task shamelessly borrowed from a recruitment process.
I'm not including the solution here as it's too long.

16.14 Static Site Generator

What does this mean? Check Google under `SSG` or `static site generator`. Generally, it's a piece of code, a program that generates static pages based on some foundation, something opposite to the current approach, i.e., creating SPAs. In my case, I decided to create a very simple system for my own blog. Here I leave the implementation details to you. Let this be part of the task - write down requirements, list functionalities, etc. Let your imagination run wild. The minimum I expect is delivering functionalities sufficient for running a blog - index with a list of posts and single posts.

```python
def find_all_posts(directory: str = "posts") -> Iterable[str]:
    """ All the md posts - both extensions .markdown and .md"""
    return find_all_markdown_and_md_files(directory=directory)


def find_all_pages(directory: str = "pages") -> Iterable[str]:
    """ Similar to the one above, but searches for
    posts - another directory. Both .md and .markdown """
    return find_all_markdown_and_md_files(directory=directory)


def find_all_markdown_and_md_files(directory: str) -> Iterable[str]:
    """ Base method that finds both .md and .markdown recursively
    in a given directory and it's children. """
    md_files: Generator = iglob(
        os.path.join(directory, "**", "*.md"), recursive=True
    )
    markdown_files: Generator = iglob(
        os.path.join(directory, "**", "*.markdown")
    )
    return chain(md_files, markdown_files)
```

There are comments in the code, so I won't describe it additionally. Next piece of code:

```python
md: Markdown = Markdown(
    extensions=["tables", "fenced_code", "codehilite", "meta", "footnotes"]
)

def render_markdown_to_html(md: Markdown, filename: str) -> str:
    """ Markdown to html. Important here is to keep the reset() method. """
    return md.reset().convert(open(filename).read())

def render_jinja_template(template: Template, context: dict) -> str:
    """ Rendering jinja template with a context and global config. """
    context_with_globals = {**context, **CONFIG}
    return template.render(context_with_globals)

def build_meta_context(md: Markdown) -> Dict[str, str]:
    """
    This builds context that we get from Meta items from markdown like
    post/page Title, Description and so on.
    """
    return {key: "\n".join(value) for key, value in md.Meta.items()}
```

Next fragment:

```
def build_article_context(article_html: str, md: Markdown) -> Dict[str, str]:
    """ Contant that'll be used to render template with jinja. """
    return {"content": article_html, **build_meta_context(md=md)}


def add_url_to_context(jinja_context: dict, new_filename: str) -> dict:
    """ Builds and adds url for a given page/post
    to jinja context. """
    jinja_context["url"] = f"{BASE_URL}{new_filename.replace(f'{DIST_DIR}/', '')}"
    return jinja_context

def save_output(original_file_name: str, output: str) -> str:
    """ Saves a given output based on the original
    filename in the dist folder"""
    new_location: str = os.path.splitext(
        os.path.join(DIST_DIR, original_file_name)
    )[0] + ".html"
    new_directory, _ = os.path.split(new_location)
    if not os.path.exists(new_directory):
        os.makedirs(new_directory)
    output_file: TextIO = open(new_location, "w")
    output_file.write(output)
    return new_location

def gather_statics() -> None:
    template_statics = os.path.join(TEMPLATE_DIR, "static")
    if os.path.exists(template_statics):
        shutil.copytree(
            template_statics, os.path.join(DIST_DIR, "static"),
            dirs_exist_ok=True
        )
```

Putting it all together:

```python
def render_blog() -> None:
    """ Renders both pages and posts for the blog
    and moves them to dist folder."""
    posts: Iterable[dict] = reversed(
        sorted(render_posts(), key=lambda x: x["date"])
    )
    render_all_pages()
    render_index(posts=posts)
    gather_statics()


def render_all_pages() -> None:
    """ Rendering of all the pages for the blog.
    markdown -> html with jinja -> html"""
    template: Template = jinja_environment.get_template("index.html")
    for filename in find_all_pages():
        render_page(filename=filename, md=md, template=template)


def render_page(
    filename: str,
    md: Markdown,
    template: Template,
    additional_context: dict = None
):
    additional_context = additional_context if additional_context else {}
    page_html: str = render_markdown_to_html(md=md, filename=filename)
    jinja_context: dict = {
        "page": {"content": page_html}, **additional_context
    }
    output: str = render_jinja_template(
        template=template, context=jinja_context
    )
    save_output(
        original_file_name=jinja_context.get("slug", filename), output=output
    )


def render_posts() -> Iterable[dict]:
    template: Template = jinja_environment.get_template("detail.html",)
    return [
        render_and_save_post(md=md, filename=filename, template=template)
        for filename in find_all_posts()
    ]
```

```
def render_and_save_post(md, filename, template) -> dict:
    """ Renders blog posts and saves the output as
    html. md -> html with jinja -> html"""
    article_html: str = render_markdown_to_html(md=md, filename=filename)
    jinja_context: dict = build_article_context(article_html=article_html, md=md)
    output: str = render_jinja_template(template=template, context=jinja_context)
    new_filename: str = save_output(
        original_file_name=jinja_context.get("slug", filename), output=output
    )
    return add_url_to_context(
        jinja_context=jinja_context, new_filename=new_filename
    )


def render_markdown_to_html(md: Markdown, filename: str) -> str:
    """ Markdown to html. Important here is to keep the reset() method. """
    return md.reset().convert(open(filename).read())


def render_index(posts: Iterable[dict]) -> None:
    md: Markdown = Markdown(
        extensions=["tables", "fenced_code", "codehilite", "meta", "footnotes"]
    )
    template: Template = jinja_environment.get_template("index.html")
    filename = "index.md"
    additonal_context: dict = {"articles": posts}
    render_page(
        filename=filename,
        md=md,
        template=template,
        additional_context=additonal_context
    )
```

This is described in more detail in my blog post, and the full source code is on github - braindead. There you'll also find commit history, description, etc. Take a look.

16.15 One-liner

Write a one-line Python generator or iterator expression that returns the sequence of integers generated by repeatedly adding the ascii values of each letter in the word "Close" to itself. The first 10 integers in this sequence are: 67, 175, 286, 401, 502, 569, 677, 788, 903, 1004. Assume any needed Python standard library modules are already imported.

Answer:

```
islice(accumulate(cycle("Close"), func=lambda x, y: x + ord(y), initial=0), 1, None)
```

Such a puzzle.

17 Case Studies of Various Applications

Here we'll discuss and analyze different implementation cases, talk about decisions and System Design, why something is done one way and not another, etc.

17.1 URL Shortener

The task was to design a URL shortening system, assuming we would handle quite significant traffic. I won't describe it step by step, instead I'll include here the notes from the conversation, a brief description, and the questions I asked.

1. Do we include authentication, or will it be handled outside the service? What's the business need? `What do you think?` For GET requests decoding shortened URLs, auth isn't needed. For creation, it might be useful to be able to assign a user to the link they shortened. This would allow for future editing or management of shortened URLs. `So we'll include it` In this case, I would use JWT auth, simply due to preferences plus the possibility of later pushing auth to the Load Balancer level when we scale.
2. What will be the write-to-read ratio? That is, how many GETs per each created URL. `Let's assume many many reads per one write.` Then a regular database like Postgres. If needed, we can add batch creation of reads with intermediate cache storage and scale the database first vertically, then create a cluster with a write-only master and several read-only replicas/slaves.

```
/ - base url for our api
/v1 - base url for v1 api
POST /v1/registration
POST /v1/login -> return a JWT token
POST /v1/short-urls
GET /v1/short-urls/{unique-id}

Table: users
Email: email | varchar
Password: --- (hashed)

Table: urls
id: integer | pk
unique_id: url | varchar | unique
original_url: url | varchar
created_at: datetime

POST[create] /short-urls
Request: {"url": https://google.com} ->
Response: {"shortened_url": "{api_entrypoint}/short-urls/{unique-id}"}
1. authenticate and authorize the user
2. take the payload and validate it ->
```

```
-> it has required fields + required fields are of proper type
3. save it to the database after generating a random unique id for the url
id in this case might be eg. couple char long alphanumeric value or a slug
4. put in cache (up for debate)
5. return the response with shortened url

GET[detail] /short-urls/{unique-id} -> {"url": "https://google.com"}
1. get the unique-id from the url
2. check if it's present in the cache
3. if it is, just return the value from cache
4. if it's not, then query the database for that given unique id
5. if not found, return 404
6. if found return the object from the db and also add it to the cache with a proper

FastAPI - async framework at the front
Cache will improve the perf drastically
As it'll be mainly reads and waiting for the network calls, cache will
help a lot here.

Route53 <- domain parked
Load Balancer <- from AWS
Microservice <- 1. EC2 that has scaling enabled or managed k8s cluster
-> or AWS provided container runtime service
Microservice cache <- redis
Persistent storage <- RDS with Postgres
```

17.2 How One Digit Can Break an Application - A Case Study

Recently at work, I came across what I consider a quite interesting case to investigate. We received a report from a client that files for selected resources stopped working - clients couldn't access thumbnails, information, sometimes there were problems with downloading itself, and so on, generally a major screw-up that needed to be fixed ASAP.

Well, I sat down to it then.

My first suspicion fell on myself. Why? Well, a few months ago, I clearly remember fiddling with something about how files are returned and so on, in any case, I was practically certain it was my doing.

I looked a bit like Nawałka after the World Cup.

17.2.1 Phew

Nevertheless, after analyzing the reports, it turned out that the problem started appearing even before I modified anything in this particular module, so I had to exclude the possibility that it was my doing.

Hmm, okay, what now. Well, let's move on, maybe something interesting will come up.

Looking through the code, I didn't find anything interesting, no changes, nothing. So what could have caused the problem to appear suddenly?

Well, to find the answer to this question, I had to look at how files are served in the client's application.

It turns out they are hosted on external servers, depending on the region, which process everything, handle authentication and so on, and then return the requested content.

Okay. Time to look at this process more closely.

17.2.2 File Path

The first common feature that emerged among all reported problematic files was that they came from one area - meaning they were all served by one server. Good, that's some clue.

I tried then to recreate the path that a typical user and their request takes, and check the application's behavior during such a path. I dug through the source code, found the code responsible for handling the display and delivery of content links to users, and generated a regular URL to a given resource, and indeed - it doesn't display at all, receiving some error. But wait. Let's look at the request itself, what's happening there.

No error appears in the response, so the server doesn't see any problem and returns a normal response. Hmmm, interesting...

17.2.3 What Turns Out?

It turns out that although the request we're making should clearly return us a response with video files, for some reason, instead of returning what it should, the response has the type... of an image.

Let's check what's in the response body. First the size - compared to the original files it roughly matches. And what do the headers say?

After a small analysis, it turns out that although the response type contains an image, the server in the response body serves nothing else but the given video file, which if saved, is a correct movie, hmm interesting. And now we know why the content isn't displayed at all - how is the browser/application supposed to display a movie when the response tells it to display this content as an image?

17.2.4 Let's Look in the Database

I decided then to browse through the database in search of other video files, to see what mimetype the server would generate for them.

Well, my search quickly showed me that the server has no problem identifying files of type .avi or .mpa as video and returns them with the correct mimetype, enabling the browser to behave properly. Why this behavior?

17.2.5 Format?

Could it be the format's fault? Because .avi works, .mpa too, but it turns out that certain .mp4 files don't.

Not at all, as I managed to find working .mp4 files - that's not it.

And how is it on other servers? On others everything works - no .mp4 file that wouldn't work for unknown reasons. Even more interesting.

How the file looks on disk I remembered a bit about how files are generally saved on disk, how you can recognize that a given file is of a particular type - certainly not by extension, and I thought that this might be it - something might be wrong in the file headers and that's why the server doesn't correctly recognize the type. Maybe something goes wrong during saving. I compared then two files in the same format using HxD - one working, the other not.

Unfortunately, despite small obvious differences in the hex code, where it should be the same, everything matched, even though at first it seemed I found an error in the file structure.

Damn.

I came up with another idea then - I'll check it with another program, mediainfo, let's see if it correctly decodes the information about the file format and so on.

And here again surprise - mediainfo had no problem reading the file correctly, all information about it. Why then, if other programs correctly recognize files as video, does the server not do it? I don't know, but nothing, I'll try on others, maybe I'll get lucky here.

And I started browsing and browsing. Nothing. Completely nothing. Suddenly. . .

17.2.6 Eureka

. . . I noticed a certain fact. It turned out that every working, correctly recognized file was encoded using a certain library in version let's say 2.11.2 or older, while all those files that were encoded with this library in any later version were incorrectly recognized by the server. And here everything became clear.

I looked at the library's website and what turned out? Around the same time when the problem started appearing, there was a release of a new version of the library. This confirmed my belief that someone somewhere updated one library on just one server, not taking into account the fact that the code of the application running on the server uses dependencies in a certain specific version and older versions, and is adapted to them, which is why files encoded with newer versions are not correctly detected, and every file that lands on the server, before being served to users, is processed using the currently installed library.

17.2.7 Summary

Even though there was no actual programming here, I must admit I had great fun while performing this task. A bit like a detective uncovering the secrets of a great crime.

It also showed me that general knowledge, about which people often say 'why do I need this', sometimes comes in handy.

