Programowanie z Górskim: Junior Python Developer

Czyli co warto wiedzieć jako nieopierzony junior.

Czyli Tech Lead bierze się za uczenie innych Pythona, podstawowych umiejętności przydatnych w karierze i nieco opowiada o sobie.

Pomoc przy korekcie: Weronika Sosnowska
ISBN: 978-83-967638-0-8 (wersja elektroniczna)
Olaf Górski/OnionMindset, Kozienice 2023

Contents

1 Wstęp

Pompatycznie i z przytupem. Poza tym notka: jeśli nie chcesz wdawać się w szczegóły historii Pythona, nie interesuje Cię kim jest autor, po co ta książka a chcesz po prostu od razu wskoczyć do akcji, to streszczę ci następne ~50 stron: Python jest fajny, zainstaluj sobie SublimeTexta/Vima/Pycharma/VisualStudio Code (na początku cokolwiek, serio) i przeskocz do rozdziału 4.

1.1 O co tutaj chodzi?

Ta książka to w zasadzie nic innego jak kolejne dzieło o Pythonie, ale nie tylko. To konsolidacja całej mojej wiedzy technicznej jak i tej miękkiej, jaką posiadałem, startując na pozycję Junior Developera, czy też nawet teraz, troszkę dalej, kiedy niekoniecznie już widzę siebie, jako nieopierzonego juniora. Poruszę tu różne kwestie, od samego procesu rekrutacyjnego, przez umowy, umiejętności miękkie, opis mojego rozwoju, przydatne rzeczy, jak i samo programistyczne mięsko – proporcja będzie plus minus pół na pół. Oprócz samego tekstu, będę tutaj również podawał referencje i materiały do samodzielnego studiowania.

Nie uważam, by była ona w jakimś dużym stopniu odkrywcza czy innowacyjna, ale z drugiej strony, to trochę tak. Dlaczego? Co wyróżnia tę książkę? To, że podczas jej pisania przyświecał mi jeden cel – by była ona jak najprostsza do zrozumienia, a zarazem dogłębnie analizowała poruszane tematy w ciekawy sposób oraz fakt, że to nie będzie to książka jedynie o Pythonie, nie.

To ma być kompletne wprowadzenie to szerokiej gamy zagadnień związanych z programowaniem i pracą programisty, jej zdobyciem, narzędziami, których się w niej używa, ale również nauka dla ciebie odnośnie najważniejszej umiejętności dla programisty - umiejętności wyszukiwania i przetwarzania informacji, bo to na tym polega informatyka.

W skrócie to taki wstęp do kompletnego zestawu umiejętności, cech, narzędzi, których opanowanie sprawi, że będziesz w jakimś stopniu gotów, albo gotowa, na zaczęcie przygody jako programista. Nie myśl sobie jednak, że przeczytasz jedną książkę i cyk do pracy. Oj nie. To pierwszy krok na bardzo długiej drodze.

Nie chcę, by był to kolejny zwykły podręcznik do Pythona traktujący tylko o nim, ich jest już wiele. Chcę byś Ty, mój czytelnik, moja czytelniczka, po przebrnięciu tej książki, wiedział nieco więcej niż wcześniej, miał nakreślony jakiś, chociaż ogólny, obraz tego, jak wygląda praca programisty, jak wygląda ten świat, czy warto w niego wejść, jakie są wady i zalety tej pracy, co trzeba konkretnie umieć, skąd się tego uczyć i tak dalej.

Moim marzeniem jest, byś po przeczytaniu tej książki był niczym ciekawskie dziecko, któremu pokazano nową zabawkę – pełen zainteresowania i postanowienia, by się tym pobawić. Czy to przez chwilę, czy też może na dłużej, bez znaczenia.

Mam po prostu nadzieję, że wpłynie ta książka w jakiś dobry sposób na twoją wiedzę, czy też może życie, pozwalając na podjęcie na przykład pierwszego kroku w zmianie kariery bądź jej zaczęciu. Po prostu, że ci jakoś pomogę. Taki jest jej cel. Wywodzi się on z tego, że moim największym marzeniem, personalnym sensem życia, jest wpłynąć na jak największą ilość osób, wnieść coś pozytywnego do ich życia.

Mnie też wiele osób pomogło, więc chcę przekazać pałeczkę dalej, to mój zamiar.

Mam nadzieję, że jakoś uda mi się go zrealizować. Jeśli tak, to pamiętaj, z chęcią posłucham twej opowieści, czy też nawet opublikuję ją na blogu – grski.pl – może dzięki niej to ty wtedy pomożesz kolejnym osobom? Zatem jeśli ta książka w jakiś sposób poszerzy twą wiedzę, czy da ci coś pożytecznego, napisz do mnie!

olafgorski@pm.me, @grskii na wykopie, grski na 4programmers, grski na githubie

Kiedy zajmowałem się udzielaniem korepetycji, było to długo, zanim zostałem programistą, nauczyłem się tego, że po pierwsze, ucząc innych, uczysz i siebie, dwa, suche fakty nie zawsze wystarczą, żeby ktoś zrozumiał dane zagadnienie. Potrzebne są ciekawe przykłady, nietypowe analogie i ćwiczenia. Tak też postaram się przekazywać wiedzę w tej książce, prosto, ale i ciekawie.

Kieruję ją zatem do osób raczej początkujących, którzy chcą powoli i dogłębnie zrozumieć pewne pojęcia związane z samym Pythonem, jak i ogólnie pojętą informatyką, zyskać trochę informatycznej ogłady, ale uważam, że ktoś doświadczony również znajdzie tu jakieś smaczki, które mogą zainteresować. Książka będzie raczej pełna uproszczeń, które poczynię w celu łatwiejszego jej zrozumienia, zatem za wszystkie nieścisłości z góry przepraszam, ale tak trzeba. Czasami czytając polskie książki naukowe, techniczne, wydaje mi się, że ich autorzy boją się, że jeśli napiszą coś prosto, zrozumiale, to ludzie uznają ich za głupców, czy coś. W efekcie dostajemy teksty, które potem sprawiają, że człowiek zastanawia się, jak głęboko autor takiego dzieła, nadział się na kija siadając.

Ta książka to też swego rodzaju podsumowanie, czy też może małe streszczenie tego, co warto wiedzieć, kiedy startujemy np. do pierwszej pracy. Zatem jeśli szukasz krótkiego przewodnika po kluczowych funkcjach Pythona, to cóż, raczej nie jest ona dla Ciebie. Jeśli jesteś początkującym programistą i chcesz poszerzyć swoje horyzonty lub poznać szerzej podstawy Pythona, to zapraszam dalej. Postaram się umieścić tu odpowiedzi na wszystkie pytania, jakie ja miałem na początku swej drogi.

Ten tekst to też efekt tego, że zadałem sobie jedno pytanie. Jakie? Wiesz, o co chodzi? Padło ono właśnie w kontekście mojej znajomości Pythona, jak i programowania ogółem, i mimo tego, że pracowałem wtedy już jako programista w tymże języku – byłem niby pełnoprawnym juniorem, to niestety, musiałem odpowiedzieć... Nie.

Podejrzewam, że w gruncie rzeczy sporo jest takich osób, mimo wszystko. Często sami nie zdajemy sobie sprawy, jak płytka jest nasza wiedza, mimo tego, że w pracy jakoś sobie radzimy.

Zwłaszcza jeśli weźmiemy pod uwagę języki stare, dojrzałe i rozwinięte, takie jak Python, Java czy C++. O tym ostatnim nawet jego twórca, boi się mówić, że zna go lepiej niż w stopniu 8/10. O czymś to świadczy. W każdym razie. Po tym, jak odpowiedziałem sobie na to pytanie i zdałem sprawę, że moje rozumienie Pythona jest płytkie, postanowiłem to zmienić, żeby wejść na następny poziom.

Jest bowiem we mnie i zawsze było, pragnienie rozwoju, które przez jakiś czas zaniedbałem, co skończyło się stagnacją a wręcz regresem. Dalej być tak nie mogło, więc to naprawiłem.

A jak już to zrobiłem, to postanowiłem spisać swoje przemyślenia, wiedzę i doświadczenia właśnie tu, ale to nie jedyny powód. Żeby napisać tę książkę, sam będę musiał się w sporej ilości tematów dokształcić, by przedstawić wszystko rzetelnie i sensownie, gdyż wstyd byłoby mi dawać Ci, drogi czytelniku, bubel zrobiony na odwal się, byle coś wydać.

W tej chwili nie wiem, ile ta książka ostatecznie będzie liczyła stron, ale pewien jestem

jednego, że raczej krótka to ona nie będzie. Mentalnie mam wstępny obraz, który mówi mi, że poniżej 250 stron, to nie zejdę. Zwłaszcza biorąc pod uwagę fakt, że jak już zacznę gadać, to czasami ciężko skończyć, bo jeden temat wiedzie do następnego i tak w kółko.

Ostatnia jeszcze, swego rodzaju, uwaga – mój styl pisania jest specyficzny. Jak widać. Tak po prostu. Mam swoją manierę, wielokrotnie było mi to wypominane przez różne osoby i jestem tego świadom, niemniej jednak uważam to za część specyfiki moich prac.

Nie zamierzam tego zatem zmieniać. Podobnie jak tego, że w tej książce mogą wystąpić zarówno słowa dziwne i górnolotne, jak i te proste, kolokwialne, potoczne. Taki już po prostu jestem. Jeśli potoczne słownictwo, infantylny styl cię przeraża to radzę szybko odłożyć tę książkę. Nie mówię tego z pozycji aroganckiej a po prostu dobrze doradzając.

Czasami zdarzą się jakieś nieciekawe żarty, czarny humor i personalne uwagi, opis moich doświadczeń, niekoniecznie politycznie poprawnych.

Chyba tyle, jeśli chodzi o moją krótką przedmowę, czy coś w tym stylu. W sumie to nie do końca wiem, jak to wszystko powinno wyglądać – jestem prosty człowiekiem, który zwyczajnie chce przekazać ci jakiś fragment, być może, przydatnej wiedzy. Tylko tyle i aż tyle.

1.2 Czyli w skrócie

. . . mówiąc, to ta książka będzie też niejako przewodnikiem i odnośnikiem do innych materiałów, które samodzielnie trzeba przerobić. Głębiej i wnikliwiej omawiałem tylko, moim zdaniem, te tematy, które często są zaniedbywane.

1.3 Podziękowania

Na sam początek mojej książki, zaserwuję sobie, a przy okazji i wam odrobinę pokory – mogłoby się wydawać, że te słowa, bo to przecież książka, a książki tworzą często ludzie poważni, pisze jakiś człowiek sukcesu, wielki programista, mądraliński.

Nic bardziej mylnego, sam w sobie jestem nikłym i prostym człowieczkiem, który zaszedł tu, gdzie jest, w dużej mierze dzięki pomocy wielu osób, część z nich zostanie właśnie wymieniona niżej i to wszystkim im chciałbym podziękować.

Jeśli Cię to nie interesuje, tak samo, jak przedziwna dedykacja, która się tu znalazła, przeskocz kilka stron.

Do jednych zwracam się bezpośrednio, do innych w trzeciej osobie, różnie. Zupełnie bez powodu. Zatem dziękuję. . .

. . . mojej najukochańszej i jedynej Babci. Za wszystko, za to, jakim człowiekiem jestem. Za miłość. Za wyrzeczenia.

. . . tobie, **Wiktorio Chmielewska**, za dwa najszczęśliwsze wyjazdy w moim życiu, za poświęcenie, za troskę i przyjaźń, za ratowanie mi skóry i nie odpuszczenie, kiedy inni dawno już to zrobili.

. . . wam, **Łukaszu i Batu**. Fajnie, że jesteście. Co ja będę dużo pisał, Bóg jest miłością i piąteczka szefuńciu.

. . . tobie, **Rafale**, wraz z rodziną, której nie będę tu wymieniał ze względu na Waszą prywatność – twoją rodziną, której w pewnym stopniu czuję się już w zasadzie częścią, za otoczenie mnie opieką, naukę wielu rzeczy i realny, wielki wpływ na mnie i moje życie.

Nie mam ojca, ale gdyby ktoś mnie spytał, kto jest mi najbliższym jego obrazem, to za długo bym chyba nie myślał.

... tobie, **Najdroższy**. K R A B O W A N I E.

... **Jakubowi Gąsiorskiemu**, osobie, która odbywała ze mną rozmowę o pierwszą pracę. Uważam, że to właśnie jemu zawdzięczam fakt, że ostatecznie padła decyzja, by mnie przyjąć, bo postanowił dać mi szansę wtedy i później, wiele razy. Tutaj należą się też podziękowania **Maciejowi Mondrzyckiemu**, CEO YumaSoft, bo finalna decyzja, czy warto we mnie zainwestować, należała do niego.

... **Sebastianowi Opałczyńskiemu**, gdyż to osoba, którą uznaję za mojego pierwszego mentora, jakiś wzorzec tego, jak się rozwijać jako programista, oraz osobę, która do powstania tej książki się przyczyniła w dużym stopniu. Wyniosła mnie powyżej tego, co mogłem osiągnąć sam i jemu bardzo dużo zawdzięczam, naprawdę. Dzięki Łysy. Czasami wskazał mi właściwy kierunek w kilku rzeczach, cierpliwie odpowiadał na różne pytania, miło wspominam też różnorakie nasze rozmowy. Jesteś bardzo dobrym nauczycielem i człowiekiem. Zastanawiałeś się kiedyś, czym jest samiec alfa?

... **Piotrkowi Nietrzebce** za bycie ludzkim człowiekiem kiedy trzeba, szczwanym dziadem w innych okazjach, za pokazanie pewnych ułomności i że się da je pokonać, albo przynajmniej można spróbować, za dotrzymanie obietnicy.

... **p. dyr. Pawłowi Boryczce** z Zespołu Szkół w Kozienicach, za to, że przez 3 lata sprawował nade mną pieczę i wiele mi w życiu pomógł. Dzięki temu udało mi się rozwinąć pewne umiejętności, poznać różnych ludzi, których prawdopodobnie nie odkryłbym w innym wypadku. Będę o tym pamiętał.

... tobie, **Emilio**. Sama wiesz za co. Eryk pozdrawia Blankę z zaświatów, odszedł już w zapomnienie.

... tobie, **Falo**, ty stary ch**u kochany. Za to, żeś jest ciepłą kochaną kluseczką, która sobie odejmie i nie będzie miał, ale da komuś. Jesteś cudownym człowiekiem, nie zmieniaj się, jeśli stare wygi sam wiesz skąd, mówią, że "jesteś za dobry na to miejsce", to wiedz, że coś się dzieje.

... wam, **Łukaszu i Olu**. Kocham jak rodzinę, przesyłam ciepłe myśli wam i dzieciakom.

... tobie, **Alino**. Dużo mnie nauczyłaś, dużo mi dałaś, uczyniłaś moje życie lepszym, tak po prostu. Przeżyłem z tobą cudowne lato.

... tobie, **Kingo**, że jesteś i byłaś.

... tobie, **Zimmermanie**, za okazanie zrozumienia kiedy potrzebowałem.

... tobie **Pawle Poldeczku**, za przygarnięcie mnie pod swój dach w ciężkich chwilach. Pamiętam.

... tobie **Szymonie Szambelanie**, WUJA, DZIĘKUJĘ.

... tobie **Muffi**. Za dobre serce bracie.

... tobie **Robercie**, za pewne dobre pomysły z nieco gorszą egzekucją.

... wam **Kubuni i Dominiczkowi**. Za zdrowe kolana i wszystko.

... tobie **Maniek**, aka Król Podwóra, Jesteś dobrym człowiekiem, dziękuję za wiarę we mnie i pomoc.

... całej załodze, byłej jak i obecnej, **Mojego ulubionego i jedynego słusznego baru**, gdzie część tej książki powstała. Kocham to miejsce.

... tobie **Bartuś**. Zawsze mi pomogłeś jak potrzebowałem.

. . . tobie **Brodziatku** boś równy gość.

. . . moim rozmaitym przyjaciołom, inspiracjom, dobrym ludziom, **Tomaszowi, Andrzejowi**, (grubemu nie, bo mi podnosi ciśnienie i musi iść do psychiatry), **Adamowi, Takurze, Amelii, Dimie**. Dobra, grubemu też.

. . . do not think that I've forgotten about you, my dear brother, **Kunaal**. I thank you and I'm grateful that you've been there for me. Always.

. . . wszystkim tym, którzy mnie czegoś nauczyli lub pomogli mi na mej drodze, a których tu nie wymieniłem, bo szczerze mówiąc, ten tekst byłby zbyt długi, tak po prostu.

Jak sami widzicie, Czytelnicy, miałem szczęście, a w zasadzie chyba więcej szczęścia, niż rozumu ogółem, spotkać na swojej drodze naprawdę sporą liczbę ludzi, którzy mi pomogli.

Oczywiście, samo nic się nie zrobiło i spełnienie swojego marzenia – podjęcie pracy jako programista i realizacja się w zawodzie, wymagały ode mnie dużej ilości pracy i samozaparcia, ale sukces nie zawdzięczam tylko sobie, bynajmniej.

O co mi z tym chodzi? O to, by przypomnieć Ci, jak i sobie, że warto dbać o ludzi, których mamy wokół i którzy nam pomagają, którzy są dla nas ważni i pomocni. Sam tego długo nie wiedziałem, robiąc różne, niekoniecznie dobre, rzeczy.

Dlatego też zamieszczam tutaj te strony, by na zawsze mieć coś, co będzie mi o tym przypominało.

Jeszcze raz. Dziękuję wam ja, Olaf.