To sum up, sometimes it's enough to change one digit from 2.11.2 to 2.11.3 for something to suddenly stop working somewhere, and a developer's work isn't always just churning out new landing pages.

18 Culture

A few words about the culture of a programmer's work. About attitude and such. While seemingly secondary matters, some of them actually play a very important role in the software development process.

18.1 A Bit About Managers/Leaders

A good manager/team leader is someone who can either lead a team to ruin or give them wings. Teams are the basic building blocks of companies. If a manager can destroy or empower a team, they can do the same to the entire company. This emphasizes how important it is to have someone normal in a leadership role. How does one empower or destroy? By operating on what the team consists of. On people. A person is the basic and most important unit in a company, whose willingness, and mainly generated unwillingness to work, is strongly dependent on the influence of leaders. I'll describe some of my thoughts on this, or rather observations about the path that some go through and how it changes them in effect.

On my programming path, I've met many managers in many different companies. Some were wonderful, positively influencing my life, others were complete idiots, and yet others are now almost like family to me. After some time, I started noticing certain patterns that occurred almost always, which allowed me to isolate three boundary levels of being a manager.

18.1.1 Level One: Excel Master

Excel Masters are often freshly minted managers, leaders who don't have much experience yet, but are full of ego, pride, and bravado. The world is open before them and ready to be conquered! They love their spreadsheets, record everything in them, optimize, sometimes practice micromanagement. These are people in the early period of their managerial career, where one can get a bit intoxicated by their power initially.

This type of person often sees only one thing: money. Oh, I've met too many of these until a certain point, way too many. Money-grubbers. And they are precisely the most dangerous type of manager that can happen to a company, among the three I've mentioned.

Superficially, Excel Masters look quite good. The numbers add up, plans, presentations, it's a dream. But not quite. In reality, everything falls apart. Why are they the most dangerous type? Because of the facade they impose - they're not the ones usually suspected of being the source of problems, but after getting to know this species, it's hard not to notice them in the future, and you can often recognize them easily. How? Just find the most unhappy team in the company. A silent team that persists in quiet resignation, seemingly for some elusive reason. Don't look for a team that complains. They still have hope. Look for something else.

Teams that complain - they still have the desire to do at least one thing - complain, grumble. True failure and resignation occur when there's nothing, when people are quiet, when they're so tired of everything that they prefer to remain silent and pretend everything is fine, but unfortunately, it's so bad that even from the outside, upon closer inspection, you can see the truth. When enemies argue about something or banter, that's not yet a fight, but a conversation, they're talking in a way, it's some form of communication. The real fight and drama begin when any form of communication ceases and actions appear, deeds and action. Deadlines aren't met, projects don't succeed. Often, team turnover is also very high. People either get entangled in such a team out of resignation and simply stay there for a very long time, rotting in the depths of despair, or they flee immediately. Usually at the top of such a team, you'll find what I call a Level One Manager - Excel Master.

Excel Masters see only numbers, only money. They never admit to a mistake, never think they made a mistake - it's always someone else or the team that's at fault. They don't see people, they don't see team colleagues, they see resources. Resources that can be utilized, or rather exploited, that can be "optimized". There's nothing more. Like simple mathematics. If you have a task for 80 hours, it will take about two weeks of work by one programmer, if you hire two programmers, it will take a week, right? Everyone always performs work at a very steady pace, regardless of context and situation, right? Private life has no influence on professional life at all, right? Margin and profits are all that matter, right? Do these numbers look good? If yes, then everything is going as it should, right? RIGHT?! Technically yes. Practically no.

A manager like this destroys the spirit. They destroy the people who make up the team.

Without people, you don't have a team. Without people, your product won't be launched. Without people, deadlines won't be met, but they don't see this. Resources are just that. Resources. At least in their heads.

This is simply insane and destructive. Nothing worse than feeling unappreciated, than feeling you're just a cog in a big damn machine that can be replaced with one wave of the manager's hand. Let's not kid ourselves, in reality, it's sometimes like that, although each of us isn't THAT special, or at least partially, but damn, leaders and managers are there to make that feeling disappear. At least partially.

I've met many such people. They have one thing in common. The havoc they produce.

The best that Level One managers can achieve is very short-term projects. Projects where people don't have a chance to get to know each other, encounter difficulties and real challenges. I'd say a maximum of a few weeks. This is possible from level one because in such a short time, they can't destroy their team members enough. They can't attack them to the state of despair, unless they're very talented at it.

Usually, these people either remain locked in this attitude for a loooooong time or try various pieces of advice to maybe improve or fix the team, because they often don't see the error in their behavior. In any case.

This usually leads to learning about empathy. Which brings us to...

18.1.2 Level Two: Empathetic Excel Acolyte

Here things get a bit interesting, I'd say. A person who has reached this level is in an interesting place. Usually driven by constant failures and striving to improve their spreadsheet, team performance, and other VERY IMPORTANT CORPORATE INDICATORS, they learn about empathy. Oh yes! EMPATHY! What we should promote to make people feel better, more connected to the team and company, so they seem to perform better? Sounds good, right?

Hell no. This is the problem with Level Two managers. They see empathy only as a tool, a tool to force the team to work. Or to better performance. In any case, this is not true empathy in any way. It's just calculated manipulation in reality.

Don't get me wrong - a true leader SHOULD and MUST know a bit about manipulation, it should be natural for them, but they shouldn't abuse it. Good managers should be able to read people and their emotions, but never use this against them, never manipulate their team to perform certain things. This doesn't work in the long run, but can sometimes bring short-term benefits. Which is pointless.

This skill should be used for compassion, knowing when to back off, give a team member some slack sometimes, for various reasons you've observed.

So although a Level Two manager is a bit better than Level One, they're still bad, at least they try and pretend. That's something, right? What kind of pretending and what kind of pretended empathy am I talking about here?

Oh, hi Susan. Good job at the empathy workshop with our Agile trainer last week. It's very important to promote empathy in our company! Your mom died yesterday, so you'd like to leave early today? Oh, not that kind of empathy. Maybe another workshop with the Agile/Empathy trainer will help you improve that? The company will pay for it! Think about it Susan and keep it up, champ!

Something like that I'd say. It's a bit hard to describe, but you can feel and notice it most of the time. It's when you feel like cattle being taken care of by not so great an owner who doesn't love their animals, they just do the bare minimum so they have meat of acceptable quality. Yeah, something like that.

This is Level Two. Although still better than Level One, it's also a threat, just a smaller one. This one is a bit harder to notice on the surface, but after talking with them a bit, you can also get it.

Level Two can actually deliver some projects. At most short- and medium-distance ones. Maybe a few months, probably at most half a year. In this period, you can still maintain the facade, pretend, so the team can still be okay, will even achieve nice results. But this is a trap when you get a longer project, where people really get to know each other or have worked together before. This is where they ultimately fail. Sometimes crushingly. There aren't many things worse than the feeling when you realize your manager/leader was a Level Two. That they cared about you only to make you perform better, manipulated you. Resentment follows bitterness, a sense of betrayal.

Back to the topic - long-term projects. Now this is where the real challenge and fun comes in, and with it enters…

18.1.3 Level Three: Human

Usually, Level Three is Level One or Two who has simply failed enough times, who has been broken enough times and put back together by the passage of time and self-reflection. Rarely is it a genius type who was just born to be a leader/manager.

In most cases, Threes have experienced many times in their life failure and total, crushing defeat. They're aware that we're just humans and nothing more. And humans, on some levels, are extremely tough and resilient, but sometimes can be so brittle and delicate that even the smallest thing can make them shatter into pieces. How do they know? Because they've been there.

They also know that even if you're broken, you can be whole again. Even if you shatter, you can be rebuilt. The same goes for the team.

These are managers who truly care about you, who you'll befriend and genuinely like.

Why do I consider them so important? Because if you have a sincere bond with them, and they have a sincere bond with you, your partnership, your teamwork will simply be better. When there's shared trust and understanding throughout the whole team, magical things happen. There's no need for micromanagement. Work just starts happening because people are willing to do it, happy to come to work.

Even if what you're working on is total garbage, at least you have each other, right? You know your manager has your back and your team's back. You know it's not just about KPIs and your results. Which ironically makes you perform better.

It's just a simple, but very deep and meaningful act of seeing you as you are - a human, not a cog in a machine. It sounds so simple, and yet sometimes reaching this level takes years and is difficult. That's how it is. People are complicated beings.

How to recognize Level Three? They're not afraid of their failures, they can be truly sensitive and show their weak side, share, make you open up a bit and relate. They have very good relationships with their team and know the people who work under them. That's the wrong word. They don't have people who work under them. They have people who work with them. That's one of the bigger differences here. Being a manager and having a partnership with the team to achieve a common goal versus having a team of slaves to manage.

I don't know what else to say here. These are simply truly empathetic people who, by not focusing on improving the performance, improved it only by this act. A bit ironically.

How well do you know your team colleagues? When was the last time you did something nice for the people you work with? Do you all feel safe at work/in the team? These levels don't only apply to managers or leaders. They can apply to almost anyone, and everyone ultimately influences what kind of manager you get.

A manager manages the team/project, but the team also teaches the manager many things. It's a two-way relationship. You're a wave - influencing other waves around you, so... Think about it for a while.

Now stop for a moment and ask yourself this question: which one are you?

To sum up. Stop focusing so much on KPIs, Excel spreadsheets, Performance data. Focus on the people you work with, see them as they are - humans, not resources or numbers in Excel spreadsheets. Get to know them, their strengths and weaknesses.

A good general knows where to assign a specific soldier based on their character, a good carpenter knows where to use a specific piece of wood based on its characteristics. A good manager is the same. It's to everyone's benefit. Be a good manager. Be empathetic. Focus on people. That's it.

We've all been there. If you're currently at Level One, or Two, don't worry. It's part of the process. I'm not a Three either. Heck, I'm not even a manager myself, so what do I know!

18.2 Design Culture

An important and essential element of any piece of code or system you create is its design. What does this specifically mean?

Well, it's about making the code you write **thoughtful**. This is where something called `incremental design` comes in. This means that generally, for projects less trivial than Hello World, you can't create a perfect solution right away. Instead, you need to think of delivering a solution as a process we go through step by step.

First, we should understand the given problem, grasp it well, and throw it into our brain to process it into analogies and concepts understandable to us.

Mistakes and being wrong are part of this process. Get used to it. Almost never will your first idea for solving a problem be complete or correct. Instead, it's worth breaking the problem into tiny pieces that, in their triviality, will approach the aforementioned Hello World. Of course, you need to have taste and feel here, as you can't break things down too much either.

Where is the boundary? The feel, in my opinion, will come with time and practice.

Something I recommend from myself is the following steps:

1. Understanding the problem and asking questions. This is the place for thoughts, analogies, and trying to get familiar with the given problem. Ask, even about seemingly stupid things, understand what this is about and what the problem is. Understand the problem's **domain** and **model** it in some way in your head. What does this mean? Present an abstract concept and imagine it in a somewhat less abstract, understandable way for yourself.
2. Draw out the concept from the point above or summarize your thoughts in written form. This is where it's worth confronting your imagination with reality, asking the right people if this is what it was about. Diagrams are incredibly useful here. I emphasize, however, that this is not about a detailed implementation plan or something like that. No. A general outline of what the problem is about.
3. Only now is it time for the first implementations of the smallest pieces. A task divided into small parts is a task easy to grasp. A small piece is easier to grasp at once than a huge problem. Maybe apply TDD now? Write some piece of code that reflects the model and some piece of the solution to the problem, test and confront with reality. It's worth noting here that code should be written so that it's as easy as possible to change, replace, refactor later. The more complicated and connected to other fragments the code is, the harder it is to edit and refactor. This is not, however, an encouragement to create a huge monster that will contain tons of boilerplate code and do everything, while doing nothing. Nothing could be further from the truth.
4. Having in hand something that sensibly actually solves the problem and reflects the problem model well, now you can refactor a bit, improve, polish, make the code easier and better written.
5. Steps 3 and 4 we repeat as needed. Maybe also step 2.

I'm not encouraging here to immediately create a **waterfall** model from the top, where we plan everything in detail.

Iteration by iteration we improve our design and code, moving forward. That's what it's about. Thanks to this, we can test tiny pieces, validate our ideas on the fly (e.g., with the client by delivering tiny fragments, but often) and live with bearable code.

Programming, paradoxically, if done well, often doesn't rely on programming.

Well, the most time should be spent analyzing the problem and thinking, planning. Changing a concept is easy, it's just a thought or diagram. It's a relatively cheap operation. Changing code, on the other hand, can often be costly. Your task is to always **think** and keep various considerations in mind.

Figure 4: Tom Scott - meme. I allowed myself to use the image, I hope it doesn't bother anyone.

The situation like in the meme above can be unpleasant. Remember that in the future, you'll probably be maintaining your own code, at least for a while. So do yourself a favor and think twice about the code you write, put in the effort. It's not worth being sloppy and lazy sometimes. I mean, paradoxically, it's the lazy ones who think the most, because it's easier to think than to type code. So let's follow the path of the true, not apparent, lazy person and take care of design. A well-designed application can be grateful to work with both for the programmer and the end user.

Sometimes it's enough to spend an hour thinking to save yourself 4-8 hours of typing code.

By the way, don't be afraid that someone will hold you accountable for conceptual work based on counting lines of code you produced. That's not what creative work is about. No sensible employer will hold you accountable for writing 20 lines instead of 40. The number of lines of code produced is not a measure. Of course, I'm not talking about extremes, when you're a junior and for a whole month you added one commit changing one line, which probably means you're pulling someone's leg.

To watch: https://www.youtube.com/watch?v=flxmpq7_tdM

18.3 About Values

Supposedly, those who don't have values and principles talk about them the most, because the rest simply live by them. Despite this folk wisdom, I'll write a bit about them, but only lightly. I'd like to mention certain values that guide me in my work and that I personally like to see in myself and others. These traits are something I consider to be the foundation of a good programmer, coder, employee.

1. Honesty and open mind This is the foundation for me in everyone. People have problems talking to each other honestly and with communication. Every relationship, especially professional, should be based on honesty. This requires an open mind and putting ego in your pocket.
2. Respect (street people, you know, keyboard) Have respect for another person, client, colleagues. Respect their time, knowledge, and money. This way you can build relationships for years. Also respect yourself first and foremost, because if you don't respect yourself, it's hard for others to do so.
3. Empathy and being human At the end of the day, we're just people. Put yourself in someone else's shoes sometimes before saying something. A good atmosphere and trust built on empathy and sympathy for another person allows achieving synergy in the team that will incredibly streamline, but also make work more pleasant. Be human. Don't be a jerk.
4. Taking responsibility and being proud of your work When you create something, take responsibility for it, be proud of it. This principle has far-reaching implications like taking care of your code, delivering holistic solutions that have been tested, sometimes `going the extra mile` if needed, having and showing initiative instead of blindly following tasks.
5. Meritocracy In my opinion, we put ego in our pocket. Criticism, meritocratic I mean, of your solution, code, or even yourself, don't take personally and as an attack. No. These are rather valuable lessons from which you can learn, you just need to push ego to the background and focus on what's important, that is, the good of the project, client, delivering value. This doesn't mean you have to constantly sacrifice yourself, no. Respect yourself first and foremost, but I repeat. We put ego in our pocket and try to base things at work on logic, arguments, and discussions conducted in an adult manner. Additionally, I'd also like to mention that you shouldn't be afraid to propose your solutions, even if they somewhat contradict solutions of, for example, the leader, senior programmer, or client. If you work with the right people, they'll appreciate it and you'll choose the solution based on facts, not other factors.

6. Collaboration Be independent in your work locally, but globally oriented towards collaboration. What does this mean? Think and have somewhere in the back of your mind a broader perspective and try to coordinate your individual efforts so that they fit into some broader plan. An example here is: "I'll start implementing this task first, because this way I'll unblock the frontend."
7. Assume good intentions When in doubt, ask. That's it in short. If you work with someone, it means they probably went through similar verification as you. This means they're somewhat competent, that's my assumption. Additionally, I assume, unless facts prove otherwise, that everyone in the team has good intentions. So before I do something, if I have even a shadow of doubt about the nature of the situation, I prefer to ask what someone meant. Maybe a bad day, maybe I misinterpreted something, maybe someone's wife gave them a hard time in the morning. Who knows.

I believe that being guided by these points above won't harm you. It's worth keeping them in the back of your mind. Because in IT, it's often not about IT but about people and their problems. When you think about it, that's what you're paid for - for solving business problems, problems of specific people, making their lives easier. That's what the big bucks are paid for. Delivering value. And this requires having appropriate traits and understanding, being a certain kind of person. I think the attributes above are ones that will allow you to shine in the team and company. These are the basics of being a normal person, although sometimes rarely encountered. Stand out not only with hard IT skills but also with being a good person.

Damn, but poetically and bravely.

18.4 Kaizen

I'd like to say a few words about the Kaizen philosophy. What does this mean? It's a Japanese thought whose main theme is continuous, constant work on yourself and making small improvements that together make up something bigger.

Generally, this works very well in our profession and actually everywhere.

Every day we should make the code a tiny bit better by adding new functionalities or improving existing ones. Every day we should at least read a tiny bit, learn something new. After some time, compound interest starts working here and it turns out that from this daily, small effort, something huge grows. 1% daily gives 3700% yearly after taking into account compound interest. But damn.

So Kaizen or The Slight Edge is something that should be present in your work culture.

18.5 Managing an Investment Portfolio

Mysterious header. What's this about? Namely, that each of us invests in life, even if we don't play the stock market. How so? Yes. You yourself are your biggest investment. Your knowledge, career, life.

Knowledge is very valuable. It's something you should constantly invest in, as it usually gives huge returns.

18.6 Relation from Facilitating the First Retro

Recently, I had the opportunity to facilitate a retrospective for the first time in my life, this will be my summary of this experience.

The idea started when one of the founders of the company I was working for at the time asked me if I'd like to do this.

Me - an agile/scrum-sceptic/whatever-sceptic, facilitating a retrospective? Sounds like a recipe for disaster.

Well, I agreed without hesitation. Despite the fact that for the most part I'm not a big fan of all the current buzz around agile, scrum slaves (I love this term and often overuse it) and so on, additionally one pleasant thing that comes to my mind when I hear retrospective is the Jinjer song. And seriously - either it's just me or some people have gone crazy, coming up with increasingly extreme and strange practices when it comes to AGILE, which don't bring greater effects beyond annoying developers and wasting business people's time.

In any case - seeing a lot of such buzz, I immediately become skeptical. Hence I decided to check what's on the other side of the fence.

Of course, I was aware that there are many valuable practices, things, habits, and thinking patterns that can be learned from the Agile crowd, it's not like I'm saying everything agile is useless. Don't interpret my words that way. It's just that some people do crazy things that aren't Agile and call them Agile. Especially in Poland it seems. And this sometimes leads to madness. I've seen it with my own eyes, no psychotherapist helped later. I digress.

Since this was my first time, I was offered help. The company arranged a few hours with our friendly agile trainer, who helps us in building certain processes in the company. We had a short conversation before I planned my retrospective.

I learned a bit about setting the stage, gathering data, insights, cool tools to use, and so on. This quick session, despite my initial doubts about it, helped me quite a bit. So big thanks here and shout out to Marcin Konkel!

18.6.1 Ready, set, go

So the day and hour came. The meeting started. I got nervous when we were all in place. If it weren't for the fact that this was my first retrospective, then due to the way we work, and we work 100% remotely, this was a remote retrospective. Personally, I think remote communication is more difficult - you need to be more precise, accurate, more communicative and attentive. It takes some time and skill to get used to it.

18.6.2 Setting the stage

Before we started, I decided to establish a few clear rules and say out loud my assumptions, which were: We don't interrupt each other during the conversation. Considering how we're conducting the retrospective remotely, we need to stick to this even more than ever. If you have something to add outside your turn/speaking time, raise your hand to the camera. The speaker or moderator should notice this and give you a chance. The first person to speak in a given round for a given exercise is chosen by the facilitator. That person gets some time to answer the question while others listen. After finishing speaking, they point to the next person who should speak. After we established these rules, I mentioned one thing.

This was a retrospective for our team. I'm part of this team, actively participating in development and work. Today, however, I was playing the role of moderator. I was nothing more. So I clearly marked that today I'm doing just that - facilitating. Monitor, manage, facilitate and moderate conversations. Let them do the rest.

Moreover, I'm a bit of a direct guy - instead of using any fancy tools, I just went with conversation. No post-it notes. No boards. Just our faces, voices, and me, quietly noting the whole retro in the background. Yes - terrifying that you actually have to talk to people. Ugh.

18.6.3 Warm-up

Initially, I wanted to lighten the mood a bit, get people talking, because I was perfectly aware of what they were feeling at that moment. Some of them were interrupted in the middle of work, thrown out of flow, to participate in this meeting. They're now in a different context, so it was time to bring them here, to this moment.

How to do this? I asked everyone a simple question.

What, whether in private or professional life, made you happy recently?

This allowed the team to have a new, small look at each other. Perhaps finding a new way of looking at each other. I know it sounds simple, but that's how it was in reality. Everyone surprisingly shared happy little moments from their private life and you know what? I felt that this strengthened the bonds that connect us all as a team.

18.6.4 Retrospective in the past

Before this, we conducted another retrospective in the past with another agile trainer, as a result of which we declared several goals. Now it was time to check them. So I asked each person who had a defined goal about the status of their goal, how it's going and their opinion, experiences so far.

All people unanimously stated that the previous goals were needed, are done or in progress and somehow improved our work. Nice.

18.6.5 Goodies - gathering data

We work in a very multicultural and international team, and yet somehow it turned out that at this retrospective there were only Polish colleagues from the team. Being Polish myself, I know about our quirks, and one of them is complaining. It's like hyperinflation, once it starts, there's no going back.

To set the right atmosphere, I decided to first talk about good things, to put people in a good mood. I asked them all the question:

What are some things that you think we do that make us deliver faster, better and generally contribute to us doing better work or what are you satisfied with recently?

Here everyone shared their perspective, observations, and so on.

18.6.6 Baddies - gathering data

So, while they were still in good moods and the aura was positive, I dropped a bomb that sometimes opens the gates of hell on Polish retrospectives - what went wrong? Let's get specific. Enough sugarcoating. The onion way.

And here the complaining started. Or so it seemed to me - that it would start. Instead, it didn't.

I asked the question:

> What do you think we did wrong recently? What have you observed that makes us deliver less, slower? What makes your work less pleasant?

Instead of mindless babbling, the team did something nice. They presented constructive and smart observations, insights, and comments. Honestly, it was quite amazing. Sometimes I'm surprised by how smart the people I work with are.

18.6.7 Happies - still gathering data

Since I like to change moods like a sine function changes its value, I changed the topic again.

This time I decided to facilitate talking about things we do that make us happy. This was a different question than the first one - good things we do - because here it wasn't about delivering, productivity, and generating value. It was about human bonds and how we behave towards each other, how we interact. I thought it would be nice if they told each other what they notice in others' behavior and what they appreciate.

I don't know about you, but I like to pat myself on the back sometimes - a sign that someone appreciates my efforts.

So:

> What do you appreciate that we do? Not in terms of productivity, coding, or technicality, but just as a human - what makes you want to be with us and spend our time together?

And this, oh my, this point was quite sweet honestly. So if we had sweet, it's time for bitter again.

18.6.8 Stop, it hurts.

Now I decided to stir the waters a bit. I threw in the topic:

> Goals for improvement. What should we stop doing? What should we achieve to improve?

And here I decided that this retrospective shouldn't end with just empty talk. I mean sure, we improved some bonds, got to know each other a bit better and had a blast talking to each other (really!), but for me that's not enough.

So I threw in an idea.

Everyone, if they had such a wish, could voluntarily choose one or more tasks, their pain points, which they want to solve by the next retrospective.

Additionally, they also had to choose a ~~witcher~~, observer for their tasks. Owner might be a better word here. This person will be responsible for checking their progress and so on. Basically someone who will make sure the task gets done, besides that person. Someone who will nag them a bit, e.g., every other day ask the famous, at least in our company, question - What's the status here?

Why? While I trust my team colleagues to fully implement project-related things, these tasks were from the nice-to-have category and all, but we won't die if we don't do them. They weren't critical, just nice to do. Hence it would be good to have some responsibility and additional motivation to do them.

This also worked out - almost everyone decided on some tasks for themselves.

18.6.9 Inception

After this came time for a small inception. A retrospective on the retrospective. Basically a short round of feedback about today's retrospective, my facilitation, generated value, and so on. Apparently they liked it, or so they said. Yeah.

That would be about it. After this the retrospective came to an end. I survived and actually found it sensible. But on the other hand quite demanding.

I'm young, I have a problem with my ego, you know. It was hard to be on the sidelines, for most of the time just observing, being a moderator, instead of an active participant. I had a few moments when I wanted to abandon my moderator role and engage as a team member, starting my endless talking, throwing in my two cents, but I restrained myself from this, managed to remain in my role.

This was a bit humbling and satisfying, honestly.

18.6.10 Summary

Just so you know, later those tasks that we defined - were done. So it really didn't end with just empty talk. Apparently this idea with guardians for tasks also helped a bit - maybe it'll be useful to you too? I have no idea.

So well, Agile processes can also be cool and useful, that's first. Second, show empathy and check the other side of the fence, if you have some specific opinion or maybe strongly disagree with someone about something, try to understand. Maybe there's something valuable there, you just don't know about it. Finally, step out of your comfort zone. Thanks to this you'll develop. Coaching all the way. Who stole your dreams?

If you're also preparing for your first retro, maybe this article will be helpful to you. Additionally: protip. Take notes. I did. Later, after the retro, I turned them into a nice confluence page and tasks on Jira.

That would be it for today, I, Olaf, was your humble host, I hope you enjoyed it.

19 Paperwork and Money

A bit about employment contracts and compensation.

19.1 Understanding Compensation and Contracts

A small note for the less familiar: net salary is the amount you take home after taxes, gross salary is the amount before taxes, and the total cost that your employer pays for you is usually a completely different amount. This works differently when billing as a contractor, but that's a different story. I'll explain that in a moment.

When starting your career, it's important to understand the different types of employment contracts available in your country. Each has its pros and cons, which you need to be aware of to make the most beneficial decision for yourself in a given situation.