1.4 Dedykacja

Tę książkę dedykuję prawdziwemu szczęściu, które znalazłem w życiu tylko na kilka chwil, w kciuku, w windzie z limonkami, na zboczu byłego sadu, przy słupie obok centralnego gdzie narodziło się Małe Zakopane, przy zasadzie 3E. Obym doznał takich momentów więcej.

Przede wszystkim jednak dedykuję ją również wszystkim tym, którzy w moich najciemniejszych i najgorszych momentach w życiu osobistym, pomogli mi. Często niejednokrotnie i ogromnym nakładem sił. Dziękuję, nigdy nie zapomnę. Wiecie, kim jesteście. Zastanawiam się co ja w życiu zrobiłem, że na was trafiłem.

2 Kilka słów do piratów i o prawach autorskich

Arrr! Wszystkie prawa do tej książki i treści w niej zawartych, które stworzyłem, sobie zastrzegam, ale wiem, że różnie to w życiu bywa. Jak jesteś w kiepskiej sytuacji, po prostu nie masz pieniędzy, albo `mame` nie da, czy `tate` nie da, boś od nich zależny/zależna, to nie musisz tej książki pirać.

Napisz do mnie. Podeślę Ci ładnego pdfa za darmo. Też byłem w takim miejscu, wiem o co chodzi. Nie musisz szukać gdzieś po jakiś szemranych stronkach. Podbij na wykopie/4p/blogu albo na mejla a coś wymyślimy bez większego problemu. `olafgorski@pm.me` / `grski.pl` / @grski na 4p

A nawet jeśli pobrałeś tę książkę nielegalnie, to i tak dziękuję, że poświęcasz czas na jej przeczytanie. Nie

3 O autorze

To jak już przez tyle gadania was przeciągnąłem, drodzy Czytelnicy, to może teraz parę słów o mnie, kimże ja jestem by prawić tu jakieś mądrości?

Otóż nikim innym, jak zwykłym Olafem.

3.1 Górski, czyli inteligentny zwierz lądowy

Jestem sobie gościem, co nie ma matury, jeszcze w liceum zaczął pracować na etacie jako programista, za nieduże pieniądze, ale wciąż. Półtora roku później, kiedy niejedno już widziałem oraz właśnie trafiłem do czwartej firmy, osiągnąłem ten niby mityczny poziom 15k, pracując zdalnie dla klienta z zagranicy. Spełnienie marzeń. Podobno. Plus trzeba przyznać, że więcej szczęścia, niż rozumu. Trzy i pół roku później, plus minus, jesteśmy tutaj. Mam około pięć lat stażu. Pracuję jako Tech Lead, jestem przełożonym kilku osób. Miałem za sobą przygodę z prowadzeniem własnej firmy, gdzie w szczytowym momencie zatrudniałem około 8 osób na pełen etat. Nie dziwią mnie już kwoty na kontrakcie typu 40k/msc czy bonus w postaci udziałów za pół miliona. Zabawne to życie. Niedawno 3k to było dużo.

Droga ciekawa, bo przed tym, jeszcze będąc w szkole, gimnazjum i liceum, łapałem się różnych zajęć. A tu trochę wykończeniówki, budowlanki, rozdawania ulotek, spawania, korepetycji, handlu klejem, wszystkiego, naprawdę. Z czego tylko był hajs, że tak powiem. I tu od razu może dla ciebie małe zaskoczenie, bo jak to, były parobek jakieś książki tworzy, programowanie, co? Czemu o tym w ogóle mówię? A bo taki jestem, raczej autentyczny, chcę, by Czytelnicy wiedzieli, kto do nich mówi.

Jakoś w połowie liceum zacząłem robić proste stronki na zlecenie, póki mi osiemnastka nie wybiła. Miesiąc później udało mi się znaleźć normalną pracę w IT. Chodziłem sobie częściowo do szkoły i pracowałem zdalnie jako jakiś tam junior python. Potem tylko szybkie olanie matury w ramach protestu i walki z systemem, kilka zmian firm i pyk, 15 koła, chciało by się powiedzieć, że euro, ale niestety jeszcze nie. 15k euro było, ale jak prowadziłem własną firmę. Teraz już nie, ale o tym może innym razem. Teraz grzeczny ze mnie etatowiec.

3.2 Obecnie

Pracuję sobie jako Lead, kodzę w Pythonie, Django, Architektura, do tego ogarniam coś Jirę czasem, jakieś szczątkowe PO, do tego menadżerowanie czy mentorowanie członków mojego zespołu i w sumie tyle. Bunkrów nie ma, ale też jest zajebiście.

Tak na codzień prowadzę sobie bloga grski.pl, chodzę na siłownię, piszę, żyję, trzeźwieję, zmagam się z demonami. Życie jest w pytę, ale nie jest kolorowe czasami.

Nie wiem, wydaje mi się, że ot, całkiem normalny gość. Czasem trochę krindżu tu i ówdzie, ale hej, taki już jestem.

Z sukcesów w życiu to pod hasłem `tinder statystyki` i `python zalety` jestem dość wysoko w googlu. Czasem nawet pierwsze miejsce. Miałem zawsze pierwsze, ale zapomniałem opłacić domeny i musiało indeksować od nowa xD

Przeszedłem drogę od jełopa, co na własną maturę nie poszedł, przyjeżdżając do Warszawy na krzywy ryj z 400 zł w portfelu, do pana programisty, wciąż jełopa, ale z

zasobniejszą głową przynajmniej, dzięki czemu nie martwię się, czy jutro jest co do gara włożyć, albo nie kalkuluję kalorii przy mleku, które ma lepszy wskaźnik liczby kalorii do ceny, to 3.2% czy to 2%, żeby taniej wychodziło dostarczyć dzienne zapotrzebowanie.

Stąd, wydaje mi się, że jak mnie się udało, to i wielu z was może się udać. Wszystkim raczej nie. Większość pewnie odpadnie po drodze.

3.3 Łatwo to nie będzie

Większość z tych co nie odpadną, będzie co najwyżej średnimi klepaczami kodu. Zostanie garstka zdeterminowanych, którzy osiągną fajne rzeczy. Nie mówię tego, patrząc z góry. Po prostu wiem, jak to wygląda. To nie jest łatwy kawałek chleba. Myślicie, że te półtora roku początkowe, to sobie pracowałem po osiem godzinek, jakiś lajtowy projekcik i do domku? Robiłem tyle nadgodzin ile mogłem, łapiąc doświadczenie od starszych stażem pracowników. Zdarzyło mi się spać w biurze, bo już nie opłacało się wracać do domu. Coś takiego normalne nie jest, ale rezultaty jakie osiągnąłem też nie są normalne. Natomiast ogółem nie polecam, piszę tu tylko o tym nie po to byś brał/brała przykład a po to by uzmysłowić ci, że ponadprzeciętne rezultaty często wymagają ponadprzeciętnych wysiłków. Chociaż z drugiej strony to aż taki wysiłek dla mnie nie był, przyjemność tak trochę. Natomiast wciąż, poświęcić coś trzeba było - czas wolny w tym wypadku.

Często siedziałem i po 12 godzin, w piątki, w soboty, w niedziele. Dzięki losowi trafił mi się idealny mentor, złoty człowiek, bo inaczej to nie wiem. Brodziłem po uszy w gównie, w legacy projektach, których nie życzę nikomu oraz przede wszystkim miałem ogromne szczęście. Nie oczekuj, że będzie u ciebie podobnie. Może być, ale prawdopodobnie nie będzie.

W wolnym czasie staram się pomagać innym ludziom uczyć się kodować, czy to poprzez prowadzenie różnych warsztatów, czy projektów, czy mentoring, zazwyczaj zupełnie za darmo. Z doświadczenia wiem, że większość po prostu rezygnuje. Swoją drogą, jak już mówię o osobach, którym nieco pomogłem postawić kroki na programistycznej ścieżce, to póki co znalazło się około pięciu, które startując od zera, wytrwały i dotarły do pierwszej pracy. Najkrócej zajęło to pół roku niecałe, najdłużej około dwa lata. Wszystko zależy. Stąd też po części pomysł na tę książkę, bo się okazuje, że coś tam jednak mam do przekazania, to po pierwsze, po drugie nie chce mi się tłumaczyć tego samego n-tej osobie. Tak to napiszę raz i będzie z głowy! Super!

Jakiś programistyczny wymiatacz to ze mnie nie jest, ale radzę sobie, raz gorzej, raz lepiej, nie narzekam. Pewnie są lepsi, pewnie są gorsi, nie zagłębiam się w to zbytnio, po prostu robię swoje. Jedni lubią moją działalność czy to na forum, czy na wykopie, czy gdzie indziej, inni nie, uważając mnie za idiotę czy bufona, jednym książka się spodoba, innym nie. Rozumiem, spoko, szanuję, nie ma problemu.

Jednym z nieco nietypowych aspektów mojej działalności jest to, że staram się też poruszać strefę poza samymi umiejętnościami twardymi i mówić o tej miękkiej stronie życia zawodowego, albo życia w ogóle. Czasami, w sumie często, to właśnie te rzeczy, ważyły nad tym, jak potoczyły się moje losy, a nie konkretne umiejętności techniczne.

3.4 Czyli tak reasumując

Na koniec dnia mam tylko jedną intencję, by ta książka, zawarta w niej wiedza, dawka specyficznego i czasem obraźliwego humoru, tendencyjnego i infantylnego stylu, w jakiś sposób komuś pomogła. Być może się uda.

Plus jest to efekt realizacji tego, że powyżej pewnego progu, pieniądze przestają odgrywać aż tak ważną rolę, a na wierzch wychodzą ważniejsze rzeczy. Potrzeba jakiegoś sensu, robienia czegoś wyższego aniżeli kolejnego tysiąca na wypłatę. Dobra, starczy mojego bajdurzenia.

Moją dokładną historię zarobkową czy takie tam szczegóły znajdziesz w innym rozdziale, gdzieś pod koniec książki. Jeśli nie moje słowa i moja osoba, to niech chociaż one przekonają Cię do tego, że być może warto tę książkę przeczytać.

4 Wybór języka

W książce tej będziemy operować praktycznie jedynie na Pythonie. Dlaczego? Powody, dla których wybrałem właśnie ten język, wypisałem w innym rozdziale, chciałbym jednak wspomnieć o jednym, innym, języku, który musi znać każdy programista i od którego należy zacząć, jeśli w 100% jesteśmy pewni, że chcemy podjąć pracę w IT. Co to za język?

4.1 Język niezbędny

Ciekawe co to za język! C, Python? Może C++ albo JavaScript? Bynajmniej! Mowa o... języku angielskim.

Żyjemy w czasach globalizacji. Wszystko się trochę ujednolica. Te same seriale, te same filmy, te same ubrania, marki i... język. Język angielski od dawna obecny był w naszym języku rodzimym — polskim, anglicyzmy istnieją od wielu, wielu lat, jednakże w ostatnich latach, latach istnienia i rozwoju cywilizacji globalnej, ilość anglicyzmów i zapożyczeń, która pojawia się w naszej mowie, zwłaszcza wśród młodzieży, rośnie w tempie wykładniczym. Nie bez powodu. I mówię to ja, gość piszący książkę po polsku. Na upartego dasz radę bez niego, owszem, ale jeśli na poważnie myślisz o karierze... Nie. Serio.

Język angielski jest absolutnym hegemonem, jeśli chodzi o międzynarodową komunikację czy komunikacje w Internecie. My zaś, jako osoby blisko związane z tym światkiem jesteśmy niejako zmuszeni dobrze go znać. Praktycznie w każdej firmie język angielski jest standardem i chcąc być dobrym programistą, musimy go znać na tyle, by swobodnie się w nim komunikować i czytać dokumentacje, tyle na temat. To zaś implikuje, że musimy znać go dobrze. Dokumentacja techniczna zazwyczaj nie jest taka prosta, zawiera specjalistyczne słownictwo, ma trudny, formalny styl.

Język angielski jest w zasadzie na początku chyba nawet ważniejszy niż język programowania, który wybierzemy, dlatego w ogóle, zanim zaczniemy się brać za kodowanie, warto by się doszkolić na ten temat. Jakie są zalety znajomości tego języka?

4.1.1 Materiały

Sprawa ma się dość jasno. Najwięcej materiałów najlepszej jakości dostępnych jest właśnie po angielsku. Ta książka jest próbą lekkiej zmiany tego faktu oraz otworzenia drogi dla tych, którzy mimo wszystko angielskim nie władają, nie chcą władać, a o karierze w IT może nie myślą. Niemniej jednak obecnie jest tak, że...

Chcesz się więcej nauczyć? Najlepiej z angielskich materiałów. Mało tego, czasami angielskie materiały to jedyne dostępne. Prawie każdy znani mi framework czy biblioteka posiada dokumentację. W języku angielskim. W polskim raczej brak albo taką, że tłumaczenie pozostawia wiele do życzenia lub jest ona szczątkowa.

No właśnie, tłumaczenie. Pewne terminy w naszym światku po przetłumaczeniu stają się dość dziwne i niezrozumiałe. Wyobrażacie sobie na co dzień pracować z ZŚP — Zintegrowanym Środowiskiem Programistycznym zamiast z IDE? Odpluskwiacz zamiast debuggera? Popłoch jądra zamiast kernel panic. Dla nas, doświadczonych już programistów, to coś niewyobrażalnego. Nikt, poza własnym krajem, by cię nie rozumiał.

Zamiast pulli i commitów wrepowzięcia (XD) i zatwierdzenia. Po co inity, skoro mogą być wstępniaki. Przykłady można mnożyć.

A dokumentacja to podstawa tak swoją drogą. Czy to przy tworzeniu poważnego projektu, czy przy korzystaniu z niego, więc będziesz ją nie tylko czytał, ale i również tworzył. Jak zachęcisz innych developerów do pracy nad twoim otwarto-źródłowym projektem, skoro nie będą go mogli zrozumieć, gdyż kod źródłowy i dokumentacja będą po polsku? W ten sposób mocno ograniczasz swoje horyzonty.

4.1.2 Komunikacja

Znowu, rzecz kluczowa przy tworzeniu projektów. W większości firm obecnie współpracuje się z klientami zagranicznymi czy wykonawcami zza oceanu. Wiele rzeczy można usprawnić i ułatwić sobie, kiedy możemy bezpośrednio z kimś porozmawiać i wyjaśnić pewne rzeczy. Często niezbędny do tego jest angielski. Zwłaszcza kiedy w twoim zespole zawita ktoś, kto po polsku nie mówi.

To nie jest takie rzadkie, jak mogłoby się wydawać, wręcz przeciwnie.

4.1.3 Samo programowanie

Nauka samego programowania, jeśli znamy język angielski, jest łatwiejsza. Praktycznie każdy język programowania ma słowa kluczowe w języku angielskim.

Kiedy dopiero się uczymy, to dość pomocne, gdyż nawet nie widząc, co dane słowo kluczowe robi, czy co robi dana funkcja, o ile są poprawne nazwane, a obecnie standardy dobrego kodu raczej tego wymagają, to praktycznie bez żadnej wiedzy, bez zaglądania do źródła, możemy zgadnąć, co dany kod robi, wnioskując z samego tłumaczenia danego fragmentu, nazwy funkcji/klasy.

4.1.4 Wyjątki

Wiadomo, zaraz znajdzie się ktoś, kto stwierdzi ‘allle jak to, ja to normalnie pięć lat w zawodzie i jakoś daję radę bez angielskiego!’. Niby tak, ale na jak długo i w jakiej firmie? Chcesz pracować w Januszsofcie? Proszę bardzo, ale ja za chemioterapię, której będziesz potrzebować po doświadczeniu skali głupoty i cebulactwa, jaka ma miejsce podczas pracy dla typowego Janusza, nie oddaje. A uwierz mi, że tak raczej będzie, bo ludzi, którzy kompletnie nie znają angielskiego, raczej się w inne miejsca jakoś zbytnio nie przyjmuje.

To, co u góry napisałem to raczej rzeczy oczywiste dla wszystkich jakkolwiek obeznanych w sprawie, ale niektórzy, na początku (łącznie ze mną) są dość toporni w stosunku do nauki angielskiego.

No bo przecież ja chcę się uczyć programowania!!! Fajnie, ale żeby to robić najefektywniej, jak się da, przydaje się angielski, a potem jest niezrównaną pomocą przy poszukiwaniu i wykonywaniu samej pracy.

To chyba tyle w tej kwestii. Chociaż pozostaje pytanie – jak się uczyć?

4.2 Jak się za to zabrać?

Już tłumaczę.

4.2.1 No, to jak?

Zacznijmy od tego, jak się nie uczyć. Prosty i doskonały przykład tego widzimy w naszych szkołach. To, jak uczy się w nich angielskiego to jakiś żart. Smutny, ale żart.

Nastawienie jest tam jedno najpierw na egzamin gimnazjalny, potem na maturę. O niczym więcej się nie myśli w 90% przypadków. A co jest na egzaminie i maturze? Niekoniecznie rzeczy, których używa się w sytuacjach codziennych.

Będąc w liceum, przynajmniej w moim wypadku, w klasie pierwszej uczeń ma 5 godzin angielskiego, jeżeli wybierze rozszerzenie z tego języka (rzadko się trafia ktoś, kto tego nie zrobi) to od drugiej klasy ma godzin 7. Całkiem sporo, prawda? Owszem, ale to, jak wykorzystuje się ten czas, to tragedia. Jak wygląda przeciętna lekcja? Dostajesz kserówkę ze strukturami gramatycznymi, którą uzupełniasz. Czasem coś napiszesz/powiesz. Dziękuję. Skupienie na gramatyce i... Niczym więcej. Tak być nie może. Nie chcę się tu wdawać w dyskusje nad tym, jak zły jest obecny system edukacji, co w nim zmienić należy i tak dalej to raz, dwa oczywiście są pewnie nauczyciele, którzy odstają od normy i się starają, bo nie zostali jeszcze przez beton zjedzeni, ale ileż ich jest... Przejdźmy do kwestii tego...