Generally, there are three basic forms of employment:

1. Full-time employment contract
2. Contract work (fixed-term or project-based)
3. Independent contractor (B2B/freelance)

19.2 Contract Work

Let's start with contract work - this is typically the most flexible form of employment for both employer and employee. A contract work agreement concerns performing specific tasks or services for a defined period.

The main advantages are: - Flexibility in working hours - Often higher hourly rates than full-time employment - Clear project scope and deliverables

The main disadvantages are: - Usually no benefits (health insurance, paid time off, etc.) - Less job security - Need to manage your own taxes and insurance

19.3 Full-time Employment

The most traditional and secure option, but often the most expensive for employers. In short, it's obvious - you get all the standard benefits: health insurance, paid vacation, sick leave, and so on.

The main advantages are: - Job security - Benefits package - Easier to get loans/mortgages - Employer handles taxes and insurance

The main disadvantages are: - Often lower take-home pay compared to contract work - Less flexibility - More bureaucratic processes

19.4 Independent Contractor (B2B/Freelance)

Now a few words about being an independent contractor, what it is, who needs it, why, and an analysis of when it's worth switching to this form of work.

19.4.1 What is Independent Contracting?

When browsing job offers for programmers, you can often come across salary ranges - it's rather standard in the industry to provide a range of possible compensation.

Often the form of cooperation is also provided. More and more often, instead of the well-known employment contract or contract work, something like "B2B" or "freelance" appears. What does this mean?

Independent contracting is a form of cooperation where you establish a relationship with a company not as an employee but as a business entity. This means that instead of becoming an official employee, you become a service provider for a given company, you need to register your own business and handle your own taxes and accounting.

19.4.2 Advantages of Independent Contracting

Being an independent contractor carries a series of responsibilities, but at the same time gives certain possibilities that we don't have when working based on other forms of cooperation.

19.4.2.1 Tax Benefits Probably the most significant advantage. As an independent contractor, you can often take advantage of various tax deductions and write-offs that aren't available to regular employees. This can include: - Home office expenses - Equipment and software - Professional development - Business travel - Health insurance premiums

The exact benefits depend on your country's tax laws, but generally, independent contractors can reduce their taxable income significantly.

19.4.2.2 Flexibility and Control The next major advantage is flexibility and control over your work: - You can choose your projects - Set your own rates - Work with multiple clients - Define your own working hours - Choose your work location

19.4.3 Disadvantages

There are quite a few of these. That's why it's worth considering whether independent contracting is the right choice for you - it's not suitable for everyone.

19.4.3.1 Administrative Responsibilities Remember what I mentioned earlier? About taxes, accounting, and paperwork? With other forms of employment, all of this is handled by your employer. You're not interested in anything, except checking if the payment has arrived.

When you switch to independent contracting, all formal obligations fall on you. Your client only needs to remember to pay your invoice.

So what do you need to do as an independent contractor? There's quite a bit of this: - Handle your own taxes - Manage your own accounting - Issue invoices - Maintain business records - Handle your own insurance

You can automate most of this by hiring an accountant, but this comes at a cost. Alternatively, you can do it yourself if you're comfortable with the administrative work.

19.4.3.2 No Employee Protections As an independent contractor, you're not protected by employment laws. Everything depends on what you have in your contract. And there can be various things in it - from lack of notice period, to responsibility for any mistakes you make in work and the damage they cause.

You're also not protected against illness. Not working? No money. At least by default, because in the contract, if the client allows it, you can include an option for paid time off.

Similarly with vacation. There's no paid vacation here. Not providing services? No money.

19.4.3.3 Harder to Get Loans Another disadvantage of independent contracting is that it's often harder to get loans, especially mortgages. Independent contractors are usually subject to various restrictions when they want to take out a mortgage.

The first restriction you'll encounter is the required period of running your business. Most banks require that you've been running your business continuously for at least 12-24 months.

Additionally, you'll need to provide more documentation: - Business financial statements - Tax returns - Proof of consistent income - Business registration documents

19.4.4 Summary

To summarize, independent contracting is not a form of employment for everyone. It has its disadvantages like responsibility for various things falling on your head, but also advantages, because with greater responsibility comes greater control over your work and finances.

Is this a good form for you? I don't know. I'll almost certainly say one thing though, that if you have less than, let's say 6, and preferably 12, months of experience, and the amount that a potential client offers is not significantly higher than what you'd make as an employee, then better leave it and wait.

19.5 Money

About a very important thing, which is money. Without shame and openly.

19.5.1 How to Talk About Salary

I often see that we're ashamed to talk about money. Well, in my opinion this is a bit of a mistake. Human greed is a thing, but only if accompanied by certain principles and rules.

If this is your first job, which suggests the fact that you're reading this book, then well, honestly speaking, money isn't very important. In your first job you're actually only doing one thing… Learning.

At the beginning, for the first few days/weeks, even someone with 10 years of experience brings losses, because they need to be onboarded to the project, get to know the team and so on.

And this is natural. When hiring a Junior, every employer must be aware that they'll need to invest in them a bit first, so they can start being useful and bringing profits. So we're aware that this onboarding period will be longer.

So don't feel guilty that you're only learning at work and that's bad, nothing could be further from the truth.

This is a calculated move by every company, you as an employee who's training, growing in value and over time, when you reach a certain level, this investment in you will very quickly pay off through the work you do.

Let's be honest, your employer has to make money on something. They might make it on their own product, maybe on outsourcing, in any case, one way or another, they indirectly make money on you and your skills. Is this bad? No, a normal thing.

Nevertheless, please, never blame yourself for spending time at work learning, especially at the beginning, it's simply, as I mentioned, the employer's investment in you, which is actually quite profitable for them in the longer term.

To summarize, if you're just starting out, then the rate isn't important. Does this mean you should work for a proverbial bowl of rice? Absolutely not. You need to respect yourself and it's not worth going below a certain level. Simple.

When setting your rate, it's worth taking into account: - The region where you live - Other offers and ranges they offer - Your skills and experience - The company's size and financial situation

Do a little research. If you ask for too much, but you're a nice candidate, then the company will probably want to renegotiate rather than reject you right away, so don't be too afraid, but on the other hand don't throw amounts from outer space, because that also says something about you.

19.5.2 Raise != More Take-Home Pay, About Time

Now a few words about how a raise doesn't always mean more money in your paycheck. Why?

Well, the matter is simple - we need to realize that we're basically selling our time to our employer and we need to calculate our paycheck relative to it. You can't always negotiate a raise in monetary form, because for example the company doesn't have additional funds, a specific rate is planned for someone with your experience, or something similar. You can then approach it from a different angle. How?

Probably the simplest example is reducing full-time - instead of working 5 days a week, ask for 4. Succeed? You've just reduced your full-time to 4/5, keeping the same salary, meaning effectively that's like a 20% raise per hour of your time. Pretty nice result in my opinion.

Another way to get a raise without a raise might be switching to remote work mode. I'll immediately note - this isn't for everyone and not in every company can such a work model be implemented.

However, if remote work suits you, there are no obstacles in the company and you manage to negotiate it - great. Why?

Because the time you spend on work isn't only the time you sit in the office. You need to calculate commuting time too. And we often spend a lot of time on this and sometimes it causes our actual rate to decrease by about 10% on average, assuming we have about 30 minutes to work.

With remote work there's no this problem - the time you spend commuting is exactly 0. Unless you go to some coworking, but that's then by choice, not by force. Your will.

Time is quite important, so let's talk about it.

The only thing we have in life is time. We need to manage it wisely, because we can't buy time, we can't add it to ourselves or make it run slower. Every day it slips away from us.

Programming is an industry you can get carried away with, so to speak. Relatively quickly you can reach really nice earnings here. This makes it possible to get a bit 'fixated' on this money and start working a lot. Too much. And they're not the most important thing after all.

Nevertheless it is. And we as programmers trade precisely our time usually - we sell the employer hours of our life in exchange for money. So know their value. Don't let yourself be possessed by the rush for money that will make you work 14 hours a day, because you'll earn more.

It's not worth it. Unless it gives you pleasure, you don't have a family, and your social life doesn't exist. Then you can.

The question is different though - how long can you endure mentally and physically?

19.6 Understanding Equity Compensation

Many tech companies, especially startups and high-growth companies, offer equity as part of their compensation package. But what exactly is equity, and why should you care about it? Let me break it down in simple terms.

19.6.1 What is Equity?

Equity means ownership. When a company offers you equity, they're offering you a piece of ownership in the company. This can be very valuable if the company grows or gets sold. Think of it like buying a small piece of a house - if the house's value goes up, your piece becomes more valuable too.

19.6.2 Types of Equity Compensation

1. **Stock Options**
   - **What they are**: The right to buy company stock at a fixed price in the future
   - **How they work**:
     - You get the right to buy shares at a specific price (called the "strike price")
     - You can exercise (buy) these shares later when the company's value has (hopefully) increased
     - You only make money if the company's value goes up above your strike price
   - **Example**: If you get options at $10 per share, and later the company's stock is worth $50, you can buy at $10 and sell at $50, making $40 profit per share
2. **Restricted Stock Units (RSUs)**
   - **What they are**: Actual shares of the company given to you over time
   - **How they work**:
     - You receive actual shares (not just the right to buy them)

    - They typically “vest” (become yours) over 4 years
    - You don’t need to pay to receive them (unlike options)
  - **Example**: If you get 100 RSUs and the company’s stock is worth $50 when they vest, you get $5,000 worth of stock

3. **Virtual Stock Option Plans (VSOPs)**
   - **What they are**: Cash payments tied to company performance
   - **How they work**:
     - Instead of actual shares, you get cash payments based on company growth
     - Often used when giving actual shares isn’t possible
     - Simpler than traditional options
   - **Example**: If the company grows 50%, you might get a cash bonus equal to what you would have earned with real options

19.6.3 Understanding Vesting

Vesting is how you earn your equity over time. Think of it like a pie that you get slices of over several years.

Common vesting schedules:

1. **4-year vesting with 1-year cliff**
   - First year: Get nothing if you leave
   - After 1 year: Get 25% of your equity
   - Next 3 years: Get the rest gradually (usually monthly or quarterly)
2. **Monthly vesting**
   - Get a small piece every month
   - More predictable but less common
3. **Performance-based vesting**
   - Get equity when the company hits certain goals
   - Could be tied to company growth, product launches, etc.

19.6.4 Evaluating an Equity Offer

When you get an equity offer, here are the key things to understand:

1. **How much equity are you getting?**
   - What percentage of the company does it represent?
   - How many shares is that?
   - What’s the current value of those shares?
2. **What’s the company’s potential?**
   - Is it a startup with high growth potential?
   - Is it an established company with steady growth?
   - What’s their plan for the future (IPO, acquisition, etc.)?

3. **What are the terms?**
   - When can you sell your shares?
   - What happens if you leave the company?
   - Are there any special conditions?

19.6.5 Common Questions About Equity

1. **"Should I take a lower salary for more equity?"**
   - Depends on the company's potential
   - Early-stage startups often offer more equity but less salary
   - Established companies usually offer less equity but more salary
   - Consider your personal financial needs
2. **"What if the company fails?"**
   - Most startups fail, so equity could become worthless
   - Don't rely on equity for your basic living expenses
   - Think of it as a potential bonus, not guaranteed income
3. **"When should I exercise my options?"**
   - When you believe in the company's future
   - When you have the money to buy the shares
   - Consider tax implications (varies by country)

19.6.6 Tax Considerations

Equity compensation can have tax implications:

1. **Stock Options**
   - You might owe taxes when you exercise (buy) the options
   - You might owe more taxes when you sell the shares
   - Different rules apply in different countries
2. **RSUs**
   - Usually taxed as income when they vest
   - You might need to sell some shares to pay the taxes
   - Additional taxes if you sell at a profit later
3. **VSOPs**
   - Typically taxed as regular income
   - Usually simpler than traditional options
   - Check your local tax laws

19.6.7 My Experience with Equity

In my career, I've worked with different types of equity:

1. **Early Stage Startups**
   - Larger equity grants
   - Higher risk but potentially higher reward
   - More influence on company direction
2. **Growth Stage Companies**
   - Mix of salary and equity
   - More stable but still significant upside
   - Clearer path to seeing the value
3. **Established Companies**
   - RSUs as part of standard compensation
   - More predictable value
   - Regular vesting schedules

The key lesson I've learned is to evaluate equity as part of your total compensation package, considering both the potential upside and the associated risks. While equity can be extremely valuable, it's important to balance it with immediate needs and other forms of compensation.

19.7 My Career Journey

I'll share my personal story to give you a real-world example of how a programming career can progress. This is my authentic journey, with all its ups and downs.

19.7.1 Early Days

I started my programming career at 18 with no prior experience. My first position was as a Junior Python Backend Developer, working remotely from a small town for a company in a bigger city. This was quite unusual - I haven't met another person who started their career fully remote without any full-time experience. Somehow, I managed to make it work.

My first contract was for a two-month trial period in November 2017. I started at 18 PLN/hour, which translated to about 3,000 PLN monthly for full-time work. After the trial period, I received a 23% raise to 22 PLN/hour (about 3,700 PLN monthly). I actually tried to negotiate for 25 PLN/hour, but my boss declined, promising we'd revisit the topic later. Looking back, I can't help but laugh at my ambitious request for a 40% raise after just two months of learning.

19.7.2 First Major Jump

After 8 months in my first role, I moved to a new company where my rate increased by about 60% to 37.50 PLN/hour (6,300 PLN monthly). This demonstrated a key lesson: changing companies can lead to significant salary increases. As they say, the easiest way to get a raise is to change jobs.

19.7.3 Transition to B2B

After about 9 months of experience, I transitioned to B2B (business-to-business contracting) when my student status ended. My rates remained similar but with better tax optimization, allowing me to earn between 8,000-10,000 PLN monthly after taxes. I also started working for multiple clients, including some international ones where my rate jumped to 60 PLN/hour.

19.7.4 Career Acceleration

At 1.5 years of experience, I received an offer of 86 PLN/hour for fully remote work. This was a significant milestone - I had officially joined what some might call the "high-earner programmer" club. I was working remotely, with a great team, and felt financially appreciated.

19.7.5 Entrepreneurial Phase

At 21, with 3 years of experience, I started my own company while also working full-time. This period was intense - I had 7-8 full-time employees and was working multiple jobs simultaneously. My earnings fluctuated wildly, from 100 PLN/hour to a record 600 PLN/hour. Monthly earnings ranged from 15,000 to 60,000-70,000 PLN.

However, this period also taught me important lessons about work-life balance. I got carried away by the money, pressure, and a bit of a "God complex." Working too much eventually took its toll, and in Q4 2021, I took a year-long break to recover and reflect.

19.7.6 Return and New Opportunities

When I was exactly 23 with 5 years of experience (or 8 counting overtime), I signed a significant contract that included: - Base salary equivalent to 36,000 PLN gross on employment contract - Stock options worth 130,000 EUR, vesting over 5 years - 26 days of paid vacation - 3,000 EUR annual development budget - 20,000 PLN signing bonus - Fully remote work

This seemed like a dream come true, but it ended abruptly when I was fired after publishing an article about human-centered management that conflicted with the company's actual practices. However, this turned out to be a blessing in disguise - I immediately found a better opportunity with another company I had previously turned down.

19.7.7 Current Situation

Today, I'm working as a Lead Python Developer, earning 36,500 PLN net on B2B (about 210 PLN/hour), fully remote. I'm much happier with how things turned out, as the new company better aligns with my values and approach to work.

19.7.8 Key Lessons from My Journey

1. **Early Career is About Learning**: Don't focus too much on money at the start. Your first job is primarily about gaining experience and skills.

2. **Changing Companies Can Accelerate Growth**: Sometimes the best way to increase your earnings is to move to a new opportunity.
3. **Work-Life Balance is Crucial**: No amount of money is worth burning out. I learned this the hard way during my entrepreneurial phase.
4. **Values Matter**: It's important to work for companies that align with your values. The highest-paying job isn't always the best one.
5. **Remote Work is Powerful**: Starting my career remotely was challenging but ultimately gave me valuable experience in self-management and communication.
6. **Continuous Learning is Key**: Each step of my journey required learning new skills, whether technical or business-related.
7. **Network and Reputation Matter**: Many of my best opportunities came through connections and maintaining a good professional reputation.

This journey shows that with dedication, continuous learning, and a bit of luck, it's possible to progress quickly in the tech industry. However, it's crucial to maintain balance and stay true to your values along the way.

20 Pros and Cons of Being a Programmer

This book is about programming, right? There are millions of such books, all encouraging you to enter the world of IT, to program, because it's the future, because it's living the high life, millions of coins in your wallet, glory and whatnot.

Well, no, I don't agree. We need to pour a small bucket of water on a significant portion of people who aspire and dream about the IT industry. This is not some magical, perfect job that has only advantages and no disadvantages. ABSOLUTELY. Get this out of your head.

20.1 Disadvantages

To fight this belief, I decided to write in my book why... programming might not be worth it.

20.1.1 Health

Programming means long hours at the computer, often too long. This is not healthy. Eyes, spine, other things. There's a reason why the saying "sitting is the new smoking" has started circulating recently.

A sedentary lifestyle and staring at a computer destroys our bodies, and it does so quite severely. If someone doesn't have enough self-discipline to counteract this sitting with gym, exercise, or physical activity, then honestly, the money earned from work won't be of much use because they won't have the health to enjoy it.

Another issue is stress. Believe it or not, but when you finally reach a place where you're a somewhat normal programmer and start being responsible for certain things, deadlines and so on, stress will inevitably appear.

How can you not stress when you've broken something in the code and suddenly it turns out the client is losing tens of thousands every hour because you did something stupid? What about when a trivial problem you were supposed to solve by the end of the day turns out to be something intricate that you'll spend the next week on, despite your declarations that it would be done in a few hours?

Even though these are foreign questions to me, as I simply can't stress about work, despite being heavily involved in it, I know that many people have such problems.

It sounds a bit like bragging, but for some reason, that's just how it is. I simply live in an untroubled belief that ultimately I'll manage, everything will work out, and it will be fine.

20.1.2 Continuous Learning

What about the time you have to dedicate to learning, constant learning? This industry changes so quickly that if you don't develop yourself, you're out of the game. Simple. I'm not joking.

If you don't keep up with new things or innovations for a year, two, three, suddenly you become somewhat worthless in the market or reach stagnation where your knowledge doesn't increase, nor do your skills.

And in our industry, stagnation means not just lack of progress, but essentially regression. This mainly applies to junior developers though. They need to develop strongly and extensively. Otherwise, it doesn't make sense.

Seniors or good regular developers have mastered certain basics and skills that are somewhat key and unchanging, which allows them to be a bit lazier.

Moreover, the greater your experience, the less you actually code, and the more you read code, mentor others, participate in meetings, do conceptual work, architectural work.

For some, the fact of being forced to continuously learn is a disadvantage, for others an advantage. I actually enjoy this because I'm really interested in what I do, I love my profession, so having to stay current with certain things is a nice excuse to spend a bit more time with code.

However, there are people for whom programming is simply work from 8 to 16 and that's it. And that's also normal, correct, and okay, so to speak. Not everyone has to be a complete enthusiast. We also need people who just do their thing and that's it. They can be good programmers, they won't be the best, the cream of the crop, but still.

Such people, meaning average, or not quite outstanding, are also needed, it's completely normal. In fact, most people operate on this principle. Enthusiasts are rare.

20.1.3 Bad Projects

We don't always work on cool apps. The reality is that sometimes there are nasty projects that no one wants to do, but someone has to. And you know what?

You, as a Junior, have a greater chance that your employer will throw you into such crap. Why? Because if something is already messed up, you can't make it worse, that's one.

Two, at the beginning you don't have much choice, unless you're a good programmer. And actually, I hope you become one.

Such are the realities, unfortunately. Because when we're already wallowing in such a mess that nothing can help, well, you can basically throw a complete newbie in there, because whatever they do, it won't be worse, and it will be cheaper.

I've given a somewhat extreme example here, but very probable if you land in a Januszsoft, which often happens in first jobs.

20.1.4 Stereotypes

IT is also a stigma - prepare yourself for the fact that all friends, family, and other people you know will expect great knowledge from you in areas that are even slightly related to programming.

Which phone to choose, which computer, which router, how to configure it, and so on. It doesn't matter that you don't work as an admin, devops, or some other help desk. Everyone expects you to know about everything related to IT/technology. It's not a major disadvantage, but after some time it can be irritating.

Protip: never offer to "reinstall someone's system" anywhere, especially for free. Then everyone will come to you with their technological problems and blame you for them.

If you want to be a good soul, at least take that proverbial bottle as compensation.

Besides, the moment you reveal yourself as a programmer, so to speak, a large portion of people will attach a certain label to you regarding how much you earn. In some, it arouses envy. Although this is actually an advantage, because thanks to this you can easily spot who cares about your material status and who is a true friend.

20.1.5 'Employee's Market'

Another disadvantage is that the market is not as rosy as it seems. What, did the bootcamp trainers tell you that 60k programmers are missing, right?

Well no, nonsense. Because many specialists are needed, but... Exactly, specialists. Not beginners. People who already have knowledge and 2, 3, 4, or 5 years in the industry are needed. There are plenty of beginners who want to become programmers because they heard it pays well, everywhere you look, there's some wannabe programmer.

Don't get me wrong, I have nothing against people who want to retrain or become programmers. Great, let them do it. I'm only amused by people who think that writing hello world in python and one CRUD app will make them programmers who will be capable of working after two weeks and that it's all so easy and pleasant.

And this is no joke, most job postings with the keyword 'junior' or 'no experience' can receive up to 500 CVs. I've seen such cases. I suspect that soon the number of CVs received for junior positions will reach four digits. There are really multitudes of people who want to get in. The problem is that most of them don't know anything or have inflated expectations and zero understanding of the industry.

You really need to stand out a bit now to get into some sensible place. Sure, there are companies that even train programmers themselves and pay them for learning, guarantee work, but this usually comes with two things: a loyalty agreement for X years, that's one, two - working on projects that no one wants to touch. Besides, it happens very, very rarely that a company would go for something like this.

Unfortunately, life is not a fairy tale, here things are calculated. Investment in an employee is not selfless - it's exactly what I wrote, an INVESTMENT. It's made with the thought that it will return with interest.

I'm saying this more towards people who think that getting a job in IT is incredibly easy, that there's a huge hunger in the market and so on. No. I'm appealing to you, to those who picked up this book with the approach that they'll read one or two little books, program for two weeks at home and boom, 15k per month. IT'S NOT LIKE THAT.

You have a position of strength when you already have several/dozens of years of experience on your back, considerable knowledge and reputation. Then you can pick and choose among offers.

Or maybe not, because everyone wants to get into these decent companies, so you have greater competition and it can go either way. The better the company, the higher the requirements it can set, because it has more candidates. End of digression.

I'm not saying here that it's impossible to get a job without experience - look at my case, but it's not as easy as some describe it to be. You need to hammer this into your head.

On the other hand, let's not exaggerate - finding a job doesn't require Herculean effort. Provided that you have something in your head, follow the NBD principle and master solid knowledge and basics.

Willingness alone is not enough. I know this will sound surreal, but there's a whole crowd of people who declare they'll even work for free for half a year, just to grab some work in IT.

As always, everything depends on you, your projects, open source contributions, and what knowledge you possess. The greater your skills, the greater the chance you'll find some work, but you also need to remember that presentation and the way you search also count.

20.1.6 Dream Job

'Do what you love and you'll never work a day in your life.' Or something like that. I don't know who said it, but they're an idiot. Even if programming excites you as much as it does me, you'll still face a lot of things at work that aren't strictly programming but are necessary in the software development process.

Meetings, discussions, paperwork, convincing the client about certain solutions or their various whims. I could list many things here.

Of course, if you work in a decent company where work is sensibly organized, some of these elements are cut out. However, you can't cut everything out.

If you think you'll come to some job and just sit at your desk for 8h, creating awesome code, you're mistaken.

However, let's assume you're not. You sit in front of your computer all day and type code. Is this an ideal situation? No. Why?

First of all, mainly, a programmer's work is based on thinking. Before you code something, you need to think about it and think thoroughly. In fact, most of the time spent at work is thinking and reading code.

The actual typing on the keyboard is a small part. And that's very good. If it were otherwise, no one would be able to work 8 hours. Even in this model, where most of the time is research and thinking, it's hard to be productive for those 8h.

It won't be like that. Simply. It's unrealistic. In programming, there are tough moments, days when you feel like you don't know anything and who knows why you're working here - the so-called impostor syndrome. Contrary to appearances, a very common phenomenon.

There are also bad projects, tasks we don't feel like sitting down to, but we have to. Programming can really be exhausting, and very much so.

And I'm telling you this, a person who once worked in construction. In fact, sometimes I miss how simple it was. Sometimes I'd gladly set up some corners somewhere on some reveals. Perfectly, to a nice right angle. Run a goldband nicely over it. Oh yes. Then plaster twice and polish to a mirror finish. There's no better feeling.

Sure, such work tires you physically, sometimes very much, especially when you have to work on scaffolding at high humidity because the paint is drying, and in 35-degree heat in summer, or when you have to move a million hollow blocks or carry a pallet of 25kg bags of plaster to the 4th floor.

After such a day at work, you come home, eat, and basically go straight to bed. Clear enough. At least at the beginning, because later you get used to it and it doesn't tire you as much. Mentally, however, you're completely free.

I, during such work, simply closed myself in my world and my thoughts were elsewhere. Time passed very quickly, contrary to appearances. Sometimes even pleasantly, especially if the work crew was nice. Plus you have tangible effects of your work. You create something you can see, touch, see progress.

Really. I'll probably remember the time spent with a trowel in my hand quite fondly. It taught me a lot. If I had to name the hardest element of this type of work, it would be the financial aspect and the limitations imposed by a meager salary. Although even this has been changing recently, because now due to a strong deficit in the market, working as a physical worker, you can earn serious money. Seriously. Programmer15k sometimes hides.

In any case. What I mean is that after such a day at work, I could be less exhausted than after a day working as a programmer.