4.2.2 Jak zatem powinno to wyglądać, panie mądralo?

Już mówię. Przede wszystkim nie można skupiać się cały czas na rzeczy zupełnie zbędnej w dużej mierze — gramatyce.

Popatrzmy zatem na małe dzieci. Takie załóżmy z 1-2 klasy podstawówki. Umie mówić w swoim języku? A no umie. Nawet zazwyczaj w miarę poprawnie. A zapytajmy tego dziecka, co wie o częściach zdania i mowy. Usłyszymy raczej tyle, co i głuchy.

Czyli dziecko gramatyki tak w zasadzie, formalnie, nie zna, ale językiem poprawnie się posługuje. Ciekawe.

My powinniśmy brać z tego przykład i przede wszystkim pamiętać, by nie skupiać się na nauce gramatyki podczas nauki języka. Oczywiście, są malutkie wyjątki od tej reguły, ale raczej one tu potwierdzają fakt.

Skupić należy się na korzystaniu z języka, a nie teoretyzowaniu. Tylko tak najefektywniej się go nauczymy. Podam przykład. W szkole angielskiego dzieci zazwyczaj uczą się już od przedszkola. Nie inaczej było ze mną — w tak zwanej zerówce zaczęły się pierwsze powolne lekcje. W każdym razie... 7 lat później mój angielski dalej pozostawia wiele do życzenia.

Całe szczęście moje podejście do nauki tego języka zmieniło się z przymusu na bardziej praktyczne. Odeszły w zapomnienie podręczniki, przyszło pisanie i czytanie w tym języku, porozumiewanie się w nim.

Nagle w dwa miesiące zrobiłem większe postępy, niż przez poprzednie 7 lat, zaskakując siebie samego niesamowicie. Gramatyka powinna wynikać z praktyki. Po pewnym czasie przychodzi ona naturalnie, a że nie znamy teorii, która za nią stoi...

Cóż. A w polskim znamy? Ja przynajmniej nie. Jedynie te kompletne podstawy, które w podstawówce się dziecka uczy. Co do reszty... Hah. Może jestem brainletem z 30 IQ, a cała reszta zna, ale wątpię.

4.3 Implementacja tego w praktyce

Moim zdaniem można wyróżnić kilka metod, które należy zastosować.

4.3.1 Tłumaczenie

Zacznijmy od rzeczy, którą lubię najbardziej, i o której mam najlepsze zdanie. Jest to tłumaczenie. Jakie tłumaczenie? Normalne — bierzesz tekst po angielsku i ciach na polski.

Nie przejmuj się, jeśli nie znasz słówek. Ja na początku w ogóle ich nie znałem. Jakieś 80% słówek musiałem sprawdzać w słowniku. Nie skacz jednak do niego tak od razu — spróbuj najpierw wywnioskować coś z kontekstu.

Możesz tłumaczyć cokolwiek. Ja od siebie polecam szczególnie komiksy/mangi/napisy do filmów/jakieś obrazki. Dlaczego?

Bo widzisz, co się dzieje na scenie i więcej rzeczy możesz wywnioskować z kontekstu. Dzięki temu szybciej się nauczysz. Oprócz tego możesz sobie też potem sprawdzać swoje tłumaczenie z tłumaczeniem innych osób — dla przykładu weź jakąś bajkę po angielsku, przetłumacz fragment i porównaj z wersją polską, przetłumaczoną przez profesjonalistę.

W taki sposób bardzo szybko się nauczysz. Mało tego — zaczniesz podłapywać pewne frazy charakterystyczne dla tego języka i nauczysz się, że nie tłumaczymy wszystkiego dosłownie, że nieco inaczej się mówi w języku angielskim, bo pewne słowa mają tam inną wagę, strukturę i znaczenie, niż u nas, a dosłowne tłumaczenie jest bez sensu. Polecam. Dodatkowo nauczysz się też 'myśleć' po angielsku, co jest niezbędne na pewnym, nieco wyższym już, poziomie.

Ja na przykład dość często mam tak, że chcę wyrazić pewną myśl, ale brakuje mi polskich odpowiedników myśli, którą bez problemu mogę wyrazić w angielskim, bo tam akurat istnieje taki zwrot.

4.3.2 Nauka słówek/fraz

Tutaj też troszkę napiszę. Fajnie jest mieć bogaty zasób słownictwa w danym języku, ale za dużo słówek na początek to nam nie trzeba. Wiecie, ile konkretnie wystarczy?

Całe 25. Tak, 25 słówek. Tyle na sam początek ci wystarczy. Bo według badań te 25 słówek odpowiada za 33% treści wypowiadanej przez przeciętnego anglika. Czyli znając 25 słówek, jesteś w stanie zrozumieć jedną trzecią tego, co mówi do ciebie obcokrajowiec.

Do tego dodajmy jeszcze zrozumienie pewnych rzeczy z kontekstu albo podobieństwo brzmienia słów i wychodzi, że prawie się dogadujecie. Ręce pewnie od rozmowy by bolały trochę, myślę, że jednak by dało radę. No ale 33% to nie taki zadowalający wynik. Przejdźmy zatem dalej — by zrozumieć 89%, wystarczy 1000 słów. Znacznie więcej, ale wciąż mało.

Znając 3 tysiące słów, zrozumiemy 95%. Czyli w zasadzie tak, jakbyśmy i w polskim języku rozumieli. Bo te 5% to podejrzewam, że stanowią słówka specjalistyczne, używane w różnych dziwnych zawodach. Przecież tak samo w polskim nie każdy musi wiedzieć, co to jest winkiel, kotwa czy paca do sztablatury. To znaczy większość osób, co na budowie kiedyś pracowało, tak jak ten oto skromny autor tejże książki, będzie kojarzyć, natomiast reszta? Niekoniecznie.

Tak samo i w angielskim są pewne słówka specjalistyczne, które raczej rzadko się przydają a nauczyć się ich można w razie potrzeby.

Czyli takim naszym celem tutaj jest dobicie do 1000 słówek. To wystarczy moim zdaniem na początek. Celujemy w przynajmniej 30 słówek dziennie. Po miesiącu praktycznie jesteśmy w domu. A jak się ich uczyć? Bo tak siedzieć i po prostu czytać to bezsensu. Oferuję dwa świetne narzędzia:

Memrise Duolingo

Obie te platformy są dostępne za darmo, oferują aplikacje na Androida czy też iOS.

Chyba nawet gdzieś jeszcze wersja na, o zgrozo, Windows Phone się znajdzie.

Te dwie świetne platformy bez problemu pomogą ci w nauce angielskiego, jak i innych języków. Którą z nich wybrać, czym się różnią?

Obie pomagają w nauce, ale memrise jest bardziej zróżnicowany i zależny od treści, które stworzą użytkownicy. Można się tam uczyć nie tylko słownictwa, ale także takich rzeczy jak geografia, historia, literatura i wielu, wielu innych. Do nauki słownictwa sprawdza się świetnie.

Co z Duolingo? Tutaj mamy nieco bardziej usystematyzowane podejście. Kursy przygotowane są przez profesjonalną ekipę (Memrise też takie ma, ale nie jest ich aż tak dużo) i pokrywają one zarówno słownictwo, jak i podstawy gramatyki.

Niemniej jednak oba serwisy są podobne, uczą słówek, wymowy, gramatyki. Wybór pozostawiam wam, chociaż przyznam, że ja z obu korzystam.

A raczej korzystałem, bo teraz jestem leniwym dziadem, któremu nie chce się uczyć nowych języków, argumentując sobie to tym, że za kilka lat będziemy mieli AI od tłumaczenia.

4.3.3 Słuchanie

To chyba rozumie się samo przez się. Polecam filmy i bajki z napisami, ale angielskimi. O polskich zapomnij. Jeśli czegoś nie rozumiesz, przystopuj, włącz napisy i przetłumacz sobie powoli. Koniecznie. Chociaż w sumie sam nie wiem, czy powinienem polecać naukę słuchania — jak już trochę zaczniesz rozumieć, to okaże się, że 90% twoich ulubionych piosenek angielskojęzycznych traci sens. To ssie.

4.3.4 Mówienie

Mów do siebie, mów do rodziny, kumpli, drzewa, psa, mikrofonu. Są specjalne grupy, gdzie możesz dogadać się z kimś, kto też się uczy, albo kto przyuczy cię w zamian za nauczenie polskiego.

Możesz napisać do mnie i umówić się na pogaduszkę na skypie albo ts, chętnie pomogę. Na blogu znajdziesz kontakt – email, GitLab, GitHub albo na forum 4programmers. Ba, możemy nawet stworzyć małą grupkę pomagającą w nauce języków. Ja jestem na tak, a co ty na to? I pamiętaj jedno — nie bój się.

Nikt nie dba o twój akcent i wymowę. Będąc w Londynie czy za granicą ogółem, słyszałem tyle różnych wersji języka angielskiego, że głowa mała. Często gęsto nawet ci, dla których język angielski jest językiem narodowym, nie mogą się między sobą dogadać ze względu na akcenty i slangi.

To tak jakbyś ze ślonzakiem gadoł. Tylko ze trzy razy gorzej i oprócz ślonzaka masz jeszcze 15 innych grup. Także spokojnie. Zrozumieją cię, nikt nie będzie cię oceniał, a raczej zachęcał i doceniał wysiłek, który wkładasz w próby nauki języka. Serio.

4.3.5 Czytanie

Konsumuj treści po angielsku i tyle. Nie przejmuj się na początku stopniem zrozumienia, spokojnie. Wzrośnie. Listę ciekawych blogów w tematyce programowania znajdziesz pod koniec tej książki, zajrzyj do spisu treści.

4.3.6 DeepL

Fajnym ćwiczeniem jest pisanie zdań po polsku, niech to będzie dla przykładu 10, następnie tłumaczenie całych tych zdań za pomocą DeepL. Proste zdania wystarczą. DeepL to coś jak Google Translate. Następnie proces odwracamy. Sami piszemy 10 innych zdań i w DeepL sprawdzamy poprawność.

4.4 Podsumowując

Słuchając się tych rad i poświęcając około 2 godziny dziennie, po miesiącu będziesz dość płynnie rozumiał jakieś 70-80% codziennych konwersacji w języku obcym!

Oczywiście chodzi tu o konwersacje proste, prowadzone za pomocą normalnego akcentu, ale wciąż. Specjalistyczne słownictwo, które potrzebne ci będzie podczas nauki programowania, przyswoisz po drodze.

Zapamiętaj sobie zatem szczególnie te dwa linki do Memrise i Duolingo oraz to, że masz tłumaczyć, czytać, słuchać i mówić jak najwięcej.

Oprócz tego gorąco zachęcam cię do obadania sobie książki Radka Kotarskiego pt. „Włam się do swojego mózgu” - to zbiór ciekawych technik i rad o tym, jak się uczyć, żeby się nauczyć tak naprawdę, żeby zapamiętać szybko i efektywnie.

Nie, Kotarski mi nie sypnął groszem za reklamę. A szkoda, zawsze chętnie przytulę jakieś szekle.

W każdym razie. Przejdźmy wreszcie do opisu języka, którym będziemy się tu w głównej mierze zajmować.

5 Python, o co chodzi?

5.1 Python – o co chodzi?

Zanim weźmiemy się za naukę Pythona, nieco opowiem o historii samego języka, skąd się on wywodzi, jakie ma cele, założenia, jak ewoluował, zmieniał się na przestrzeni lat, a jak wygląda obecnie, gdzie jest wykorzystywany i dlaczego.

5.2 Jak korzystać z tej książki

Chwila, stop. Chciałbym nieco Ci powiedzieć, jak moim zdaniem należy z tej książki korzystać. Po pierwsze, starałem się, by była ona napisana tak, żeby z jednego tematu naturalnie przechodzić do następnego – od tematów prostych, do tych nieco bardziej skomplikowanych. Dzięki liniowej budowie powinna być ona łatwa do zrozumienia.

Jednak jeśli jesteś już nieco bardziej doświadczonym programistą, czy też po prostu chcesz sobie przypomnieć pewne rzeczy, to skacz po niej śmiało – większość rozdziałów jest w miarę zamkniętymi komponentami.

Kolejną rzeczą, o której chcę wspomnieć, a raczej powtórzyć, to fakt, że w tej książce nie znajdziesz informacji wyłącznie o Pythonie. Poza informacjami, o samym języku, postaram się wprowadzić Cię też w pewne pojęcia z ogólnie pojętej informatyki, tak, byś wiedział nieco więcej, byś miał pojęcie, jak coś działa, dlaczego tak, a nie inaczej.

Uważam, że to niezbędna wiedza, by zostać dobrym programistą, który się rozwija i daleko zajdzie. Takie rozdziały będą nieco bardziej teoretyczne w swej naturze, ale nie znaczy to, że nudne, wręcz przeciwnie moim zdaniem.

Jeśli chcesz dobrze przyswoić informacje tu zawarte i naprawdę się nauczyć czegoś nowego, to dobrze radzę: po pierwsze, rób notatki, krótkie, treściwe i proste.

Po drugie – przepisuj kod. Nie korzystaj z metody Copiego Pasty. Przepisuj samodzielnie i koniec.

Na koniec, trzy – samodzielnie wykonuj zadania, które będę umieszczał na końcu rozdziałów, ale to nie wszystko – eksperymentuj z kodem. Zmieniaj go, zobacz, jakie będą efekty tych zmian. Przekonaj się o tym w praktyce, przeanalizuj swoje modyfikacje, przemyśl je i ich rezultaty, to jak wypływają na działanie programu. To jest najlepsza metoda nauki. To podstawowe założenie, które przyświecało mi przy pisaniu, że będziesz samodzielnie wykonywał/wykonywała ćwiczenia, czytał/czytała dodatkowo inne artykuły, książki, prowadzić eksperymenty z kodem.

Więcej o tym, jak powinno się uczyć i jakie metody się sprawdzają, możecie przeczytać we wpisie na blogu Gynvael'a Coldwind'a – Poradnik początkującego programisty. Grski poleca. W ogóle cały blog wam polecam. Mało jest miejsc w Internecie, gdzie znaleźć można tak dobre i ciekawe treści. Googlnijcie, bo warto.

Dodatkowo pragnę zaznaczyć, że nie będę w tej książce pisał o wszystkich najdrobniejszych rzeczach. To nie jest encyklopedia Pythona, założenie jest tutaj takie, że po szczegóły różnych rzeczy sięgniecie sami. Ja chcę zwrócić uwagę na często pomijane rzeczy. **Zalecam zatem, by obok tej książki poczytać sobie materiały z dokumentacji Django, Pythona, wyśmienitą książkę Learning Python, 5th edition oraz kurs CS50**. Ja bardziej chcę uświadomić wam istnienie pewnych rzeczy i niektóre koncepty

objaśnić, dodatkowo zająć się tą często pomijaną wiedzą praktyczną, szczerym spisem swoich doświadczeń.

5.3 Część interaktywna

Na moim githubie znajdziesz repozytorium 'junior-python-exercises - https://github.com/grski/junior-python-exercises. Jeśli życzysz sobie, by twoje rozwiązania zadań i odpowiedzi były sprawdzane, sforkuj to repo (potem dowiesz się co to znaczy) następnie w folderze odpowiednim dla rozdziału dodaj folder ze swoim nickiem z githuba a w środku odpowiedzi. Ja zrobię code review/przejrzę te odpowiedzi i dam jakiś feedback :)

5.4 Python 2 – Python 3?

Python obecnie jest głównie w dwóch wersjach – Python 2 i Python 3. Są to dwa 'główne' wydania tego samego języka, jednakże wersja 3 jest nowsza, wprowadza pewne nowe rzeczy, które nie są wstecznie kompatybilne z wersją nr 2, stąd ten przeskok numeru.

Wprowadzenie Pythona 3 nastąpiło wiele lat temu, obecnie mamy czasy, kiedy Python 2 nie jest już rozwijany. Umarł i koniec. Jedyne projekty, jakie w nich są, to jakieś grubsze legacy. Poza tym, gdzieś tam słychać powoli jakieś głoski o Pythonie 4.

Co to znaczy z twojej perspektywy, jako początkującego? Nic. Po prostu wiedz, że obecnie uczysz się Pythona 3 i tyle. To wciąż ten sam język, jednak występują między nim, a Pythonem 2 pewne różnice, które w razie czego, możesz bez problemu poznać w kilka chwil. Ja wybieram wersję najnowszą, by przedstawić ci najświeższe informacje to raz, a dwa, tak szczerze, Python 2 staje się już powoli reliktem przeszłości, a kto tworzy w nim obecnie nowy software, robi błąd, chociaż nie wydaje mi się, by takie przypadki można było jeszcze gdzieś znaleźć, poza paroma korpo wyjątkami.

Jak już, to Pythona 2 wykorzystuje się teraz tylko do utrzymania starych aplikacji w nim napisanych, a uwierz mi na słowo, częściej niż rzadziej, nie chcesz pracować przy utrzymaniu starych molosów. Chyba że płacą Ci dużo pieniędzy. Dużo, dużo pieniędzy. Jeszcze więcej niż dużo. Poważnie. Chociaż i tak nie warto. Co po pieniądzach, gdy dosłownie zmieniasz zawód z programisty na szambo-nurka, nurkując w ekskrementach programistycznych jakichś ludzi, którzy bardziej niż zwykle nienawidzili swego życia, więc stworzyli potwora?