To summarize, a programmer's work, especially when you're fascinated by this field, is wonderful. However, it's not without disadvantages, it has quite a few. Some are prosaic, others serious, but they exist. For me, most of them don't bother me at all, but they can be persistent. It's important that you're aware of them before you make a decision, on a whim, to retrain and so on.

20.2 Advantages

Enough complaining, now it's time for advantages. Oh boy. Where to start.

20.2.1 Money

What's there to say. The money is right. However, it's worth noting that there are extreme differences. I know people who have similar experience to me, but earnings 20% to 80% lower than mine. In IT, there's a chance to earn well, quite a large one, but you also need to put in effort and know what you're doing. Plus soft skills come into play.

20.2.2 Development Opportunities

IT is such a dynamic industry that you can never complain about boredom. Things evolve here so quickly that development paths and possibilities are practically countless. That's super cool. On the other hand, as I already mentioned, for some this can be a disadvantage. You either move forward or die, because if you stand still, you're actually moving backward. It's starting to smell like Coach Mike. Every day we stray further away from God.

20.2.3 Interesting Work

To write a piece of code and create a sensible product, you need to understand how it works, what the business case is, what the context is, and so on. This means that you actually need to learn about that business issue. This means you can temporarily become like a person from another industry and try new things. For me, it's amazing that while working on new products, I often learn a lot about other industries, professions. That's super cool and interesting in my opinion.

Plus, the technical details and implementation specifics of our work can be really non-trivial, especially if someone has that engineering knack.

20.2.4 Often Remote

In current times, work in IT is very often remote. This carries certain disadvantages, which I've already written about, where socialization and social life especially suffer, at least for some, but at the same time it gives us many possibilities. Remote work is primarily great flexibility in schedule, life. Convenience - we don't lose time on commuting. However, the most important thing for me in remote work is probably that we can work for companies from all over the world and leverage the purchasing power of stronger currency from wealthier countries, living in a place where the local currency has weak purchasing power.

Speaking plainly, if you live in such Poland where the złoty is a token made of cardboard and the government regularly arranges to screw this token without lube, where the dollar is at five złoty, then earning in euros or dollars will probably give you even greater eldorado here compared to your colleagues who earn in złoty.

Additionally, a company that's from abroad and works in remote mode can also count. It's better to pay an engineer from the USA, where living costs are higher and they won't agree to work for less than 20k USD per month, let's say, or give 10k USD to a Pole who will probably do the work similarly or just as well, will be super happy with the salary and yet be half the price because with 10k USD in Poland you can probably live at a higher level than in the USA with 20k USD? Of course, the numbers are made up, but they illustrate the situation.

Poland is probably one of the few countries where the differences in earnings between the average salary and IT salary are so huge. You don't have this in Norway. Not necessarily in Switzerland either. Similarly in Germany. I mean, it exists, of course, but it's not differences of 10x the average salary.

20.2.5 Employee's Market

Despite debunking rumors about this in the part about disadvantages of working as a programmer, this didn't apply to everyone. If you really know your job and understand anything sensibly, the market demand is such that it's mind-boggling. Really. And it won't get smaller.

Recession is coming. Okay, inflated crap that has no right to live in a free market will die. However, the remaining companies will grow. The desire for optimization and cost reduction in various enterprises will also grow. What does this mean? Often automation of processes, reduction of employment, optimization of existing processes. Familiar things, often done using technology.

So in my opinion, in the IT sector we currently have a VERY strong employee's market, a qualified employee's market. This situation will only deepen in my opinion.

20.3 Summary

Being a programmer has its advantages and disadvantages. Honestly, more advantages in my opinion. However, there are also certain disadvantages that make this profession and some people's expectations more realistic. This is not some holy grail of work. No. It's a job like any other, currently fashionable and lucrative. For some, however, it will be torture that's not worth the effort, as they have better predispositions in other directions and should rather aim there.

I've listed here both disadvantages and advantages, maybe this will give you a bit of a broader perspective.

21 Remote Work

Let's talk a bit about remote work and how to do it properly.

Many people don't realize that remote work requires certain aspects to be handled well. Otherwise, it can become unpleasant in the long term. Remote work isn't just about working from home. When properly implemented, it requires changes not only in the work location but also in how we approach work.

21.1 Ergonomics

When working remotely, we sometimes forget about ergonomics and work hygiene. Being young, it's easy to fall into the myth of indestructibility. After all, the consequences of certain actions aren't immediately felt. It's only after years that certain things come to light. I'm saying this as a 23-year-old who has spent hours at the computer for many years and has been in the industry for 5 years. Despite this, I've already experienced what professional burnout and work-related health problems mean. I don't want you to share my fate. So take care of the ergonomics of your workspace. This will not only help you stay healthier but also make you more effective in your work in the long run.

I'll explain what this means specifically using my own example.

21.1.1 My Ergonomic Setup

Let's analyze my situation and the equipment I use.

1. External monitor + laptop. Simple matter. An additional external monitor allows me to work more efficiently by providing more screen space. It's nice if it's at least 27", high resolution. For some, FullHD is enough, for others 2K is too little. Personally, I lean towards 2K/4K. 4K might be a bit too much, so 2K seems like a good compromise. A good monitor won't strain your eyes. I use a 27" Dell refurbished monitor that I bought for 800 PLN. 2K resolution. This additional monitor really improved and streamlined my work.
2. Microphone When working remotely, it's nice to have a good microphone. For 200-300 PLN you can get really good equipment that ensures you're heard clearly, with noise reduction, etc. I recommend Thronmaxxy. The price-to-quality ratio is amazing. It's always nice when there are no problems understanding your coworker. The devil is in the details.
3. Webcam Usually, the built-in one is enough, or for more demanding users, an external webcam, or for the most demanding geeks, a digital camera connected to the laptop. It's nice to have occasional eye contact with coworkers. However, don't overuse it - a webcam isn't always necessary.
4. Standing desk. I'm terribly hyperactive. A standing desk helps me manage this. Additionally, the fact that I stand up from time to time is good for my health, back, and overall physical condition. Sitting isn't very healthy. Muscle contractions, spine problems. You can really mess yourself up.
5. Good chair If we're sitting a lot, it's best to do it on something comfortable. Some like exercise balls, others ergonomic chairs, others stools. There are many preferences.

I personally recommend a good chair like Ergohuman or something similar but cheaper. It's a one-time expense that pays off.

The above items are, in my opinion, the basics of ergonomic remote work that are useful if we want to maintain our health for years. Of course, you can work without these. You don't need to buy a thousand-zloty chair or a three-hundred-zloty microphone just for learning. No. However, if you're already a professional, don't be afraid to invest in your equipment. These are your work tools just like any others. You'll make life easier for yourself and others.

I know this might seem funny, but I subconsciously take people with a nice setup a bit more seriously. It's not an indicator of anything, but that's just how I am. It subconsciously signals certain things to me.

At the end of the day, of course, what matters is whether you deliver and aren't a jerk, but still.

Here are some tips for people working remotely:

1. Create an appropriate workspace: Make sure your workspace is comfortable and ergonomic. Avoid working from bed or the couch, as this can lead to spine problems and other ailments.
2. Set time boundaries: It's important to set time boundaries for your work to properly separate work and private life.
3. Maintain good communication: Remote work requires good communication with other team members and superiors. Make sure you have access to appropriate communication tools like internet messengers or video conferences.
4. Take care of your health: Remote work can lead to a sedentary lifestyle, which isn't healthy. Make sure you maintain proper posture at the computer and get up from time to time to stretch and do some exercises.
5. Maintain discipline: Remote work can be difficult if you don't have proper discipline. Set your daily schedule and stick to it as if you were working in an office.
6. Keep a routine: Remote work can be distracting, so it's important to maintain a routine, such as setting fixed working hours and taking regular breaks.

22 Loose Reflections

This chapter is a collection of my various thoughts, appeals, and whatever else comes to mind.

22.1 Backend is Not a Product

I wanted to write a few words not about programming per se, but about programming as work. I'm a relatively young developer, with about two years of commercial experience, and even less when it comes to meaningful team work.

That's not much. I'm not very old either, as I haven't even hit 20 yet. That's why I'm constantly learning many things and at the same time I see that other people, with more experience, older people, are similar - they're also learning. Hence the impression that it's worth addressing the topic I'm writing about today, because it seems that regardless of age and experience, many people don't understand this.

22.1.1 A Small Enlightenment

Recently, a light bulb went off in my head - the team's task is to create a solution, a product, something that satisfies the client, fulfills some goal, solves a given problem. Exactly.

What does this mean? I'm writing this from a backend developer's perspective, and here's the thing: backend is not a product in itself.

Neither is frontend. What exactly do I mean? Until recently, I had an approach like - I don't care that functionality X wasn't delivered on time/doesn't work/there are implementation problems - I did my part, the backend works and is beautiful, so it's frontend/devops/qa/whoever who didn't deliver and the product doesn't work.

This is completely the wrong approach, I understood that. Partly it helped me understand how the application delivery process looked in our team at one of my previous companies, and what problems we encountered along the way, which mainly resulted from such thinking. On the other hand, I was slightly directed to look at this problem in my thinking by my de facto mentor.

Here's a nice thing - instead of immediately telling me something, I was allowed to try something on my own, think about it, and then come to the correct solution independently - respect for that.

22.1.2 How It Should Be

Ending the digression - when we write backend/frontend, whatever, we can't just think about how cool, beautiful and amazing it will be from a purely backend/frontend perspective.

We need to think about how others will work with the code, the service we're creating.

Will the solution we're creating be convenient for the front? Did I clearly describe everything so that the person writing the API integration won't have any questions, because everything was clearly described in the documentation, additional notes? Maybe I can help them in another way, by talking to them, before, during and after implementation?

Good questions that, I think, are worth considering. It seems to me that we need to be careful not to fall into a kind of egoism, where we think that our work is the most important, we did it well, so someone else is completely to blame and that's it, we have clean hands. That's not how it works.

Of course, there are exceptions to the rule - someone deliberately sabotages work, whatever, but these are such rare situations that they're not worth mentioning. Programming is usually teamwork, and a key element of teamwork is... well, teamwork.

The team is very important, just as it's important that individual 'elements' of the team work well together. Thanks to this, the whole mechanism will work great. And that's what it's about.

Don't get me wrong - I'm not talking about constantly doing some 'team building', 'integrations', 'meetings', 'gatherings' or other crap often forced. No. I'm not an advocate of scrum or other similar or less similar methodologies. No.

22.1.3 An Appeal

The fundamental goal of this text is merely to plant a thought, I'd even say a request.

It goes like this: while creating your part of the code, service, whatever, think about the person who will read your code, who will use your API, maybe consult with them, describe everything, document it.

Put yourself in the shoes of that frontend, backend, ops person.

Maybe you can do something differently, which will still be a technologically good solution, and will allow the other side to do something faster, without slowing down your work? Who knows.

Just such a question. It's probably worth asking yourself. Of course, there's no need to go overboard in the other direction - creating a programming mess because the front asked or whatever - no, that's the wrong direction.

As usual - you need to find some middle ground, because going overboard in either direction is bad.

Concluding: let's not be such egoists. Backend is not a product, frontend isn't either. You can have a state-of-the-art backend/front/devops side, but if that other component doesn't work, the product will suck.

Such an approach, where you voluntarily do more for others, will simply make your project work better. The more you help others, the more they will be able to help you. Simple.

This absolutely doesn't mean you have to do someone else's work, I think you understand.

This is very important, especially now. It seems to me that we're starting to attach more and more importance to how we work with someone, because otherwise the team will fail.

So this is very important, which I've already emphasized more than once and will probably repeat many more times, especially to you, junior. Because often your character and attitude decide whether it's worth investing in you and hiring you.

So really take this advice to heart.

And what? That's about it.

And since we're on appeals...

22.2 Want to, Write Good Code

A bit of an absurd heading, right?

Yes, but not entirely. There's one very important thing I'd like to tell you, preferably several times, so you remember it.

The thing is...

Some people, after some time, give up. They stop wanting to, they stop trying. They just have enough. They no longer strive to be better, for the code they write to be better day by day. For each change to make the repo in a better state than before.

Burnt-out employees.

Never be like that. Please, this is my personal appeal.

Try to be a person who develops every day, who WANTS to.

And that's it.

22.3 Disgusting Curiosities in the IT World

Recently I've been observing a disgusting practice. The practice of politicizing the IT world and software development as a whole. What do I mean?

The regular process of destroying our community with political correctness.

For a long time I've been watching this process, but recently when I saw that branches of pull requests were merged into Python's official repository, whose only task was...

Changing the terminology master/slave in the context of processes and so on. As if that wasn't enough, quite recently in the Linux kernel development group a mandatory CoC - Code of Conduct appeared, which sounds... Well, quite specific.

Linus himself, who is known for his 'characteristic' behavior, and for being able to tell everyone what he thinks, often quite sharply, announced that for now he's taking a break to 'work on his character in light of certain changes'. Oh dear people.

These two events made me decide to write my two cents on this topic. I find this highly unpleasant. I consider it completely disgusting, repulsive and unacceptable to turn the IT development process and our society into some politically correct abomination that in no way resembles the beauty of its original form. What exactly do I have a problem with?

Here's a list of a few things.

Let's start with the discussed example of changing Python's terminology. This is not an isolated case. A similar situation already happened a few years ago, but in the context of the Django framework's code. As we can see, this is starting to become a trend.

Highly worrying and stupid, frankly. As we probably all know, in the IT world we have a certain specific dictionary of terms that have more precisely defined, known to all, meanings. Seeing certain words, regardless of the language they appear in, we can to a considerable extent guess what a given function, variable or generally, given code does.

These terms existed for many years, the whole nomenclature is based on them and uses them, often quite accurately describing the abstraction they represent. And now a problem appears.

Because someone comes along who in the name of political correctness starts changing these terms, because they might offend someone, despite the fact that they have no right to, as they're used in other contexts, other meanings, which are also used in dictionaries and frankly no one cares about this, except for a few loud screamers who have gone crazy about certain issues. Unfortunately, from what I see, in the IT world there are rather more somewhat passive people, not very dominant. The introvert type, intellectual, who wants peace.

Unless someone enters their comfort zone so much and disrupts it, such a person won't oppose something much. That's the problem. Because certain circles see this and use this fact to their advantage, to push their agenda everywhere.

This is sad. For me this is complete stupidity that only introduces unnecessary confusion. What if such dilemmas start in all languages?

What if suddenly several different replacement names start functioning for 'uncomfortable' terms, because some will be better than others? Less/more politically correct? This is a recipe for disaster. It's like describing one object with several different names in code. This is not good practice, it leads to bad code and problems with maintenance, debugging and everything basically. That's how I see the effects of these changes.

It certainly won't help here that Guido is starting to withdraw from the Python world, gave up the reins, and recently has had quite some issues - I mean for example his recent declaration, when he stated that he won't teach, consulting, white men, because they're so awful. Instead he intends to help only oppressed minorities.

Oh dear people, please...

Next to discuss is the creation of regulations for the Linux kernel development community. The newly created regulations sound like babble straight from the wet dream of a militant LGBT activist.

Let me be clear - I'm for freedom. I don't care at all who you are, what you do, who you love and who you like. Live and let live. Simple.

In principle these newly created rules sound okay, the problem however is in how they will be implemented, because history has already shown how this usually happens. Something that is supposed to be a tool of freedom for all becomes an apparatus of oppression.

It's simply that SJWs (Social Justice Warriors) gain here a powerful tool that will enable them to completely get rid of people who don't suit their taste. And what doesn't suit their taste is rather the majority than the minority. In principle the rule 'who is not with us is against us' prevails there. Do you see this discrepancy? This dissonance?

In theory they are fighting for social justice, but in practice they are often tyrants who destroy people. Often good people, competent people. If they managed to influence even Linus, who is rather stubborn and controversial in his views, then what will happen to other people? What when some really decent developer 'offends' someone or falls out of favor with the SJW caste?

Do we really need committees of people anywhere who will tell us how to behave, so as not to offend someone? Seriously?

Get this crap out of here, get out of our IT world. Go back to your dumps at universities and other places where you sow this plague. Live there with your political correctness, not with us, in the IT world. Get lost!

Don't ruin something beautiful, because the IT world is like that. Beautiful, inclusive and open, in itself. Look how it looked from the beginning of creation - random people from the Internet, regardless of anything, age, origin, skin color, religion, worked together on various projects, created software that solved the next problems of this world, the smaller and the bigger ones.

Why make a mess in this, interfere and impose some frameworks? Let's move on.

Let's talk about so-called 'positive discrimination', or whatever those idiots call it now. What's it about? Well, if you're a white, heterosexual man, God forbid a Christian, then your life needs to be made difficult somehow, because you're privileged, 'check your privilege!'. Do you understand this hypocrisy?

Next point - quotas in IT. More and more companies are starting to recruit for positions only women/blacks/gays/whatever, to meet 'diversity' standards.

You might think that this doesn't concern Poland, that these are some American problems. Nothing could be more wrong. Imagine that such things also happen in our backyard.

Other, even more terrifying things too - like a sponsor withdrawing funding from a given event because 'diversity' standards weren't met, meaning at a free event, which everyone attends of their own free will, too few women signed up. Why didn't you force them to come, what is this?! Psychosis. And this is a real example from Poland.

How many stories have I heard about a woman being hired for a given position, even though she was less competent than other candidates? I'm not saying women are less competent. Absolutely not. Simply comparing two candidates, one has better skills, will be a better employee, the other won't. It's logical to hire the better person.

When we make this logic a prostitute and hire someone weaker just because they're a woman, or they're black/yellow/green, then something is wrong. St. Thomas is crying somewhere quietly.

Such things, such movements in open source, in companies, in the community, will ultimately lead to worse code quality, departure of pillars and original propagators of open source movements. I don't see a bright future if we don't oppose this plague. Let's just look at one thing. At knowledge, at what someone says and how true it is. At nothing else. Only that should count and that's it.

And I'm aware that by releasing these words into the ether, saying this publicly, I may harm myself, becoming a target for some.

You know what? I don't give a damn. I don't intend to sit quietly and just nod. I've always had a big mouth and said what I thought. I don't intend to change that.

If you're reading this text and you're outraged, then you know what... Tough. That's what freedom of speech is about, that everyone can say what they think, even if you don't like what they say. Deal with it.

22.4 Let's Respect Readers, Let's Respect the Internet

Before you go further, this is a purely opinion piece, zero technical things in it, just my complaining about a certain topic and my perspective. If you're here only for technical topics, you can skip this post, because it's a subjective approach to a certain matter.

What am I talking about? Look at how a large part of websites, especially Polish ones, look today. Without AdBlock they're impossible to browse, because simply most of their surface area consists of ads. Some in video form, others - a regular image. Automatically playing ads with sound. That's a real nightmare. Just like how much bandwidth this whole crap can eat up.

A million cookies and spying on every step of what you do, just to get additional information about you that can be sold.

Pushy newsletter consents, cookies, desktop notifications, GDPR, whatever, popups like: turn off adblock you bastard because we don't have money.

Adding Facebook, Instagram, Snap or whatever else plugins everywhere.

Uncompressed images weighing several megabytes. Tons of unnecessary JS that makes the page load at a snail's pace.

For me this is simply a terrible lack of respect for the reader. As soon as I encounter such a site, if I don't have to, I don't use it. That's why I try to run my site in a way opposite to those described above.

On my blog there are no integrated plugins for Facebook, Instagram or whatever else, there are no comments, because they're not needed in my case. Graphics in news items either, although something might be found somewhere, but in fact the only image here is my face, which, in my opinion, should be here - I write various things, sometimes smart, sometimes less smart, but I'm not ashamed to sign them with my name, surname and image.

In return for this I'm willing to give up anonymity and sympathy for privacy - so my reader knows who writes what they read. Besides, the blog is also for me a possibility to build a professional image, so an additional plus.

There are also no ads here, and never will be. There are only links to my LinkedIn or GitLab profiles. That's it.

My entire site weighs 25.3 kB, and to download it, 4 requests are enough. Loading it takes about a second. Everything served over https. In today's times this is necessary. And yes, I know someone will tell me that this isn't 2005, where every kB counts, but that doesn't matter. Why should I, as a user, waste time waiting for your super modern SPA landing to load after 10 seconds? No way.

The global average internet connection speed is currently about 70-80 mbps. Despite this, pages still load at the same snail's pace. What pace? Usually it's from 2 to 10 seconds with a median of 6 seconds to completely load the page and 4 seconds to render some meaningful content. THAT'S A LOT. For comparison, my entire site loads in ~1 second.

Everything is cached via Cloudflare, so CDN is running and speeding things up. How much?

You also won't find Google Analytics here, or any other contraption to track how many people read this blog and what you do here. It doesn't matter to me - in fact it doesn't matter at all in the case of personal blogs. The only cookie you get from me is probably the one from Cloudflare.

I can tolerate sponsored articles or content, cool, some product placement. Normal thing, you have to earn. There are various forms of monetizing reach or content that can be used, which will make both the creator earn and the reader get something out of it. And that's what it's about, because when it's different, it's simply nothing else but a lack of respect for the reader and that's it. I don't respect such things.

So what exactly is this about? It's about you sometimes thinking, as a creator, whether you really need that two megabyte image as your blog background. Whether buttons for sharing on Facebook, Instagram, Snap or whatever else must be everywhere. Whether you must attack the user with ads, newsletters and other nonsense.

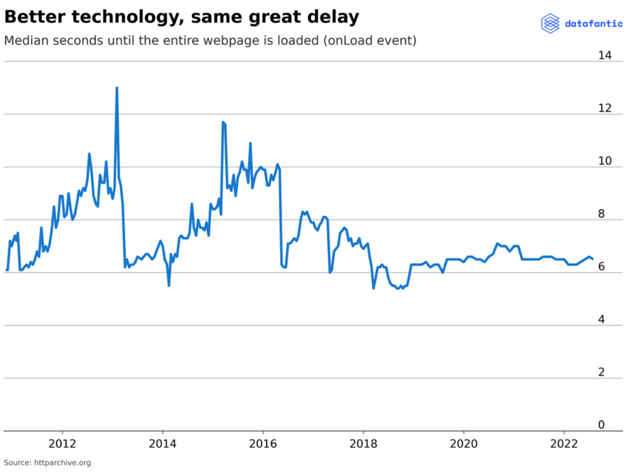


Figure 5: Average speed of complete page load

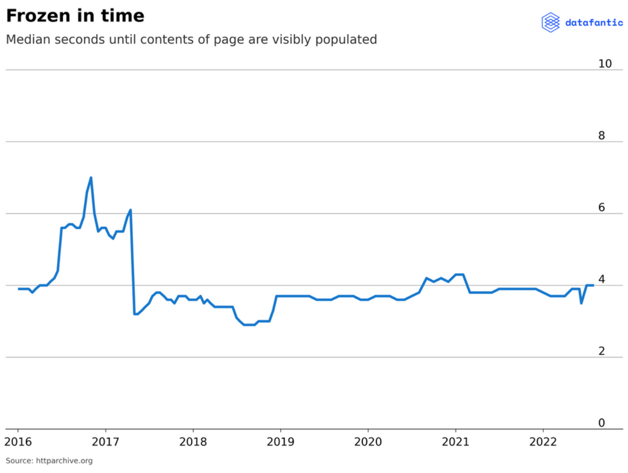


Figure 6: Average speed to render content

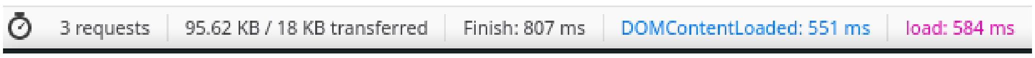
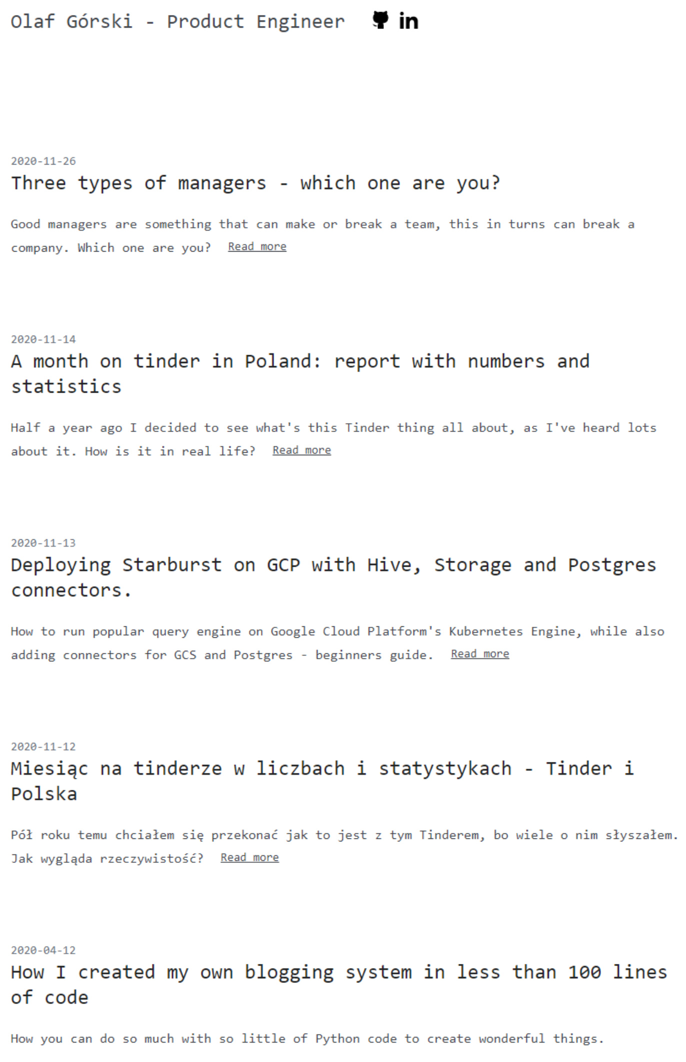

3 requests | 95.62 KB / 18 KB transferred | Finish: 807 ms | DOMContentLoaded: 551 ms | load: 584 ms

Figure 7: Meanwhile at grski's place. . .



Olaf Górski - Product Engineer

2020-11-26
Three types of managers - which one are you?