Python 2 przechodzi do przeszłości i dobrze, niemniej jednak czasem napomnę o różnicach, tak w ramach ciekawostki w zasadzie.

5.5 Krótki opis długiej historii Pythona

Python jest językiem dość starym, że się tak wyrażę. Starszym ode mnie, chociaż to żadne osiągnięcie akurat.

Obecnie jest pełno nowych języków na rynku języków dość nowych, dzieci jak Scala, Dart, Elm, Elixir, Kotlin czy wiele, wiele innych. W porównaniu z nimi Python jest staruszkiem, pojawił się on bowiem na początku roku 1991 na CWI – Centrum Matematyki i Informatyki w Amsterdamie. Nie jest to co prawda dziadunio taki jak np. C z roku 1972, ale taki pełnoprawny język w średnim wieku, to już jak najbardziej.

Jego głównym twórcą był Guido van Rossum, który do dziś ma przydomek „Benevolent Dictator for Life" (a w zasadzie miał, o czym za chwilę) i w zasadzie uznawany jest za najwyższy autorytet w świecie Pythona.

Jeśli o takie nietypowe fakty chodzi, to można dodać, że jeśli uda ci się znaleźć adres e-mail Guido gdzieś w sieci (nie jest to trudne) i wyślesz mu ciekawy mail z pytaniem, czy czymkolwiek, to istnieje spora szansa, że odpowie. Tak przynajmniej wynika z doświadczeń sporej ilości innych użytkowników Pythona.

Sama nazwa języka nie pochodzi, jak by się mogło wydawać, od gatunku węża, a od serialu emitowanego w latach siedemdziesiątych przez BBC - „Latający Cyrk Monty Pythona", którego to Guido był fanem i uznał, że nazwanie swojego języka programowania 'Python', to będzie coś. No i w sumie jest. Samo to powinno wystarczyć, by zachęcić Cię do używania tego języka.

Python rozpoczął swój żywot publiczny w wersji 0.9.0, ale za długo w niej nie pożył, szybko wychodziły kolejne wersje. Wiązały się z nimi różne perturbacje, gdyż sam van Rossum, jak i najważniejsi członkowie zespołu, wiele razy zmieniali swój 'dom', przechodząc z jednej organizacji do drugiej. Kod kodem, ale za coś żyć trzeba.

Wszystko się jednak ustatkowało wraz z nastaniem wersji 2.1, która została wydana już pod szyldem Python Software Foundation – fundacji non-profit, która działa do dziś i wzorowana jest na Apache Software Foundation.

W założeniu Python miał być następcą języka ABC – innego prehistorycznego tworu, o którym nie będziemy się zbytnio rozwodzić, mimo tego, że wpływ jego jest ewidentny w Pythonie.

Oprócz ABC, w Pythonie wyraźne są pewne wpływy lub elementy zapożyczone z takich języków jak: C, C++, Haskell, Java, Perl czy Lisp. Czy powinno ci coś to mówić? Zdecydowanie nie, jeśli jesteś początkujący albo początkująca. Wiedz po prostu, że Python ma pewne elementy wspólne z innymi językami i nie jest jakimś tam wielkim dziwolągiem.

5.6 Abdykacja Guido

W okolicach czasu pisania tej książki, w sumie na samym początku (cóż mogę powiedzieć, robiłem długie przerwy...), stała się rzecz niesłychana, otóż Guido van Rossum, autor Pythona, postanowił oddalić się od łańcucha decyzyjnego w świecie Pythona i zrzucić swój tytuł BDFL, powoli w ogóle przechodząc na emeryturę. Całość spowodowana była PEP 572, który zaproponował między innymi sam Guido, a który wywołał dość nieprzychylne reakcje społeczności. O co się rozchodziło?

O operator := i przypisanie w wyrażeniach. Spora część osób bardzo głośno i donośnie zaczęła krytykować ten pomysł, często bez jakichkolwiek podstaw, gdyż, przynajmniej mnie, sam PEP wydaje się raczej przemyślany i fajny, ta funkcjonalność na pewno się gdzieś przyda w Pythonie. O samym tym PEP-ie porozmawiamy później jeszcze, więc na razie bez szczegółów.

Poniżej zamieszczam, przetłumaczony przeze mnie tekst e-maila, który opublikował Guido.

„Teraz kiedy sprawa PEP 572 została wreszcie zamknięta, nie chcę już nigdy musieć włożyć w coś tyle wysiłku i walki, tylko po to, by przekonać się, że ogrom osób potępia moje decyzje.

Chciałbym kompletnie odizolować się od procesu decyzyjnego w kwestiach związanych z Pythonem. Przez chwilę wciąż zamierzam pozostać aktywnym jako zwykły Core Developer, dalej będę mentorował ludzi — teraz w sumie nawet może więcej. Niemniej jednak oficjalnie daję swojej osobie takie permanentne wakacje od tytułu BDFL, musicie radzić sobie sami.

No bo tak w zasadzie to cóż, jest to coś, co prędzej czy później i tak by was czekało — za każdym rogiem wciąż czai się gdzieś jakiś tir, który chce mnie rozjechać, to raz, a dwa, że lat mi nie ubywa z biegiem czasu... Oszczędzę wam listy moich problemów zdrowotnych.

Nie zamierzam wyznaczać swego następcy.

Co zrobicie? Jakaś demokracja? Anarchia? Dyktatura? Federacja?

Nie przejmuję się tymi trywialnymi, pomniejszymi decyzjami podejmowanymi codziennie w Issue Trackerze, czy też na GitHubie. Rzadko ktoś mnie tam o opinie prosi, a jak już to raczej w sprawach niezbyt ważnych. Tutaj zatem nic się nie zmieni. I o to jestem spokojny.

Nad czym trzeba będzie jednak pomyśleć to: - to, w jaki sposób będą zatwierdzane lub odrzucane nowe PEP-y; - to, w jaki sposób wybierać nowych Core Devów;

Myślę, że może nam się udać, zdefiniować to, jak te procesy mają wyglądać, w formie nowych PEP-ów, które to będą swoistą nową konstytucją dla tej społeczności. Cały haczyk w tym wszystkim polega jednak na tym, że ja wam tutaj nic nie podpowiem — wy, comitterzy, będziecie musieli sami wszystko wymyślić, ustalić.

Pamiętajcie jednak o tym, że wciąż istnieje tu coś takiego jak CoC (Code of Conduct, czyli taki jakby regulamin), jeśli się wam to nie podoba, to nie pozostaje wam nic innego niż opuścić tę grupę. Może to też trzeba przemyśleć, kiedy kogoś stąd wyrzucić (czyli zbanować z grupy python-dev albo python-ideas).

Na sam koniec przypomnę tylko, że archiwa z tej grupy mejlowej są dostępne publicznie.

Nie odchodzę, wciąż tu jestem, ale chcę, byście zaczęli być samodzielni. Jestem zmęczony i potrzebna mi długa, długa przerwa". —Guido van Rossum

Cóż można tu więcej wspomnieć.

To był smutny moment w historii Pythona, tym bardziej że tak naprawdę Guido to Python a Python to Guido.

To tyle. Szokujące wiadomości.

Cóż, niechaj się wiedzie panu, panie Guido.

Co prawda nie do końca darzę go pełnią sympatii, przez różne mocno poprawnie polityczne zachowania, wręcz bym powiedział, że paranoiczne, ale wciąż. Zrobił kawał dobrej roboty i za to szacunek mu się należy.

Pytanie jednak jak będzie wyglądał świat Pythona bez tego człowieka? Jaki kierunek wybierze? To okazja do wzrostu, ale zarazem zagrożenie, że zgubimy kierunek, który obecnie jest dość fajny.

W każdym razie... Nowy wiatr zadmie w żagle, dokąd nas zaprowadzi? Czas pokaże. Za rok, dwa, pięć. Zdecydujemy o tym my, społeczność, która tworzy Pythona, czyli w zasadzie za niedługo i ty, drogi czytelniku.

Notka: tak z perspektywy czasu, to dalej jest git. Zwinięcie Guido nie zmieniło jakoś Pythona na gorsze.

5.7 Wężowe cele

Z tym Pythonem, to tak sprawa wygląda, że naprawdę warto go używać. Osobiście uważam, że jest on jednym z najlepszych języków do nauki podstaw programowania, czyli taki, jak miał być w zamyśle. Pozwala on szybko przejść do zrozumienia pewnych pojęć programistycznych, gdyż uczeń nie musi skupiać się zbytnio na opanowywaniu skomplikowanej składni czy zawiłych wyrażeń, jak to czasami ma miejsce w innych językach.

Python jest zwięzły i prosty – większość kodu może być bardzo łatwo zrozumiana przez kogokolwiek, kto choć trochę mówi po angielsku, lub ma słownik pod ręką, a sam kod jest zazwyczaj krótki i elegancki.

Wyjaśnienia tak naprawdę wymaga tylko kilka symboli, używanych, by skrócić zapis. Poza tym Pythona czyta się tak naprawdę podobnie, jak zdania w języku angielskim. To jest też właśnie pierwszy wężowy cel – prostota przedkładana nad złożoność.

Python z założenia miał być prosty, przyjemny i elegancki. Uważam, że taki jak najbardziej jest.

Kolejnym celem jest przenośność. Pythona można uruchomić chyba na większości obecnie popularnych platform – Windows, prawie dowolny Linux, macOS. Do wersji 3.7, wydanej 27.06.2018, oficjalnie wspierane było nawet coś tak dziwnego, jak FreeBSD w wersji <= 9.

Dzięki temu Pythona można odpalić prawie wszędzie, jeśli tylko pozwalają na to zasoby sprzętowe.

Następnym celem, który miał przed sobą Python, jest otwartość. Niektóre języki są dość 'hermetyczne' - można ich używać tylko po opłaceniu licencji albo tylko w określonych warunkach, celach.

Z Pythonem tego problemu nie ma – jest on kompletnie darmowy w użytkowaniu, modyfikacji, dystrybucji i czymkolwiek tam jeszcze sobie użytkownik życzy.

Dodatkowo Python jest otwarto-źródłowy, a za jego rozwojem stoi społeczność. Co to znaczy w praktyce? Każdy ma wgląd do źródeł języka to raz, dwa, że jeżeli coś ci się w Pythonie nie podoba, uważasz, że coś mogłoby zostać zrobione lepiej, to...

Nie ma problemu. Weź daną funkcję i po prostu ją dopisz, zmień. Jeśli społeczność uzna, że twoje zmiany są zasadne i przydatne, to wylądują one w samym języku. Każdy może mieć zatem realny wpływ na to, jak wygląda Python, jak on działa. Świetna sprawa.

To tylko kilka z idei, które przyświecają Pythonowi, wszystkich ich tutaj nie opiszę, ale uważam, że te najważniejsze udało mi się zawrzeć. ## Wąż co jakiś czas zrzuca skórę

O cóż mi chodzi? O fakt, że Python i jego zastosowania cały czas się zmieniają. Podobnie jak wąż, zrzuca on swoją starą skórę i zyskuje nową.

Pierwotnie był to język, który wykorzystywano raczej jako język skryptowy. Automatyzacja pewnych procesów na serwerach, jakieś operacje na plikach, tekście i tak dalej. Nudne rzeczy ogółem. Python nie zdobył rynku szturmem, troszkę mu to zajęło. Na samym początku był raczej niszowy. Z czasem i ewolucją samego języka, powszechnie zaczęto dostrzegać jego zalety i piękno.

Dlatego też na przestrzeni lat Python stawał się coraz to popularniejszy, jeśli chodzi o rozwój backendowych części aplikacji webowych, a jak sami wiemy, te od początku ostatniego tysiąclecia tak naprawdę, przeżywają nieustanny rozkwit. Sprawę ułatwiało pojawienie się na rynku kolejnych frameworków pythonowych, przeznaczonych właśnie

do tworzenia aplikacji internetowych, takich jak Flask, Pyramid, Pylons, Web2py czy wreszcie Django, które było prawdziwym game-changerem. Taka ciekawostka. Jak sobie instagramika przeglądasz, to wiesz na czym stoi? A no na Django właśnie.

Dzięki temu Python stał się całkiem popularny jako język wykorzystywany do pisania aplikacji internetowych, ale to nie wszystko. W obecnych latach możemy zauważyć nieustający wzrost zapotrzebowania na różnych specjalistów związanych z Data Science, Artificial Intelligence, Machine Learning czy Neural Networks.

Wszystkie te i pokrewne branże rozwijają się niesamowicie, a językiem, który w zasadzie praktycznie tam króluje, jest Python. Ma tam swojego konkurenta w postaci R, czy szybko rosnącej zawodniczce w postaci Julii, ale wciąż, trzyma się mocno ten nasz wąż.

Dlaczego? Spójrzcie na idee, jakie przyświecają Pythonowi – samo się wyjaśni. Analityk nie ma być dobrym koderem, on ma za zadanie przetworzyć dane, zatem potrzebny jest mu język, który szybko pozwoli mu, bez zbędnego zagłębiania się w składnie czy to, jak działa sam język, przelać swoje myśli w kod.

Python idealnie się do tego nadaje z racji swej prostoty i wszechstronności. Już sobie wyobrażam jak siedzi taki analityk jeden z drugim i patrzą, czy na pewno zwolnili całą zaalokowaną wcześniej pamięć w ich super pięknym kodzie napisanym w C albo C++. Nie ma takiej opcji po prostu. I dobrze.

Co prawda powstaje tutaj pewien problem w postaci wydajności, ale o tym później, bo jest to coś, co można przeskoczyć i rozwiązać o wiele łatwiej, niż gdyby próbować nauczyć każdego C/C++.

Oczywiście Pythona wciąż używa się w różnego rodzaju skryptach, automatyzacji i tak dalej, niemniej jednak uważam, że nie jest to już jego główne zastosowanie, jak to było lata, lata temu.

Tak więc, jak widzicie, Python się rozwija i pojawia w coraz większej ilości projektów, dziedzin i stref związanych z ogólnie pojętą informatyką. Osobiście uważam, że ten trend raczej się utrzyma, podobnie zresztą, jak do tej pory, i Python z roku na rok będzie zyskiwał coraz większą popularność, wejdźmy jednak w szczegóły – dlaczego?

5.8 Zalety Pythona

5.8.1 Ekspresywność

Python jest bardzo ekspresywny. Co to znaczy? Otóż w Pythonie za pomocą relatywnie niewielkiej ilości kodu, można osiągnąć to, co w innych językach zajęłoby czasami kilka razy tyle. Żeby nie być gołosłownym, popatrzmy na przykład klasycznego programu, którym zaczyna się naukę programowania – Hello World, czy po polsku, Witaj Świecie. Co w ogóle jest ironią, bo jak zaczynasz programować, to się raczej ze światem powinno żegnać, bo więcej go już za bardzo nie będziesz oglądał/a ze swojej piwnicy. W Pythonie wygląda on tak:

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

Dość proste i zrozumiałe, prawda? Jedna linijka i gotowe. Przyjrzymy się natomiast innym językom. Zacznijmy od Javy:

```java
public class HelloWorld{
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Tu dalej w miarę jasno, mimo kilku pozornie tajemniczych komend, wciąż można łatwo odczytać, co dany program robi, ale weźmy, załóżmy, na celownik C++:

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

Tutaj już troszkę mniej oczywistym jest, co dany program robi, prawda? W dodatku popatrzcie na ilość linii użytych do wykonania zadania. Nie ma porównania.

Zaznaczam jednak, że w obu tych językach można by wydrukować hello world'a za pomocą krótszego kodu, nie znam tychże sposobów, bo żaden ze mnie Javowiec czy gość od C/C++, jednocześnie zbyt leniwa buła, by ich poszukać, niemniej jednak chodzi mi tutaj o samo pokazanie idei.

Innym przykładem tego, jak krótki może być kod w Pythonie względem porównywalnego kodu w C++/Javie czy innych języków, jest niżej, to kod nieco bardziej skomplikowany, ale być może coś z niego zrozumiesz. Dostarczył mi go w komentarzu @jacekw na Steemit i to jego autorstwa jest tenże kod. Dzięki :)

A więc cóż będzie nasz zadany kod robił? Jego zadanie jest proste:

1. stwórz listę liczb w kolejności malejącej 19 do 0.
2. Pomiń liczby parzyste.
3. Podnieś każdą liczbę do kwadratu.
4. Posortuj rosnąco.
5. Wypisz wynik.

Swoją drogą, jest to forma algorytmu, przedstawiona w postaci krokowej, coś, do czego jeszcze wrócimy w przyszłości. Obecnie zapamiętaj sobie jedno – algorytm to po prostu jednoznaczny zbiór instrukcji służących wykonaniu jakiegoś celu.

W każdym razie.

Zacznijmy tym razem od *C++* może.

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

int main() {
```

```
    vector<int> s, a;
    for (int i = 19; 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;
}
```

Nie wiem jak wy, ale dla mnie to dość zawiły fragment kodu. Jako początkujący programista miałbym problem z jego zrozumieniem. Sporo dziwnych symboli, znaczków. O co tu tak dokładnie chodzi?

Trudno powiedzieć na pierwszy rzut oka.

Następny uczestnik tego porównania to...

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:

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

Alternatywnie moja wersja w Pythonie wyglądałaby tak:

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

Pozostawię bez komentarza. Na koniec dodam jednak jeszcze, jako ciekawostkę, przykład z innego języka – Haskella.

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

Też ciekawy, prawda?

Czy to znaczy, że te języki są gorsze, a Python jest królem? Absolutnie nie, nigdy tak nie myśl.