Good managers are something that can make or break a team, this in turns can break a company. Which one are you? Read more

2020-11-14
A month on tinder in Poland: report with numbers and statistics

Half a year ago I decided to see what's this Tinder thing all about, as I've heard lots about it. How is it in real life? Read more

2020-11-13
Deploying Starburst on GCP with Hive, Storage and Postgres connectors.

How to run popular query engine on Google Cloud Platform's Kubernetes Engine, while also adding connectors for GCS and Postgres - beginners guide. Read more

2020-11-12
Miesiąc na tinderze w liczbach i statystykach - Tinder i Polska

Pół roku temu chciałem się przekonać jak to jest z tym Tinderem, bo wiele o nim słyszałem. Jak wygląda rzeczywistość? Read more

2020-04-12
How I created my own blogging system in less than 100 lines of code

How you can do so much with so little of Python code to create wonderful things.

Figure 8: 2022 and a site without JS. Possible? Yes.

This is my appeal. Let's take care of the Internet, how it looks and how it presents itself. It's our common good.

To read, source of charts: https://www.datafantic.com/how-much-time-do-we-waste-waiting-for-websites-to-load/

22.5 First Days as a Programmer

A few words about how the first days looked, at least for me.

Let me start by saying that before my work started, I had a few concerns. The first was probably that I wouldn't manage or that the responsibilities would overwhelm me. Natural, though unjustified. Why?

It's the company's and its recruitment process's task to check if you're suitable. If you got the job, and you didn't lie in your CV and despite this you don't manage, then it's the company's fault, not yours. Simple, the recruitment procedure failed here, not you. And as for regular stress, it's probably normal, no need to worry. I'm of the opinion that there are people everywhere, and you can get along with people.

Okay, so how was it?

Well, contrary to appearances... Honestly, it was calm.

22.5.1 Calm First Days

The first days at my work I actually spent installing things I needed, waiting for access to appropriate services, setting up the environment, getting to know the company structure and how we work, my coworkers. Generally if it's about programming then practically nothing. Really, first days are calm.

22.5.2 Onboarding

Only when I had everything set up did I start programming something. At the very beginning my Project Leader, to kind of onboard me into the technologies we use, assigned me increasingly difficult tasks to do. Very nice thing. Thanks to this later, when I started working on the project itself, certain things were familiar, this allowed me to feel more confident and calmer, this is helpful, especially when you're starting in a new environment, learning new things and want to make a good impression.

I won't elaborate on technologies here, because they're different for everyone - depends on what position, what project and so on. However, I'll write below about certain things that are probably universal. In any case.

22.5.3 Important Matter

Here it also shows how important a good Project Leader or superior/mentor is. I happened to get a good one, thanks to which really, in my opinion, my onboarding into the new reality and work went quite pleasantly. Of course, there was some stress accompanying me, because I cared about this job, but despite everything I felt good.

22.5.4 Atmosphere

The atmosphere and way of communication in the company was somewhat a surprise to me. At the beginning I had an impression that we would all be somewhat 'formal' at first - Mr/Ms, formal speech and so on, but... Nothing could be more wrong. From the very beginning a rather loose atmosphere surrounded me. I don't know how it looks in other companies, as I work in a rather medium-sized company, where basically you know most people anyway and that's how it looks.

No one gets angry here either. People are simply... helpful. This was a small shock to me, because earlier I worked a bit in construction, and there people can be, well, different, but not here. Therefore also

22.5.5 Don't Be Afraid to Ask for Help

At the very beginning I had an impression, before I started, that I would be somewhat left to myself - you just get a task and that's it, do it. Can't do it? Get out.

Well, after starting work it turned out that no - nothing could be more wrong. First of all, there are people with more experience and often they're happy to help, share knowledge. Heck, more often than not they're paid for it. You can ask for help.

Of course, you can't run around and ask about every little thing, but when there's really no other way, we can't find a solution ourselves and we're wasting time, it turns out there are people willing and even assigned to help.

So don't be afraid. Ask. No one will bite your head off for not knowing something, not understanding something. Calm down. If you go and say 'Hey, can you help me with X? I've already tried this and that, but somehow it's not working.' then there probably won't be a problem, but if you run to your superior or another team member with every little thing and expect a ready solution, then... Well, it will be irritating for sure.

Just remember to somehow write down the solutions to these problems. It's silly to ask about the same thing several times. Making notes is a great thing. Sometimes it's also enough to just browse through message history.

22.5.6 It's Obvious You Won't Know Some Things

As juniors or generally young programmers, we rather have a certain distance from the boss as a gift. I mean that it's rather expected that we won't know everything, sometimes there will be some mistake, or in some topic you'll need to be educated. It seems to me that any serious employer takes this into account when hiring a novice/junior.

This is also a reason why you should adjust your requirements. It's known that you need to have some respect for yourself and earn something, to work for a proverbial two zloty, you can't, but it's also a mistake to think that you'll learn to write Hello World in JS and someone will then give you 15k netto just for breathing. No, that's not how it works.

In any case.

Therefore, if you don't know something, you did something wrong, don't try to clumsily mask your mistakes or lack of knowledge, just be honest and admit your mistakes.

22.5.7 Honesty

I wanted to say something more about honesty. Never lie. Simply. You'll lie in your CV? It will come out in the interview. Then the word spreads, IT is a small circle despite appearances, and you'll be burned. You broke something, lied and covered it up? It will come out too, sooner or later, usually sooner. You don't know something, but officially this is your task and it's almost done? It won't work, because ultimately you won't finish it and only make everything unnecessarily prolonged.

It seems to me that a large part of responsibilities in the first job is simply being honest and responsible for your actions, nothing more.

22.5.8 Universal Things

I'd like to mention here certain things that you can check out even before coming to work, because 90% they'll be used in your company too. Usually these are simple things that you understand immediately, but the more familiar interfaces, the more confident you'll feel and easier it will be to handle this whole flood of information, so I'll mention a few technologies that are worth getting to know somewhere. Heck, maybe you'll even score some points with this in the interview ;)

22.5.8.1 Jira/Confluence From what I know, Jira is a fairly popular task management system. How it looks and what's worth knowing about it, you can easily find on the internet. Generally rocket science isn't here - it's just a dashboard for easier managing who has what to do. Confluence is a kind of knowledge base/documentation.

22.5.8.2 Git/VCS Some VCS is used in every company, you should already know that. If you ended up somewhere where such things aren't really used, and projects are passed e.g. in archives or sent by email, then... Well something's wrong. Generally VCS is a necessity and standard.

Currently Git is such a standard VCS. I think most use it. Others happen, but it's rather rare.

For Git there are various 'services' that provide repository maintenance services, sometimes companies do it themselves, on their servers. In any case you'll most likely use a repo on GitHub, GitLab or BitBucket. You can browse how these services look and check what and how.

By the way, if you don't have an account on GitHub or somewhere yet, with your projects and code, I recommend you quickly get one. Similarly, if you don't know Git. You should know the basics.

22.5.8.3 Slack/Google Meets/Microsoft Teams/Zoom Standard communication tools. In addition there's also company email, which you'll probably get, so before you can set up Thunderbird or another email client, but that's also rather for the chosen ones. The standard is rather that you simply get a new Gmail or Outlook.

22.5.9 Summary

That's how my first days at work looked. Does it look like this in every case? I don't know, but probably similarly. Things I described here may seem obvious to some, but... Well, not to everyone.

To watch: https://www.youtube.com/shorts/0P1RuLgxPNE

22.6 Nootropics - Brain Doping

Sometimes it happens that the world is against us and we have to be productive for certain reasons, not necessarily when we want to. Everyone makes decisions for themselves, we're adults. There are certain substances and ways to boost our productivity and efficiency at work. I'll describe two of them here that are worth attention and consideration in crisis situations.

The first drug I'll describe is modafinil. What is modafinil? Remember NZT from the movie Limitless with Bradley Cooper? Something like that, but without side effects. Officially it's a drug for narcolepsy that enhances alertness and concentration, preventing sleep. In short - one tablet, you sit down and for 15h you lock yourself in a room and work. Modafinil is a drug and doesn't show euphoric properties like for example cocaine, which supposedly has similar effects. It stimulates the brain and neuroplasticity, arouses concentration, gives energy to act. 200 to 400mg usually.

I won't go into details about how this drug works, how it's metabolized etc., I'll just write from experience that it works. You sit down, you do a million times more work. Just like that. I don't recommend combining with caffeine because it might be too much.

In Poland officially it's medium hard to get in pharmacies, unofficially you can buy it for decent money. Attention: I'm not encouraging consumption here. I'm just mentioning that such a thing exists, it works on me, and maybe somewhat explaining how it's possible that some managers or presidents work 16h a day and still have energy. :-)

There are other drugs similar to modafinil, like adderal for example, but adderal is actually an amphetamine salt. Modafinil isn't. Modafinil king of nootropics is like a lion, like the king of the jungle.

Methylphenidate - another creature to read about.

Personally I sometimes take modafinil when I need to sit down and work very productively or learn something difficult like now, when I'm writing these words.

23 Epilogue

Well, that's about all I had to say.

Surprisingly, quite a bit has accumulated. Wow, over 200 pages. Not bad.

A touching moment.

23.1 What Now?

In addition to reading this book, simultaneously or even before reading it, I recommend familiarizing yourself with the following resources:

1. https://realpython.com/ <- Here you'll find excellent summaries of various topics, whether it's list comprehensions, types, printing, or loops. Look for everything there and use it to read about details I haven't covered here.
2. Learning Python, 5th edition - Mark Lutz <- Definitely browse through. You can read chapters selectively, I haven't gone through the whole thing myself, skipping chapters that didn't interest me, but this tome is a damn encyclopedia. Highly recommended. Unlike Real Python, where content is somewhat simplified, here things are described step by step and very extensively. I believe this is an ideal foundation for further development and should be read in parallel or even before my book.
3. CS50 - probably the best course on Computer Science basics. In parallel with this book and Learning Python.
4. Effective Python, 2nd edition - Brett Slakin <- An excellent book to work through after you've read the current book and Learning Python. You'll significantly polish your Python skills.
5. Fluent Python - Luciano Ramalho <- A book similar to Effective Python. Optional.
6. Designing Data Intensive Applications - Martin Kleppman <- Once you finish this book, CS50, Learning Python (part), and Effective Python, it's good to broaden your horizons and look here. An excellent book about designing not just code but entire systems.
7. Understanding Software - Max Kanat-Alexander <- Another thing to read.
8. Zalando API Guidelines <- at least skim through
9. PyCoder's Weekly / Python Weekly / UnKnown news <- Nice newsletters that send you a collection of interesting links/articles and tools from the Python world and beyond every week.
10. python-awesome, django-awesome, fastapi-awesome <- GitHub repos that collect lists of popular, proven, and interesting projects that will often make your life easier.
11. Sebastian Opałczyński <- I recommend his work and various projects, he also creates educational content. Courses, training, etc. Kilograms of pure knowledge. This is a person who shaped me to a large extent and pulled me up to where I am. Highly recommended.
12. Remote - Jason Fried <- about remote work, which today is an indispensable element.
13. Blog: The Pragmatic Engineer

Resources to broaden your horizons, which can be read a bit later.

1. Understanding Programming - Gynvael Coldwind <- if you think you've just scratched the surface of low-level basics in this book, read this. An exceptionally valuable book, but I'd mark it as optional because it's not easy and might discourage you. However, it's worth working through this volume. I also recommend Gynvael's other work, a very inspiring and well-known figure in the scene.
2. The Pragmatic Programmer - Thomas Hunt <- what can I say. A classic.
3. Web Application Security - Securitum <- An excellent book about security in the context of web applications from world-class experts. By the way, an interesting fact - Poland has one of the stronger representations when it comes to security. Poland stonks.
4. Modern Software Engineering - David Farley <- About modern ways of software development.
5. Extreme Programming - Kent Beck <- A bit about how the software delivery process looks and ways of collaboration.
6. Deep Work - Cal Newport <- about working with focus.

As for practice, stay tuned - I should soon release a collection of exercises worth working through to gain some practice. Check the blog/LinkedIn.

I believe that if you diligently read and work through the materials above, you'll gain solid knowledge allowing you to start applying for jobs. However, it's required that you actually perform tasks independently, read with understanding, create analogies and comparisons, and be able to speak on a given topic. Additionally, I'll note that I know some tasks may seem trivial now, but still, do them. The point is to get you as accustomed as possible to writing code and familiar with Python's standard library.

23.2 This is the End, There is Nothing More

Yup. Thank you for reading and good luck. To those who purchased this book, I offer a small bonus. If you need advice on something, you can reach out to me. I'll try to help.

Email: olafgorski@pm.me 4programmers: @grski wykop: @grskii (two i's) hejto: @grski github: @grski

Feel free to reach out with questions about your career or perhaps for a mock interview. I can't promise I'll help everyone, but I'll try my best. I'll also honestly assess your knowledge and provide feedback if needed.

I might try to recommend the more capable ones somewhere. Who knows.

Remember, someone helped me once too, and I want to pass the baton forward. This means I'm just a human, not some unknown entity. If you write to me, it's okay, I don't bite. I'll try to help as best as I can.

Don't forget about https://github.com/grski/junior-python-exercises where by making pull requests to the repository you can get additional tips plus it will always be something that enriches your GitHub with contributions ;)

Thank you for reading.

That's all from me, have a nice day and goodbye.