Każdy język jest jak narzędzie – ma swoje zastosowania, w których jest dobry, wyśmienity, ale ma też takie, do których kompletnie się nie nadaje. Tak jest i tutaj. Tak jest wszędzie. Owszem, czasem zdarzają się fanatycy danych technologii czy rozwiązań, którym językowo-technologiczne zapalenie opon mózgowych przyćmiewa obiektywny osąd, ale to nic. My nie chcemy tacy być. Bądźmy mądrzy i rozsądni, ułatwiajmy sobie życie, używając odpowiednich narzędzi do odpowiednich zadań.

Niemniej jednak Python pozwala nam pisać więcej w mniejszej ilości kodu. To oczywiście przychodzi w zamian za pewną cenę, którą trzeba zapłacić, i która sprawia, że Python dobry jest w pewnych sytuacjach, a w innych już nie.

5.8.2 Prostota

Wróćmy do poprzedniego punktu – jak popatrzymy na kod Pythona, to można by rzec, że to w zasadzie po prostu zapisane polecenie po angielsku. Cóż bowiem znaczy słowo print? Nic innego jak drukuj/wydrukuj.

Od razu można się domyślić, że programiście chodzi o to, by komputer coś wypisał na ekran. Podobnie sprawa się ma z innymi elementami języka, naprawdę, wystarczy znać angielski i już prawie możemy zrozumieć sporą część Pythona. W dużej liczbie języków jest podobnie, ale nie są one aż tak bardzo podobne do angielskiego, jak Python.

Naprawdę, drugiego tak prostego języka jeszcze nie spotkałem, a trochę ich, przynajmniej pobieżnie, zdarzyło mi się używać – czy to JavaScript, Java, C, C++, Dart, Scala.

Jedynym językiem, który może konkurować z Pythonem prostotą, jest chyba C – ale to z racji tego, że core tego języka jest po prostu malutkie. Gdy przyjdzie nam do zarządzania pamięcią, wskaźników i innych, równie fajnych rzeczy z C, to zaczyna się tęsknić za Pythonem.

5.8.3 Python językiem dynamicznie typowanym

Co to znaczy? Cóż, jeśli jesteś nowicjuszem w programowaniu, to możesz kompletnie nie mieć pojęcia, o co chodzi, ale to nic.

W skrócie sprawa tyczy się faktu takiego, że w statycznie typowanych językach, podczas deklaracji zmiennych, należy podać, jakiego typu dane będzie ta zmienna przechowywała. Za przykład niech posłuży nam tutaj Java:

```
int someNumber = 123;
```

Zapis powyżej mówi ‘programowi’, który będzie wykonywał nasz kod, że chcemy utworzyć zmienną o nazwie someNumber, która będzie zawierała dane typu int – integer, czyli nic innego jak liczby całkowite. A co to w ogóle znaczy, że ma utworzyć zmienną?

Spróbuję wytłumaczyć to dość prosto, ale może mi się nie udać i jeśli nie do końca zrozumiesz zasadę działania tego mechanizmu, lub trudno będzie Ci wyobrazić sobie jak to funkcjonuje, to nie przejmuj się, wrócimy do tematu później.

To polecenie spowoduje ‘powiedzenie’ naszemu komputerowi czegoś takiego:

Słuchaj komputer, tu masz jakieś dane, (w tym wypadku '123'), zapamiętaj sobie tę wartość, zapisz ją gdzieś, bo będę chciał w przyszłości tego do czegoś użyć, i od teraz, za każdym razem, gdy napiszę w programie `someNumber`, to wiedz, że chodzi mi właśnie o wartość zapisaną w tamtym miejscu.

Próba zapisania innych danych do tej zmiennej, załóżmy, tablicy czy liczby rzeczywistej, nie skończy się zbyt dobrze lub po naszej myśli.

W Pythonie takich obostrzeń i wymogów nie ma. Po pierwsze, podczas inicjalizacji zmiennej, nie musimy podawać jej typu, a po drugie później, możemy bez problemu zmieniać rodzaj danych, jaki przechowujemy w danej zmiennej, zatem ekwiwalentem zapisu wyżej w Pythonie, byłby kod:

```
some_number = 123
```

Później bez problemu zaś, możemy wpisać sobie:

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

Z czego to wynika, dowiesz się już nieco dalej w książce, ale głównie chodzi o to, że w Pythonie zmienne to tak naprawdę referencje do obiektu, a nie sam obiekt per se. Jak powiedziałem, wrócimy do tego, na razie nie musisz zaprzątać sobie tym głowy. Zapamiętaj tylko to, że Python ci życie ułatwia i nie narzuca za dużo, robi robotę za ciebie, poczciwy wąż.

5.8.4 Społeczność

Python ma jedną, naprawdę dużą zaletę. Jest to jego społeczność, która raz, że jest naprawdę pomocna, dwa, że jej rozmiar jest imponujący. Dzięki temu ilość dostępnych materiałów, poradników, bibliotek, framework'ów i skryptów potrafi po prostu pozytywnie zaskoczyć.

Dzięki otwarto-źródłowej kulturze Pythona wiele niesamowitych narzędzi codziennie jest oddawanych w nasze ręce do użytku, zupełnie za darmo, tak po prostu.

Powoduje to, że często nie musimy wymyślać koła na nowo – wystarczy import jakiejś biblioteki, którą ktoś już kiedyś napisał. Oszczędza to często czasu i zmartwień, pozwalając się skupić na tym, co w naszej implementacji ważne.

Poza tym, co zrobić, kiedy utkniemy w którymś miejscu pisania programu i nie wiemy co dalej, gdy napotkamy jakiś błąd, którego nie potrafimy rozwiązać? Cóż, z racji wieku samego Pythona oraz rozmiaru jego społeczności, można w większości przypadków założyć, że ktoś przed nami napotkał już podobny problem i zapytał o to w Internecie, lub opisał rozwiązanie danego problemu.

Ludzie chętnie dzielą się wiedzą wbrew pozorom. Dzięki temu nie musimy sami szukać rozwiązania godzinami, grzebiąc w dokumentacji, źródłach czy po prostu eksperymentując. Możemy najzwyczajniej w świecie kogoś zapytać, bo dużo osób zna Pythona, albo znaleźć odpowiedź innych, którzy rozwiązali ten problem przed nami. Nie każdy język ma tak rozbudowaną bazę wiedzy i społeczność.

Jako kontr przykład, podam język Dart. Dość nowy język, niezbyt popularny, ogółem mała społeczność. Niemniej jednak czasem tworzyłem w tym języku i zdarza się nierzadko, iż napotykam jakiś problem, o którym informacji nie znajdę nigdzie, bo po prostu akurat

nikomu innemu się nie przydarzył jeszcze, lub też nikt inny go nie opisał, więc muszę sam szukać rozwiązania, przeczesując dokumentacje, źródła i po prostu eksperymentując.

To zaś jest często droga przez mękę.

Do tego czasami dochodzi niedogodność w postaci tego, że niektóre rzeczy, które np. w Pythonie czy Javie zostały już napisane przez kogoś innego, ładnie zapakowane w paczkę i rozpowszechnione do użycia, w Darcie niekoniecznie są i trzeba je napisać samodzielnie.

Podobnie, jeśli chodzi o samą naukę języka – materiałów jest znacznie mniej, często są one przestarzałe z racji tego, że Dart jest językiem ciągle się rozwijającym i to mocno, co tydzień wychodzą nowe wersje, a przynajmniej kiedyś tak było, czy jest tak dalej, to nie wiem, samego języka, a z racji niskiej popularności mało osób o nim pisze, jeszcze mniej tworzy o nim książki czy poradniki.

To nie ułatwia nauki, zwłaszcza początkującym programistom. Dobrze, że chociaż dokumentacja jest całkiem dobra, co prawda nie tak dobra, jak dokumentacja np. Django, ale i tak – nie jest źle.

Dlatego też twierdzę, że społeczność Pythona jest jego największą zaletą i to ona, dosłownie, tworzy ten wspaniały język, czyniąc go tym, czym jest.

5.8.5 Mnogość zastosowań

Python jest językiem ogólnego przeznaczenia. Można w nim stworzyć praktycznie wszystko, poza pewnym, raczej wąskim gronem zastosowań, do których kompletnie się nie nadaje i do których nie był projektowany.

Znając Pythona, możemy tworzyć aplikacje desktopowe, gry, aplikacje webowe, skrypty, emulatory, interpretery, kompilatory, aplikacje do obliczeń naukowych, aplikacje do wizualizacji danych i ich scrapowania z sieci, uczenie maszynowe i tak dalej. Lista jest naprawdę długa.

Oczywiście, do jednych zadań Python jest lepszy, do innych gorszy, bo na przykład rzadko zdarza się, że aplikacje desktopowe czy gry tworzy się w Pythonie, gdyż są do tego lepsze języki, ale w każdym razie, jest to możliwe i niezbyt trudne tak szczerze.

W tym, że Pythona można wykorzystywać do wielu rzeczy, pomaga to, o czym pisałem wyżej – czyli duża społeczność tworząca ogromne ilości bibliotek, frameworków i gotowych skryptów.

Pozwala to na wygodne użycie Pythona w różnych dziedzinach i to dzięki temu oraz samej prostocie języka, zdobywa on szturmem inne pola, poza webdevem i devopsami, jak Data Science, Artificial Intelligence, Neural Networks czy na ogół obliczenia naukowe.

Po prostu czasem utworzenie programu w Pythonie sprowadza się tak naprawdę do zaimportowania jakiegoś modułu i dodaniu kilku drobnych komend mówiących mu, co ma dla nas zrobić. Prościej się nie da.

5.8.6 Czytelność

Python projektowany był z czytelnością w zamyśle. W Pythonie przynależność kodu do danego bloku oznaczamy za pomocą wcięć, czyli nieco inaczej, niż w większości języków, gdzie zazwyczaj używa się do tego celu klamer lub nawiasów, ewentualnie słów kluczowych jak BEGIN czy END.

W Pythonie zaś liczą się wcięcia, których niepoprawne użycie powoduje błędy uruchomienia. Daje to efekt w postaci tego, że praktycznie każdy poprawny kod Pythona jest w miarę elegancki i łatwo czytelny. Oczywiście, istnieją odstępstwa od tej normy, ale mówię o ogóle kodu, gdzie stosuje się dobre praktyki czy standardy, takie jak PEP8, chociażby, o którym to pomówimy jeszcze później.

Gdy dodamy do tego prostotę i ekspresywność samego języka, to szybko wyjdzie nam na to, że kod w Pythonie jest często po prostu ładny, łatwy do odczytania, zmodyfikowania i przyjazny nowicjuszom.

Owszem, osobom przechodzącym z innych języków może się to wydać, przynajmniej na początku, dziwne, że w Pythonie używamy wcięć zamiast nawiasów czy klamer, ale jest to ładne rozwiązanie moim zdaniem.

Dodatkowo brak w Pythonie jeszcze jednej rzeczy – średniki na końcu wyrażeń nie są konieczne. Mniej pisania o cały jeden znak na linię i czystszy kod.

Oczywiście, czasem stosujemy średniki w Pythonie, niemniej jednak są to sytuacje rzadkie i z góry określone, naprawdę nieliczne.

To chyba też kolejna rzecz, która może dziwić programistów innych języków, chociaż w obecnych czasach nie jest to aż taka rzadka praktyka, by w języku nie były konieczne średniki.

Dlaczego w ogóle jednak czytelność jest ważna? Czas programisty jest drogi, nasze mózgi mają mocno ograniczone zdolności. Dobrze jest, gdy pewne rzeczy od razu widać, gdy nie musimy się nad czymś zastanawiać, bo jest to oczywiste.

Jeśli program jest bardzo czytelnie napisany, to szybciej uda nam się go zrozumieć, a to jest krytyczne w tym, by wykonać zadanie – wbrew pozorom, praca programisty nie polega na ciągłym klepaniu kodu, wręcz przeciwnie.

Osobiście, to większość czasu w pracy spędzam na czytaniu kodu innych ludzi – czy to współpracowników, czy też autorów bibliotek, frameworków a czasem nawet swój własny. Czytelny wygląd dużo ułatwia, a to ważne, bo czytanie i rozumienie kodu jest o wiele trudniejsze niż jego pisanie.

5.8.7 Automatyczne zarządzanie pamięcią

W Pythonie zarządzanie pamięcią odbywa się automatycznie – programista nie ma w tym udziału, robi to za nas sam język za pomocą takiego mechanizmu jak Garbage Collector, dba on o odpowiednie uwalnianie zasobów i pamięci po obiektach, których już nie używamy.

Także nie musimy przejmować się takimi rzeczami jak alokacja i de-alokacja pamięci, jak to ma miejsce np. w C czy C++.

Dlaczego to zaleta? Z racji tego, że niepoprawne zarządzanie pamięcią może doprowadzić do bardzo poważnych błędów, które narażają na szwank cały system, a to, by takowe nie wystąpiły, jest na głowie programisty i często nie jest to rzecz prosta, ba! Czasami banalne konstrukcje związane z alokacją i de-alokacją pamięci, rzeczy, które wydają się oczywiste, mają skomplikowane podłoża, które doprowadzają do poważnych błędów, jeśli źle zrozumiane.

W przypadku Pythona tak nie ma – programista na ogół nie ma nawet dostępu do bezpośrednich operacji na pamięci. Jest to bardzo mądre ograniczenie, przydatne w tego typu języku. Podobnie jest, chociażby, w Javie.

5.8.8 Wspieranie różnych paradygmatów programowania

Są języki, które wspierają mocno w zasadzie tylko jeden paradygmat programowania – jak np. Java, czy Smalltalk, które zaprojektowane są, by ściśle spełniać założenia paradygmatu obiektowego, czy Haskell, który jest językiem funkcyjnym i tylko funkcyjnym, ale są też takie jak Python, które wspierają wiele paradygmatów. O co z tym chodzi, tak po ludzku?

W Javie czy Haskellu masz nieco z góry narzucone to, jak masz 'myśleć' i w jakim kluczu powinieneś realizować rozwiązania pewnych problemów za pomocą kodu. Co to znaczy tak dokładnie, omówimy innym razem.

W Pythonie natomiast masz wolność wyboru. To ty sam decydujesz o tym, które podejście Ci się podoba i którego chciałbyś użyć. Uważam to za zaletę, gdyż ponownie — w jednych sytuacjach lepiej sprawdzają się jedne rozwiązania, w innych drugie. Mając wybór, możesz użyć tego właściwego i już.

5.8.9 Wiele wspieranych platform

Jak już wspomniałem gdzieś wcześniej, Python obsługuje praktycznie dowolną używaną dziś platformę. Windows, Linux, AIX, IBM, macOS, OS/390, z/OS, Solaris, VMS, HP-UX. Co sobie kto zażyczy, prawie na pewno jest. Okej, niby teraz używamy tylko Windowsa, Linuxa i macOS, tak praktycznie mówiąc, ale nawet te systemy są przez niektóre języki niewspierane.

5.8.10 Dojrzałość

Python jest językiem, który powstaje od roku 1991 – obecnie ma prawie 30 lat. Przez ten czas jego ekosystem, narzędzia i biblioteki zdążyły dojrzeć, przejść przez problemy wieku dziecięcego, które niektóre języki mają jeszcze przed sobą.

Znaczy to mniej więcej tyle, że Pythonowi zazwyczaj można ufać. O ile sam programista czemuś nie zawini, to język raczej nas nie zawiedzie, z racji tego, że przetrwał próbę czasu, a większość błędów i rażących bugów, została już dawno wyłapana, załatana.

Czy to znaczy, że to język idealny czy bez błędów? W żadnym razie. Niemniej jednak, z racji tego, że wykorzystywany był/jest on w setkach tysięcy ważnych aplikacji biznesowych, można śmiało stwierdzić, że pewną dozą zaufania warto naszego węża obdarzyć.

5.8.11 Prostota w integracji z innymi językami

Pythona dość łatwo można integrować z innymi językami na różnych platformach. Programy napisane w Pythonie zazwyczaj dość łatwo współpracują z innymi programami, napisanymi, chociażby w odmiennych językach.

Nie każdy język ma tę cechę, gdyż niektóre języki tworzą dość hermetyczną, specyficzną i zamkniętą kulturę, gdzie połączenie czy integracja ich z innymi środowiskami jest nad wyraz lub niepotrzebnie trudna.

Dodatkowym atutem Pythona jest to, że można pisać do niego 'rozszerzenia' w języku C czy C++, które będą działały o wiele szybciej, niż sam Python. Dzięki temu możemy mieć większość aplikacji napisaną w Pythonie – kod prosty, krótki i przyjemny tam, gdzie może być, a akurat jakieś wąskie jej gardło, które musi wykonywać się naprawdę szybko,

w C czy C++. Raczej się tego nie stosuje, ale czasem pojawiają się różne, dziwne powody, dla których warto.

5.8.12 Szybkość tworzenia kodu

Z racji prostoty i mnogości bibliotek w Pythonie, aplikacje, jak i sam kod, można w nim tworzyć wręcz błyskawicznie. To niewątpliwa zaleta, zwłaszcza w czasach, kiedy większość klientów chce, by ich produkt był zrobiony na wczoraj, a terminy zawsze gonią.

Mało tego, zazwyczaj ten zrobiony na szybko kod jest też dość przyzwoitej jakości.

No i faktem jest też, że nawet jeśli nie chcemy wykorzystać Pythona produkcyjnie, to wciąż możemy go użyć, by stworzyć malutkie MVP. Co to MVP? Minimal viable product – czyli taką appkę, która będzie miała minimum funkcjonalności, ale gdzieś tam ktoś już za to będzie chciał zapłacić, bo do czegoś mu się przyda, co ucieszy inwestorów i w ogóle ludzi, bo jest super, mamy MVP, VC sypną, znowu, groszem, kolejna runda finansowania, hajs i hype się zgadza, nasz statek zwany startup'em płynie dalej.

Zapamiętaj ten skrót – MVP to gorący buzzword w zwariowanym świecie STARTAPUFF!

Kończąc dygresję, nawet jeśli nie używamy Pythona produkcyjnie a tylko do MVP, czy tworząc jakiś prototyp po prostu, w Pythonie możemy zrobić to błyskawicznie, sprawdzić, czy dane rozwiązanie działa, jeśli tak, to cóż, zawsze można wersję produkcyjną zaimplementować w innym języku. Jakimś szybszym.

5.9 Wady Pythona

To teraz o tych gorszych stronach.

5.9.1 Python językiem dynamicznie typowanym

Chwila, moment, przed sekundą jeszcze, pisałem, że jest to zaleta. Co jest? Masz gościu rozdwojenie jaźni, czy jak? Co? Kto to powiedział? Halo. A tak serio, to...

Dynamiczne typowanie Pythona to zaleta, która umożliwia nam tworzenie pewnych świetnych mechanizmów, ale i również, w rękach niedoświadczonego programisty, wada.

Pozwala ona bowiem na stworzenie kodu, który spowoduje kompletnie nieoczekiwane, trudne w debugowaniu błędy, którym można by zapobiec w języku statycznie typowanym, w którym to taki kod w ogóle by nie został skompilowany.

W Pythonie, czy innych dynamicznie typowanych językach, takiego mechanizmu nie ma, więc trzeba tutaj nieco uważać, by nie stworzyć błędu, który będzie później trudny do zdiagnozowania i zdebugowania.

Oczywiście obecnie mamy narzędzia, które nam ułatwiają to zadanie, czy nawet upodobniają w pewnym stopniu Pythona do języków statycznie typowanych, gdyż istnieją, chociażby, adnotacje typów, pozwalające nam podawać to, jaki typ powinna mieć zmienna/funkcja.

Niemniej jednak nie jest to obowiązkowy czy konieczny element języka i nie spowoduje on błędu podczas próby uruchomienia aplikacji, a jedynie co najwyżej ostrzeżenie IDE albo analizatora kodu, które można zwyczajnie zignorować.

Zatem dynamiczne typowanie jest nieco jak nóż, z jednej strony możesz użyć go do zrobienia czegoś fajnego, dobrego posiłku na przykład, a z drugiej strony, musisz żyć, ze świadomością, że należy zwracać szczególną uwagę, gdy się z nim obchodzisz, bo możesz się skaleczyć.

Jednakże czy z tego faktu należy rezygnować z korzyści i zastosowań, jakie on ma? Rzucę klasykiem – Nic bardziej mylnego!

Nie wiem jak wam, ale ja to zdanie czytam ze specjalnym akcentem i pewnym głosem w głowie.

5.9.2 Wydajność

Jedno jest jasne – jeśli chodzi o kwestie ściśle wydajnościowe, Pythonowi daleko do miana króla. Ogółem fajny ten Python, taki nie za szybki można powiedzieć.

Oczywiście, obecnie się to zmienia, ale sama natura Pythona jako języka interpretowanego sprawia, że nigdy nie będzie on tak szybki, jak kompilowany do natywnego kodu, C, czy inne języki tego typu. Trzeba się z tym pogodzić i już.

Oczywiście nie twierdzę tutaj, że Python jest bardzo powolny, czy ociężały. Nie. Python nie jest wolny, wręcz przeciwnie – dzięki różnym optymalizacjom poczynionym na przestrzeni lat, Python naprawdę zyskał na szybkości i dziś śmiało stwierdzam, że jest to język wystarczająco szybki, ale mocno należy zaznaczyć, że nie jest to język najszybszy. I tyle.

A jak już o wydajności mówimy to i o rozmiarach wspomnę – wymagania sprzętowe Pythona sprawiają, że na niektórych platformach go po prostu nie uruchomimy. Są pewne obszary świata embedded, gdzie króluje C czy Assembly, Python tam nie istnieje i nie ma co z tym dyskutować.

Oczywiście są też projekty jak RaspberryPi, gdzie faktycznie, Python również rządzi wszystkim.

Czyli jeśli chcesz pisać wysoce wydajne gry z piękną grafiką, czy też może wielowątkowe aplikacje, które w rzeczywistym czasie obsługują ogromne ilości obliczeń, albo może malutkie mikro-kontrolery, to cóż, Python nie jest raczej zbyt dobrym wyborem w takim razie.

W innych przypadkach Python sobie poradzi i nie trzeba przejmować się szybkością wykonania/zasobami. Dlaczego? Otóż żyjemy w takich czasach, że czas serwera jest znacznie tańszy niż czas dewelopera. To znaczy lepiej, żeby język był może i kapkę wolniejszy, ale za to, jeśli pisze się w nim znacznie szybciej, to go wybieramy. Tak jest po prostu taniej, lepiej, zdrowiej.

Nie oznacza to, że mamy tu przyzwolenie na to, by pisać byle jaki kod, który działa siermiężnie i wolno, ale działa. Absolutnie! Trzeba szanować czas użytkownika, zasoby sprzętowe, które mamy i kilka innych rzeczy. Oszczędzaj RAM gdziekolwiek jesteś. Jak we wszystkim – należy znać umiar i granice. Chodzi mi tutaj bardziej o sytuacje, teoretyczną, gdzie mamy jakiś kod obsługujący zapytanie do serwera.

Przejście zapytania przez sieć zajmuje, załóżmy, sekundę. Wykonanie i zwrócenie odpowiedzi przez Pythona zajmie około 0,1 sekundy. Potem znowu powrót do użytkownika, czyli kolejna sekunda. Łącznie 2,1 sekundy. Możemy przepisać ten kod w innym języku, załóżmy, Javie – kod będzie kilka razy dłuższy, pisanie zajmie go więcej czasu, ale za to wykona się, powiedzmy, 10 razy szybciej. Czyli użytkownik, zamiast poczekać 2,1 sekundy,

poczeka 2,01 sekundy, gdyż zazwyczaj to nie sam serwer i kod naszej aplikacji, jest wąskim gardłem, a np. baza danych, połączenie sieciowe czy dysk.

Czy ma to sens w większości przypadków? Przeskok z 2,1 do 2,01s? Sami sobie odpowiedzcie. O takie sytuacje mi chodzi – wtedy nie ma sensu zazwyczaj bawić się w optymalizacje i Python jest po prostu wystarczająco szybki.

Tak to przynajmniej się ma w znacznej większości projektów, gdyż tych, które nie mogą sobie pozwolić na to minimalne zwolnienie, jest niezbyt dużo. Zresztą ty – jako początkujący programista, raczej nawet na oczy takich projektów na starcie kariery nie ujrzysz, bo nie pora na to.

5.9.3 GIL

W Pythonie mamy coś takiego jak GIL – Global Interpreter Lock. Nie będę się tutaj rozwodził nad szczegółami tego mechanizmu, wystarczy, że będziesz wiedział, iż to przez niego Python nie do końca jest idealnym wyborem, kiedy przyjdzie nam rozmawiać o wielowątkowych aplikacjach, gdyż tylko jeden wątek w danym momencie może mieć dostęp do interpretera w procesie, bo GIL blokuje resztę.

A o co chodzi z tą wielowątkowością i tak dalej? W skrócie i dużym uproszczeniu to chodzi tu o wykonywanie wielu rzeczy jednocześnie, najczęściej za pomocą wielu rdzeni procesora, by przyśpieszyć działanie jakiejś aplikacji.

Bo nie wiem, czy kojarzysz, ale masz w komputerze coś takiego jak CPU – tak zwany procesor. Ten procesor odpowiada za większość obliczeń, kalkulacji i wykonywanie twoich poleceń, tak w dużym skrócie.

Podczas rozwoju technologii, doszliśmy w pewnym etapie do momentu, kiedy trudno było już sprawić, żeby jeden rdzeń był szybszy. Zatem, żeby to wszystko działało jeszcze szybciej, a ty mógł/mogła odtwarzać pięć aplikacji w tle, obecnie procesory mają po kilka rdzeni.

Rdzenie to takie jakby małe procesory wewnątrz procesorów. Wyobraź sobie pracownika. 1 rdzeń = 1 pracownik. I wracając do tego wcześniej – pracownik jak to pracownik, ma ograniczoną wydolność, bo ogranicza go np. fizyka. No jeden człowiek, nieważne jak silny, nie jest w stanie przenieść więcej worków betonu niż X na godzinę. My, po pewnym czasie doszliśmy właśnie do tego momentu, że technologicznie stworzyliśmy pracownika, czyli procesor, co to do tego X się zbliżył, powiedzmy.

Pojawił się zatem problem, bo mamy wydajność pracownika X. Mamy jednego pracownika na budowie, chcemy szybciej skończyć pracę, jak możemy to zrobić, skoro wyciągnięcie więcej niż X worków na godzinę, z tego jednego pracownika, będzie trudne lub niemożliwe na chwilę obecną? Możemy spróbować sprawić, by był jeszcze bardziej wydajny i np. zafundować mu dobrą kurację sterydami, co by silniejszy się zrobił, albo kokainą/amfą go odżywiać, sprawiając, że wydajność wzrosłaby o te 5%, ale koszt tego przedsięwzięcia byłby zupełnie niewspółmierny do uzyskanych rezultatów. Co zatem zrobić? Zatrudnić więcej pracowników. Wtedy, kiedy do pracy zaprzęgniemy kilku robotników, mogą być oni nawet słabsi od tego jednego, ale będzie ich kilku. Łączna moc przerobowa wzrośnie.

Plus minus tak wygląda sytuacja z procesorami i tym, że są one obecnie wielordzeniowe.

Tu wchodzi Python, który jest takim trochę upośledzonym brygadzistą. Dobrze sobie radzi z zarządzaniem 1 pracownikiem, ale jeśli przyjdzie mu ogarniać np. 4, to ma już pewne ograniczenia, których trzeba być świadomym.

Co prawda da się to teraz w miarę łatwo obejść, ale wciąż coś takiego pozostaje i trzeba nauczyć się z tym radzić, bo można wpaść w pułapkę.

5.9.4 Wysoka ekspresywność

Ponownie, coś, co jest zaletą, jest również troszkę wadą. Dlaczego? Python ukrywa pewne rzeczy przed tobą, programistą, co powoduje, że nie zawsze wiesz, jak jest to zrobione 'od podszewki'. Nie jest to do końca dobre, bo czasami przydaje się wiedzieć, jak pewne rzeczy zostały zaimplementowane i dlaczego akurat tak.

Sporo to wyjaśnia. Prostym tego przykładem jest częste pytanie – dlaczego indeksujemy listy czy tablice od 0? Jeżeli jesteś programistą C/C++, najprawdopodobniej znasz odpowiedź.

Programiście języków wysokopoziomowych zaś nie zawsze ją znają. Ba! Wręcz powiedziałbym, że rzadko. Nie bój się, jeśli nie wiesz, ten temat poruszymy w książce, ale nieco później.

Jest to jednak mała cena, jaką trzeba za płacić w porównaniu z tym, co ta ekspresywność i wysoka abstrakcja oferuje.

Po prostu to problem łatwy do naprawienia – wystarczy trochę chęci, by poczytać kapkę więcej. A czas, który poświęcimy na zgłębienie tych różnych tematów, jest o wiele krótszy niż czas, który poświęcilibyśmy, pisząc swój program w języku o niższym stopniu abstrakcji/ekspresywności.

5.9.5 Python nie istnieje w świecie mobile

Aplikacje mobilne i Python to raczej dwa odmienne światy. Tak po prostu i już. Na pewno istnieją projekty próbujące coś w tym zakresie wskórać, ale nie ma co się łudzić na to, że znając tylko Pythona, stworzymy fajną appkę na Androida.

Jak ktoś ci mówi inaczej, to lepiej olej typa, bo Python mu wszedł za mocno i bredzi.

5.9.6 Zbytnia wygoda

Często może być tak, że po tym, jak zaczniesz pisać w Pythonie, przesiadka na inne języki, gdzie pewne rzeczy musisz zrobić zupełnie inaczej, jest troszkę bolesna. To również potencjalna wada Pythona.

Piszesz sobie szczęśliwie swoje programy w Pythonie, sporo rzeczy robisz jedną linijką kodu, jest fajnie pięknie, ale nagle przychodzi ci napisać coś w Javie i następuje brutalne zderzenie z rzeczywistością, które powoduje, iż lądujesz w otchłaniach ciemności, rozpaczy i depresji, twoje życie traci sens a żona musi rano zrzucać cię z łóżka, żebyś wstał. Nie no, żartuje, pisanie w Javie nie jest takie złe, nie mam nic do języka. Po prostu mało który język jest tak fajny, jak Python.

5.9.7 Tyle

Pisząc o wadach i zaletach Pythona, starałem się być w miarę obiektywny. Oczywiście jest to niezbyt możliwe z racji tego, że to książka jest o Pythonie, a ja sam jestem jego entuzjastą. Niemniej jednak uważam, że udało mi się przestawić Ci mocne i słabe strony

Pythona, dzięki czemu możesz zdecydować czy warto się go uczyć. Moim zdaniem jak najbardziej tak!

Poza tym, kurka, jak już masz tę książkę, to korzystaj i się ucz!

5.10 Kto używa Pythona?

W tym wypadku lepiej byłoby zapytać o to, kto Pythona nie używa.

Dla przykładu jednak podam kilka mniej lub bardziej znanych firm, które z Pythona korzystają, są to: 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.

Cały YouTube w zasadzie stoi (stał) na Pythonie. W Google mówi się: 'Tam, gdzie możemy – Python, tam, gdzie musimy – C++' (podobno).

Całkiem sporo, prawda? No cóż, nic dziwnego, z racji tego, że Python, według indeksu TIOBE jest obecnie 3. najpopularniejszym językiem programowania na świecie. Nad nim są już tylko Java i C. Dodatkowo Python co roku zdobywa coraz większą popularność i rośnie w siłę. Jak to ktoś kiedyś powiedział, niekoniecznie mądry, `tej siły już nie powstrzymacie`.

Poniżej widzicie tabelkę z indeksem TIOBE, który to, powiedzmy, jest standardem w świecie programowania, jeśli chodzi o mierzenie popularności pewnych technologii, trendów i tak dalej.

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%

Jak widać, Python spokojnie pokonuje takie języki, jak C#, PHP, JavaScript, SQL, R, czy Ruby. Tego ostatniego w zasadzie 8-krotnie.

Przyszłościowy ten wąż.

Naprawdę uważam, że obecnie znajdujemy się w punkcie, lub dość blisko takiego punktu, gdzie w Pythonie zostało napisane zbyt dużo kodu, by w przyszłości został on wyparty. Zerknijmy na to, jak Python zyskuje popularność na przestrzeni lat.

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

Imponujące, prawda? 20 lat temu Python nie znajdował się nawet w czołowej 20. W 2004 już 6. A teraz? Na podium.

Popatrzcie też na starego potwora - COBOLa– mimo tego, że nie był to jakoś bardzo przemyślany czy piękny język, to do dziś jest on rozwijany i używany z racji tego, ze swego czasu sporo softu się w nim pisało, zwłaszcza dla banków i najzwyczajniej w świecie pozbycie się wszystkich systemów działających na podstawie tego języka i przepisaniu ich na coś nowszego, byłoby zbyt kosztowne.

Czyli mimo tego, że COBOL jest językiem dość specjalistycznym, o wąskim zastosowaniu, to wciąż jest używany, a przecież pierwsze wzmianki o COBOLU pochodzą z roku 1959, czyli na upartego, jeśli ktoś chce, może dziś programować w technologi sprzed w zasadzie 60 lat i znajdzie w tym pracę.

Do przyjemnych to raczej nie należy, ale cóż. Można? Można.

Jak już wcześniej wspomniałem o Pythonie – tej siły już nie powstrzymacie. Python przyszedł, rozgościł się i na razie nigdzie się nie wybiera, a prawdopodobnie zostanie na dobre.

Tego ostatniego pewien nie jestem, oczywiście jako programista Pythona będę się starał, by tak było – dokładając swoją cegiełkę i tworząc jak najwięcej kodu w Pythonie, niemniej jednak (jeszcze) gwarancji dać nie mogę na to, że zostanie on z nami na zawsze.

Mogę ją dać na co innego – że w przeciągu następnych 15 lat Python nie zostanie wyparty z ogólnie pojętego programowania popularnego i nie będzie problemem, by znaleźć pracę, znając ten język, także, jeśli o to się boisz, to masz moje słowo, że tak nie będzie.

Zatem idąc w Pythona, robisz dość mądry ruch, jeśli chodzi o twoją karierę. To zrób to, posłuchaj Górskiego, i do dzieła.

5.11 Python w porównaniu z...

5.11.1 Javą/C

Wrzucę te dwa języki do jednego worka, ku oburzeniu niektórych. Zaraz mi się tu zlecą fanboje z hejtem, ale co tam.

Niemniej jednak zacznijmy od tego, że oba te języki wspierane są/kierują nimi wielkie korporacje. Python nie. Dla jednych to wada, dla innych zaleta. Python jest też zdecydowanie prostszy w nauce od tych dwóch języków, bez dwóch zdań.

Jest też o wiele bardziej ekspresywny – kod jest zazwyczaj krótszy, o wiele krótszy.

Niestety, często też wolniejszy od jednego, jak i drugiego.

Python jest również mniej popularny od Javy, która jest obecnie na samym szczycie i to bez dwóch zdań, ale za to też wyraźnie popularniejszy niż C#.

Do tego Java/C# jest częściej wykorzystywana przez duże korporacje aniżeli Python. Ogromne projekty to często domena Javy czy C#. Dla mnie kolejna wada.

5.11.2 Perlem

Cóż, mało kto obecnie używa Perla, ale z racji tego, że niegdyś często porównywano te dwa języki, to wspomnę i o nim. Osobiście nigdy nie pisałem w Perlu, lecz zdarzało mi się widzieć kod w nim napisany.

Ewidentnie bywa on czasami niezbyt czytelny, a ilość stosowanych w Perlu nawiasów potrafi człowieka przytłoczyć. Python jest też obecnie znacznie popularniejszy niż Perl. Przytłaczająco popularniejszy. Nie będę się zatem dalej nad tym porównaniem rozwodził, bo to sensu nie ma.

5.11.3 C

C jest językiem 'drobnym' - jego core jest niewielkie, ale dające niesamowite możliwości, niemniej jednak wciąż jest to język opatrzony pewnymi ograniczeniami.

C króluje w miejscach, gdzie Python nie ma racji bytu – sterowniki, embedded, aplikacje, gdzie naprawdę krytyczna jest szybkość lub też wydajność. W C programista jest w stanie co do bajtu zarządzać pamięcią wykorzystywaną przez jego program. W Pythonie? Zapomnij o czymś takim.

Python jest zatem od C o wiele wolniejszy, zajmuje więcej pamięci. W zamian za to zyskujemy o wiele większą ekspresywność, szybkość pisania kodu i bezpieczeństwo – Python jest po prostu prostszym językiem w porównaniu do C, lecz raczej nigdy się ich nie porównuje. Dlaczego?

Gdyż domyślna implementacja Pythona napisana jest właśnie w... C to raz, a dwa, służą one zupełnie innym zadaniom. Są to języki, powiedziałbym, komplementarne, a nie przeciwne, gdyż dobrze uzupełniają się w swoich wadach.

Dużo bibliotek, które dla szybkości, pisze się w C, ma swoje wrappery, napisane w Pythonie właśnie. Co to te wrappery? Pomyśl sobie o swoim samochodzie. W środku ma on prawdopodobnie dość skomplikowany komputer. Ty, jako użytkownik, jesteś w stanie z nim prosto obcować za pomocą różnych guziczków, przycisków, menu i tak dalej – łatwa rzecz. Samodzielnie jednak nie jesteś w stanie modyfikować tego, jak ten system konkretnie działa, co robi w jakiej sytuacji itd., ale nie przeszkadza to jakoś bardzo, bo w 99% nie ma takiej potrzeby, a dla tego 1% brak sensu, by porzucać samochód i zacząć chodzić na piechotę. A jak już potrzebujesz zmienić coś w tym, jak ten komputer działa... Takie rzeczy może zrobić mechanik za pomocą konkretnych narzędzi i modyfikacji softu.

Python i jego programista jest tutaj tobą, a C to komputer pokładowy samochodu.

Tam, gdzie potrzebna jest niesamowita wydajność, którą pokonać może chyba tylko Assembly, można użyć C, tam, gdzie ważniejsza jest czytelność i szybkość produkcji kodu, można użyć Pythona. Super kombo. Swoją drogą to spora część programistów Pythona zna również C. To chyba naturalna kolej rzeczy.

Koniec tych porównań, bo można by tak bez końca. W tej chwili sam powinieneś dobrze znać silne strony Pythona, jak i te słabe.

5.12 Alternatywne implementacje

Jak wspomniałem gdzieś po drodze, Python to język, który posiada swój interpreter służący do uruchamiania programów napisanych w tymże języku. Czyli to taki program

do odpalania programów. Incepcja. We need to go deeper.

Standardowo ten interpreter napisany jest w języku C – czyli CPython. Istnieją jednak inne implementacje tego interpretera, które czasem mają inne priorytety niż CPython, rozszerzają jego funkcjonalność czy po prostu są napisane w innym języku, bo tak i już, bo można. Kto bogatemu zabroni.

Oczywiście, nie wszystkie z nich spełniają w pełni założenie pełnej kompatybilności z całym standardem Pythona, niektóre nieco od tego odbiegają, nie są czymś praktycznym, a raczej takim proof-of-concept – po prostu czymś zrobionym tylko dlatego, że można.

5.12.1 Stackless Python

Zacznę może od implementacji zwaną Stackless Python, która bazuje na CPythonie, z tym że skupia się ona na współbieżności, wielowątkowości, czyli po prostu rozszerza CPythona, poprawiając pewne rzeczy, w których on akurat nie jest może zbyt dobry domyślnie. Takie tam różne tematy związane z GILem i spółką.

5.12.2 Cython

Cython to nic innego jak, w zasadzie, kompilator Pythona do... C. Piszesz kod w Pythonie, a w efekcie otrzymujesz program działający z prędkością programu napisanego w C. Nieźle. Nie jest to co prawda kompletna implementacja tego języka a, jak wspomniałem, 'tłumacz' do C, ale o takich narzędziach też napiszę.

5.12.3 PyPy

Python napisany w... Pythonie. Incepcja. We need to go deeper. Z drugiej strony nic dziwnego, w końcu, z tego, co kojarzę, to od pewnej wersji, kompilator Go też jest napisany w Go. Podobnie z innymi nowoczesnymi językami.

Jest to jedna z trzech implementacji, które są raczej praktycznie w 100% kompatybilne ze standardową implementacją Pythona. Cóż takiego ciekawego jest w tej implementacji?

Otóż fakt, że posiada ona JIT – kompilator Just In Time, znany, chociażby z Javy. Co to takiego?

Kompilator JIT to taki kompilator, który działa dopiero po uruchomieniu danego programu i kompiluje nasz akurat używany fragment kodu w locie, często do formy natywnej dla danego CPU to raz, a dwa, że kompilator JIT ma zazwyczaj informację o środowisku uruchomieniowym, co pozwala mu poczynić lepsze optymalizacje względem statycznego kompilatora, który tych informacji nie posiada. O co tu chodzi?

CPython 'kompiluje' nasz kod Pythonowy, do bajtkodu zrozumiałego dla maszyny wirtualnej Pythona, która to następnie wykonuje zadany bajtkod, który skompilowany został przed uruchomieniem skryptu. PyPy robi to 'w locie', mając możliwość czynienia lepszych optymalizacji niż statyczny kompilator.

Dlaczego zatem PyPy nie jest domyślną implementacją, skoro jest szybszy, niby lepszy? Otóż by kod został wykonany szybciej, musi być spełnione kilka warunków.

Przede wszystkim PyPy i jego JIT potrzebują chwili, by się 'rozgrzać' - czyli nasz skrypt musi wykonywać się przynajmniej kilka sekund, by mieć możliwość zyskania przyśpieszenia w związku z użyciem PyPy.

Drugim wymogiem jest to, że wąskim gardłem naszego programu muszą być Pythonowe instrukcje, a nie np. instrukcje zawarte w jakiś modułach wewnętrznych napisanych w C.

Jeśli te wymogi są spełnione, to bardzo prawdopodobne jest, że użycie PyPy przyśpieszy wykonanie naszego programu i być może nawet zmniejszy zużycie pamięci. W innych przypadkach... Cóż... Prawdopodobnie nie zyskamy zbyt dużo na użyciu PyPy. O PyPy wypowiada się nawet sam twórca Pythona – Guido van Rossum.

> "Jeśli chcesz przyśpieszyć swój kod, to prawdopodobnie powinieneś użyć PyPy."
> Guido Van Rossum – twórca Pythona

Dodatkowo PyPy domyślnie implementuje np. cechy z Stackless Python, czy Sandboxing, który obecnie jest jeszcze raczej prototypem niż produkcyjną implementacją, ale wciąż.

5.12.4 IronPython

Python i C#. Co go wyróżnia? Przede wszystkim to, że nie ma GIL'a, to, że można korzystać z biblioteki .NETowej, łatwo można go osadzać w aplikacjach .NET. Druga z głównych alternatywnych implementacji, która raczej spełnia standard Pythona.

5.12.5 Jython

Python na wirtualnej maszynie Javy. Zalety? Łatwa integracja z programami Javowymi, możliwa kompilacja programów Pythonowych do klas Javowych. Programy uruchamianie na wirtualnej maszynie Jythona mają pełen dostęp do klas i API Javy. Można go też kompilować statycznie i tworzyć servlety, applety, beany. Jython również jest wielowątkowy w prawdziwym tego słowa znaczeniu, czyli nie musimy się tutaj martwić GIL'em. Trzecia, ostatnia, praktycznie w 100% kompatybilna implementacja.

5.12.6 Brython

Brython to wynalazek, który sprawia, że twój kod, napisany w Pythonie, zadziała w... przeglądarce, po stronie klienta poprzez tłumaczenie Pythona do JavaScriptu. Aktywnie utrzymywany i w miarę aktualny. Ciekawy projekt.

Z innych implementacji wspierających kompilację do JS, istnieją: RapydScrypt, Transcrypt.

5.12.7 MicroPython

Mówiłem, że Python na mikro-kontrolerach to niezbyt, prawda? Cóż, ten projekt ma na celu to zmienić. Podobnie jak PyMite.

5.12.8 CLPython

Python zaimplementowany w język Common Lisp. Podobnie jak Jython czy IronPython, CLPython może mieszać kod Pythonowy z kodem języka, w którym został napisany, ma dostęp do jego bibliotek.

5.12.9 TinyPy

Python w 64Kb kodu. Bo tak.

Oprócz tego istnieją implementacje Pythona w Haskelu, PHP, JavaScripcie, Rubiniusie i innych różnorakich wynalazkach jak połączenie LOLCODU z Pythonem czy Like, Python.

5.13 Podsumowanie

Starczy już tej całej teorii i gadania, streśćmy zatem to, o czym do tej pory powiedziałem.

Python jest ogólnego przeznaczenia dynamicznie typowanym językiem interpretowanym, którego zaletami są ekspresywność, społeczność, ilość gotowych rozwiązań, szybkość tworzenia aplikacji oraz czytelność kodu w nim tworzonego, łatwość w nauce.

Dobrze sprawdza się w webdevie, skryptach, zastosowaniach naukowych, związanych z Data Science czy podobnymi dziedzinami. Nieco gorzej radzi sobie w aplikacjach wielowątkowych, tych, które wymagają bardzo wysokiej wydajności czy też w środowisku z mocno ograniczonymi zasobami. Dodatkowo Python nie istnieje w świecie tworzenia appek mobilnych, gier czy desktopowych.

Niemniej istnieją rozwiązania na różne bolączki Pythona związane z tymi problemami, jedne lepsze, inne gorsze, ale są. Nie zmienia to faktu, że jest on lepiej przystosowany do pewnych zadań, niż do innych i warto o tym pomyśleć, przy wyborze technologii do danej aplikacji.

5.14 Pytania

1. Na jakiej wersji Pythona będziemy się skupiać i dlaczego?
2. Podaj przykłady, przynajmniej 3, obecnie popularnych zastosowań Pythona.
3. W jakich zastosowaniach Python nie jest zbyt dobry?
4. Podaj kilka zalet Pythona, jak i jego wad. Kto z niego korzysta? Do czego można go użyć?
5. Podaj przykłady alternatywnych implementacji Pythona – poza Cpythonem.
6. Jakie są główne założenia Pythona?
7. Jak Python wypada w porównaniu do Javy? Główne różnice/cele.
8. Czy Python działa na wielu platformach? Jeśli tak, to na jakich? Wymień 3+.
9. Jakim językiem jest Python, statycznie czy dynamicznie typowanym? Co to znaczy?
10. W jaki sposób ogranicza nas GIL – tak w skrócie?

6 Ustawiamy środowisko

Przejdźmy wreszcie do czegoś konkretnego – ubrudźmy sobie nieco ręce, bo do tej pory tylko gadam i gadam. Zaczniemy od ustawienia środowiska/zainstalowania Pythona oraz czegoś do edycji tekstu. Tylko moment, czego? Istnieją dwie opcje – prosty edytor tekstu lub IDE. Co wybierzemy my?

6.1 W czym pisać kod na początku?

W czym pisać kod na początku? IDE/Edytor tekstu? PyCharm? Sublime? VS Code?

Gdy pomyślę sobie o moich pierwszych dniach nauki programowania, jedną z wielu rzeczy, które ciągle mieszały mi w głowie i przyniosły sporo kłopotu w ich rozstrzygnięciu, był wybór środowiska, w którym będę pisał swoje pierwsze programy.

Nie ma co się tu jednak co dziwić — teraz praktycznie do każdego języka istnieje co najmniej kilka darmowych IDE, do tego jeszcze dochodzą płatne, mamy jeszcze edytory tekstu i tak dalej. Łatwo można się zagubić. Ja tak przynajmniej zrobiłem i kilka dni, zamiast się uczyć programowania, siedziałem i zastanawiałem się nad wyborem środowiska.

6.1.1 IDE? Edytor tekstu?

No to jak to w końcu jest? IDE czy edytor tekstu? A czym w ogóle jest IDE? Tego dowiedz się sam — jako informatyk twoim podstawowym zadaniem jest przetwarzanie informacji. Wyszukaj sobie zatem w Internecie znaczenie tego skrótu, wyszukaj różnice między IDE a edytorem tekstu.

Nie podam ci tego na tacy, bo umiejętność szukania informacji jest kluczową cechą, która składa się na dobrego informatyka/programistę, którą trzeba ćwiczyć na początku i to będę się starał w tobie wykształcić na przestrzeni stron tej książki.

Co zatem wybrać? Odpowiedź może cię zadziwić — brzmi ona: bez większego znaczenia. Już tłumaczę.

To, czy wybierzesz CodeBlocks, VisualStudio czy Atom'a, nie ma, moim zdaniem, większego znaczenia, gdyż to są tylko narzędzia. Narzędzia w rękach programisty i to od niego zależy, jak dobrze zostaną te narzędzia wykorzystane. Podobnie jak z językami.

Nic ci nie da nawet najlepsze IDE, skoro nie będziesz umieć go obsłużyć. Z drugiej strony, jeśli dobrze opanujesz obsługę pozornie prehistorycznych, jak na dzisiejsze standardy, programów jak np. Vim czy Emacs, możesz zdziałać cuda.

Dlatego też zalecam, by nie przywiązywać zbytniej uwagi do tego, jakie środowisko się wybierze, bo praktycznie każde można przystosować i sprawić, by pracowało się w nim wyśmienicie.

Mimo tego można jednak chyba nakreślić jedną ze ścieżek, którą warto obrać i która według mnie jest dość rozsądna. Jaka to ścieżka?

6.1.2 Podstawy

Niezależnie od tego, czy korzystamy z IDE, czy jakiegoś edytora tekstu dobrze jest wiedzieć, jak to tam wszystko działa od podszewki, co powtarzam już któryś raz i jeszcze niejednokrotnie powtórzę.

Żeby nie było tak, iż ktoś bez swojego IDE nie jest w stanie skompilować kawałka kodu C++ czy odpalić skryptu Pythona z poziomu konsoli. Takie podstawowe rzeczy należy poznać, bo pozwalają one lepiej zrozumieć, jak tworzy się programy i jak to wszystko działa, te informacje czasem są zwyczajnie niezbędne.

Poza tym, na początku funkcja podpowiadania kodu powinna być wyłączone. Przepisując nazwy instrukcji, metod czy klas szybciej się ich nauczymy. Ja przynajmniej tak miałem. Korzystając z auto uzupełniania, często wystarczyło, żebym wpisał dwie, trzy litery, tab i pyk — gotowe.

Problem pojawiał się, gdy auto uzupełniania zabrakło, bo np. musiałem napisać coś na szybko w zwykłym edytorze tekstu/skorzystać z nieswojego komputera lub, o zgrozo, poszedłem na rozmowę rekrutacyjną, gdzie trzeba było napisać kod na tablicy, pisakiem. Nie będę mówił, jak głupio przez to kilka razy wypadłem.

Wtedy nagle wszystko z głowy mi wyleciało i zapomniałem połowy poleceń, które podobno tak dobrze znałem. Dlatego też na samym początku jednak zrezygnowałem z automatycznego uzupełniania na rzecz ręcznego pisania.

Zatem na początku lepiej zapomnij o podpowiadaniu składni, zwłaszcza jak wybierasz się za jakiś czas na rozmowę rekrutacyjną. Chociaż wątpię, skoro czytasz tę książkę.

Później, kiedy już zapamiętałem dostatecznie słowa kluczowe, instrukcje, ponownie do niego wróciłem. Na sam początek jednak odradzałbym auto uzupełnianie.

6.1.3 Jak to u mnie wyglądało

Jakie środowisko ja wybrałem?

Powiem szczerze, że wiele różnych wypróbowałem, idąc mniej więcej taką ścieżką podczas nauki Pythona:

Na początku pisałem w przeglądarce, robiąc kurs na jednej z interaktywnych akademii programowania — nie potrzeba mi było żadnego programu ściągać, jednak później, gdy już chciałem zacząć pisać coś swojego na dysku, to zajrzałem w neta i znalazłem różne wątki, gdzie zazwyczaj polecano PyCharma i kilka innych popularnych IDE. Ściągnąłem je zatem. Na krótko jednak.

Były one dla mnie zbyt przytłaczające — ogrom opcji, złożoność. Poza tym mój leciwy laptop słabo radził sobie z wymaganiami sprzętowymi takich IDE. Praca w nich była w sumie niezbyt możliwa.

Przerzuciłem zatem się na SublimeText, gdzie zagościłem na troszkę dłużej, powoli oswajając się z konsolą i tym, że niektóre rzeczy muszę zrobić sam. Po pewnej chwili zaczęło mi czegoś w Sublimie brakować, więc znów zacząłem poszukiwania — trafiłem na Vim'a i z niego przez jakiś czas korzystałem.

Wspaniałe narzędzie. Szwajcarski scyzoryk — wszystko z nim można zrobić. Może być to lekki edytor tekstowy albo rozbudowane IDE w zasadzie, przy odpowiedniej ilości plug-inów. Przypadł mi on do gustu możliwością konfiguracji, magią, którą można w nim wyczyniać.

Na początku było trudno, ale potem się już troszkę przyzwyczaiłem. Niemniej jednak Vim'a raczej odradzam jako pierwszy edytor tekstowy, jest on mimo wszystko troszkę inny w obsłudze niż standardowe programy.

Vim zmieniłem na SublimeText do krótkich i prostych skryptów, a całą resztę aplikacji piszę jednak w PyCharm. Oferuje on zbyt dużo przydatnych opcji, by z niego nie korzystać

to raz, a dwa, że stał się on niejako standardem w naszej branży i naprawdę niewiele jest firm, które z niego nie korzystają.

W Sublimie zaczęło mi czegoś brakować, zatem... Powrót do Vima+PyCharm. I to jest mój ostateczny stack teraz od kilku lat.

Na potrzeby tej książki w zupełności wystarczy Ci SublimeText, potem ewentualnie przeskoczymy na PyCharma. Przejdźmy do instalacji, skoro już wiemy, co będzie nam potrzebne.

6.2 Windows?! Linux?! macOS?! Co wybrać?

Bez znaczenia.

Zaraz zjawią się tutaj albo gdzieś indziej, fanatycy jednego czy drugiego systemu i usilnie będą próbować wmówić Ci, że to akurat system wybrany przez danego gościa jest najlepszy i tylko on się nadaje do użytkowania/programowania/czegokolwiek.

Nie zajmuj sobie nimi głowy. System jest tylko narzędziem i to od ciebie zależy, czy sprawnie je wykorzystujesz, czy nie. O ile nie interesują cię języki, które w jakiś sposób ograniczają swobodę wyboru, bo, dla przykładu, chcąc pisać aplikacje na iOS'a w Swifcie, raczej nie będziesz siedział na Windowsie, analogicznie pisząc soft w czystym C#, też raczej nie będziesz używał macOS'a. Prosta sprawa.

Nas to jednak nie dotyczy. My prości Pythonowcy jesteśmy. A on sobie w miarę fajnie śmiga wszędzie.

Dobrze jest jednak mieć, chociaż podstawowe rozeznanie, jeśli o Linuxa chodzi. Dlaczego? Praktycznie każdy serwer, na którym hostujemy nasze aplikacje w sieci, jest postawiony na Linuxie. Prędzej czy później się z nim zetkniesz, a jeśli wcześniej będziesz z nim nieco znajomy, to nie będzie to bolesne zderzenie z rzeczywistością, gdzie przywalisz głową w mur, a raczej spotkanie przy piwie z dobrym kumplem, którego dawno nie widziałeś. To raz. Dwa, ogółem pewne rzeczy na Linuxie masz za od ręki, gdzie na Windowsie musisz sam się z niektórymi rzeczami kłopotać, instalować itd. Czy to znaczy, że masz instalować nowy system, żeby poznać Linuxa, inaczej żyć się nie da?

Absolutnie nie. Wystarczy ci do tego wirtualna maszyna z dowolną dystrybucją. Ja od siebie polecam Ubuntu, mimo że nie przepadam za Canonical, oraz drugą opcję, czyli Manjaro. Oba są Linuxami, ale z pewnymi różnicami. Osobiście, jak korzystam z Linuxa, to właśnie z Manjaro. Na desktopie przynajmniej. Bo na serwerach Manjaro raczej odpada.

Wirtualka, co, jak, gdzie? Przeczeszcie spis treści, w tej książce powinieneś znaleźć odpowiedź, tylko pewnie będzie ona nieco dalej.

Mało tego, od niedawna dostępna jest jeszcze inna. WSL. Windows Subsystem for Linux. Windowsowy podsystem dla Linuxa. To taka Microsoftowa zabawka, która umożliwia ci posiadanie zintegrowanego z twoim Windowsem, Linuxa. Linux w Windowsie od Microsoftu. Bum. Zalety to łatwość instalacji, łatwiejsza integracja etc. Ogółem polecam.

Podsumowując tak krótko, zarówno Linux, jak i Windows mają swoje wady i zalety, są tylko narzędziami. Ja, osobiście, jednak korzystam (wreszcie), z Linuxa jako hosta. Jeśli siedzisz na Linuxie to spoko, jeśli nie to koniecznie zainstaluj sobie WSL. ## Instalacja na Windows Zazwyczaj w książkach do Pythona znajdziemy opis instalacji Pythona krok

po kroku, pomocne zrzuty ekranu i tak dalej. Cóż, nie tutaj. My zainstalujemy sobie potrzebne nam rzeczy w nieco nietypowy, jak na Windows, sposób. Za pomocą konsoli. Nie, nie oszalałem.

Nie wiem, czy jesteś świadom, ale użytkownicy Linuxa zazwyczaj instalują programy w nieco odmienny sposób, niż Windowsiarze.

Otóż każda popularna dystrybucja Linuxa zawiera w sobie tak zwany menadżer pakietów. Możesz o tym myśleć, jak o 'programie', które zarządza wszystkimi oficjalnie wspieranymi i dostępnymi programami dla danej dystrybucji. Zrozumiałe, prawda?

Dzięki rzekomym menadżerom pakietów Linuxiarze mogą robić coś fajnego – większość programów można na Linuxie zainstalować za pomocą jednego polecenia, podobnie sprawa się ma z ich aktualizacją, aktualizacją całego systemu, czy usuwaniem 'programów'. Na Windowsie jest troszkę inaczej. Jak dla mnie sposób zarządzania pakietami Linuxa jest wygodniejszy i pewnie wiele osób się ze mną zgodzi w tym momencie.

Niestety, domyślnie, Windows nie posiada sensownego menadżera pakietów, jednakże nie bez powodu użyłem tutaj zwrotu 'domyślnie'. Otóż powstało bardzo fajne narzędzie, zwane chocolatey, które umożliwia nam uzyskanie takiej funkcjonalności, jaką dają pacman czy apt.

W skrócie, dzięki niemu instalacja sporej części programów jest zredukowana do:

choco install nazwa_programu

i tyle. Wygodne, co? Dlatego też my skorzystamy z tegoż to menadżera pakietów, by zainstalować to, co nam potrzebne.

Instalujemy Choco Dokładne instrukcje znajdują się na stronie chocolatey (https://chocolatey.org/install) i tam też musisz się udać, by dowiedzieć się jak zainstalować choco.

6.2.1 Instalacja potrzebnych nam rzeczy za pomocą choco

Chocolatey zainstalowane? Wyśmienicie, teraz już tylko jedno polecenie nas od finiszu.

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

I gotowe. Co, kiedy będziemy sobie chcieli zaktualizować wszystkie/wybrane programy?

Ponownie, nic prostszego:

```
choco upgrade all
```

lub

```
choco upgrade nazwa_programu
```

Tłumaczyć chyba nie muszę. Listę dostępnych do zainstalowania w ten sposób paczek, można znaleźć tutaj. Większość popularnych programów tam jest.

Jak nam się nie chce za każdym pytaniem o upgrade klikać 'y', to wystarczy dodać `-y` na koniec polecenia, analogicznie do przykładu z instalacją sublimetexta.

Teraz jedynie otwieramy SublimeText, tworzymy gdziekolwiek dowolny katalog i w SublimeText klikamy File → Open Folder, wybierając utworzony wcześniej folder, który posłuży nam za główny folder dla plików z tej książki.

6.3 Instalacja na Linux/macOS

Co do instrukcji dla Linuxa lub macOSa i jego użytkowników. Cóż, dla was nie ma.

Ale jak to? Co to ma być za dyskryminacja? Otóż prawdopodobnie macie już zainstalowanego Pythona, sprawdźcie tylko w jakiej wersji. Jak?

```
python --version
```

(dwa myślniki) w konsoli i gotowe, lub, zależnie od dystrybucji, może być to

```
python3 --version
```

Zależy, z jakiej dystrybucji korzystasz i jakiej wersji Pythona używa jako domyślnej.

Najlepiej, żeby był to Python w wersji >= 3.7.0. Jeśli nie jest, to zupgrejdujcie sobie.

Jak to zrobić? Bądźmy szczerzy, jeśli korzystasz z Linuxa, to raczej nie musisz się mnie o to pytać. Podobnie z instalacją SublimeText. A jeśli naprawdę nie masz pojęcia to... Przypomnij sobie, co mówiłem o najważniejszej umiejętności informatyka – przetwarzanie informacji. Szukaj zatem.

A jak z maca, to użyj narzędzia `brew`. I tyle.

No i jak wspominałem, nie musi być to SublimeText – możesz używać swojego ulubionego edytora. Bez znaczenia. Tylko proszę, nie używaj windowsowego notatnika do tego. Za każdym razem, gdy to robisz, gdzieś na świecie umiera kotek. Nie rób tego.

6.4 Przechodzimy do programowania! Nareszcie!

Znaczy zaraz. Teraz musimy jeszcze, za pomocą konsoli, a jakże, przejść do głównego katalogu kodu dla tej książki, który niedawno utworzyliśmy.

By to zrobić, musisz znowu otworzyć konsolę, tym razem niepotrzebny ci już będzie tryb administratora, zatem otwórz ją jako zwykły użytkownik, czy to na linuxie, czy na windowsie.

I co teraz? Pora zapoznać się z podstawowymi komendami, które mogą ci się przydać. Głównie chodzi o cztery podstawowe i to je omówię, jest ich oczywiście znacznie więcej i część z nich, które mogą być ci potrzebne w codziennej pracy, omówimy później.

Nie chcę po prostu zarzucać cię na samym początku informacjami, które nie będą ci na razie potrzebne, a jedynie dorzucą rzeczy do listy zagadnień, które musisz pojąć.

6.5 Czterej jeźdźcy konsoli

1. `dir` (Windows) lub `ls` (Linux). To polecenie, które wypisuje zawartość katalogu, w którym obecnie się znajdujemy w konsoli. Skąd wiedzieć, gdzie akurat jesteśmy? Sprawa prosta – nasz katalog roboczy (CWD – current working directory) jest wyświetlany po lewej stronie od naszego kursora terminalu lub wpisując pwd na Linuxie/macOS. Windowsiarze googlują. Zachciało im się, systemów od MS. To teraz macie. Co nam daje to, że jesteśmy w jakimś katalogu? Otóż nasze polecenia będą wykonywane względem tejże ścieżki. Czyli jak wpiszemy sobie `python plik.py`, to jeśli będziemy w folderze, załóżmy, `C:\Users\Olaf` to Python będzie szukał sobie tego pliku, plik.py właśnie w tym folderze.

2. `mkdir` – wspólna dla obu systemów, tworzy katalog o podanej nazwie np. mkdir folder utworzy w obecnym katalogu roboczym folder o nazwie folder. Makes sense.
3. `del` (Windows) i `rm` (Linux) te komendy służą do usuwania plików. Jeśli chcemy usunąć folder, możemy do tego użyć rd na Windows czy rm -rf na Linuxie. Przykład użycia: rm -rf plik.txt.
4. `cd` – wspólna dla obu systemów. Cd, czyli change directory. Zmiana bieżącego katalogu roboczego na zadany. Przykład użycia - cd .. - to polecenie przeniesie nas do katalogu nadrzędnego w strukturze plików. Czyli jeśli jesteśmy w `C:\Users\Olaf\Test\` to użycie cd .. przeniesie nas do `C:\Users\Olaf\`.

Przydatna informacja: kliknięcie TAB powoduje, że konsola sama nam dopisuje nazwy katalogów, czy plików;

Na Linuxie (zazwyczaj) znakiem ~ oznaczamy katalog domowy użytkownika, więc jeśli chcesz do niego przejść, nie musisz podawać pełnej ścieżki. Wystarczy wpisać cd ~ i już.

Jeszcze jedna uwaga, nie wszystkie linuxy mają zaimportowany obecny katalog roboczy do PATH, zatem czasami może wystąpić potrzeba, że trzeba będzie wpisać, jako lokację pliku w obecnym CWD, ./plik.txt zamiast plik.txt, zależnie od distro. Łatwo można to naprawić, dodając CWD do PATH. To samo może być z samym Pythonem. Pamiętaj, by podczas instalacji dodać go do PATH. Ewentualnie jak nie podczas instalacji, to ręcznie sobie dodaj.

Jak? Googluj, linuxiarzu. Linuxów się zachciało instalować, to teraz masz. A męcz się, męcz, jak chciałeś, czy chciałaś!

Nawigować już umiemy, ale do konsoli jeszcze wrócimy. Jak odpalić nasz kod źródłowy?

Prosta sprawa.

```
python nazwa_pliku.py
```

Uwaga, jeśli jesteś na Linuxie i miałeś wcześniej zainstalowaną wersje Pythona 2, to być może będziesz musiał używać komendy python3 do uruchamiania, ale pewnie o tym wiesz.

Zaś by uruchomić interpreter pythona wystarczy wpisać:

```
python
```

I tyle. Odpali nam to interpreter. Różnica między interpreterem a odpalaniem z pliku jest taka, że w interpreterze wpisujemy sobie polecenia a python na bieżąco je interpretuje. W przypadku pliku mamy mniej interaktywne podejście. Pobaw się i przekonaj

Środowisko ogarnięte, poruszanie się w konsoli też. Bierzemy się wreszcie do pracy!

7 Witaj, Świecie!

Wreszcie zaczynamy zabawę z kodowaniem! Uf, trochę nam to zajęło, co? Tylko jakieś 50 stron. W każdym razie.

7.1 Wypisujemy tekst na ekran

Witaj, Świecie! Te słowa, to dość popularny zwrot w programowaniu, przynajmniej ich angielska równowartość – Hello World!

Jest to coś, co tradycyjnie uznaje się za tekst, jaki wydrukuje pierwszy program ,który początkujący programista stworzy w swej karierze – wydrukowanie napisu 'Hello World!' czy 'Witaj, Świecie!' w konsoli.

My zrobimy podobnie, ale nieco inaczej, bo na dwa sposoby. Dlaczego? Otóż w Internecie często można spotkać się z opinią, że w Pythonie na wszystko istnieje już rozwiązanie, gotowy kod, który ktoś inny stworzył, my jedynie go importujemy, wykorzystujemy i w sumie na tym polega całe programowanie w Pythonie.

Takie sklejanie programu z gotowych klocuszków. Grunt to wiedzieć, z jakich klocuszków skleić.

Cóż, często to racja. Bardzo często. Nie inaczej jest z hello worldem. Przykłady pokazane tutaj powinno się przepisywać do pliku i uruchamiać jak opisano w poprzednim rozdziale lub na bieżąco wklepywać w interpreter. Polecam tę drugą opcję, przynajmniej póki mamy jednolinijkowce/kilka linijek. Przy dłuższych rzeczach polecam jednak pracować na plikach nie na samym interpreterze.

Otóż w Pythonie, standardowy Hello World, można zastąpić:

```
import __hello__
```

Co ukaże się naszym oczom?

```
Hello world!
```

Yup. Python nawet na hello worlda ma gotowe rozwiązanie, ale to tak bardziej w ramach ciekawostki – teraz pokażę wam, jak można wydrukować coś na ekranie za pomocą Pythona w sposób normalny.

```
print("Witaj, Świecie!")
```

I tyle w zasadzie. W konsoli wyświetlić się: „Witaj, Świecie!". Tylko tyle i aż tyle.

Mała notka też dla tych bardzo początkujących: jeśli nie wiesz, gdzie ten kod wpisać, to sprawa jest prosta – utwórz sobie dowolny plik, który będzie miał rozszerzenie .py, w swoim folderze roboczym, czyli CWD, zajrzyj do rozdziału o czterech jeźdźcach konsoli, jeśli już zapomniałeś, o co chodzi.

Następnie w tym pliku wpisz właśnie tę linijkę za pomocą edytora tekstu. Potem wystarczy tylko:

```
python nazwa_utworzonego_pliku.py
```

i gotowe. Ewentualnie zamiast python, być może będzie trzeba wpisać python3, zależy jak tam sobie poinstalowałeś wszystko.

Co tutaj się stało? Skorzystaliśmy z jednej ze wbudowanych funkcji Pythona, które umieszczone są w standardowej bibliotece języka, czyli zbioru funkcji, którą każda instalacja Pythona3 posiada. Funkcja ta nazywa się print – z angielskiego, wydrukuj.

Hmmm, co zatem może robić funkcja, która nazywa się „wydrukuj"? Dobre pytanie. Wydaje mi się, że tutaj nastąpi prawdziwy test tego, czy nadajesz się na programistę. Jeśli jesteś w stanie określić, co robi funkcja print/wydrukuj, to prawdopodobnie nadajesz się na programistę. Gratulacje.

Tejże funkcji przekazujemy argument (czyli coś, na czym funkcja ma zadziałać) w postaci tego, co ma wydrukować. Zostanie to wypisane na ekranie, razem ze znakiem nowej linii. Znak nowej linii, który automatycznie Python doklei nam do tego, co drukujemy, oznacza, że jeśli teraz użyjemy następnego printa, by znowu coś wypisać, to nasza wartość, z drugiego printa, zostanie wypisana w nowej linijce.

Bo chodzi o to, żeby komputer wiedział, kiedy 'entera' nowego wyświetlić i zakończyć obecną linię. To żeby mu dać znać, kiedy ma to zrobić, mamy coś takiego jak 'znak nowej linii'. Jak sobie enter wciskasz w Wordzie, to pod spodem tam jest wstawiany ten właśnie znak. Czyli:

```
print("Linijka 1")
print("Linijka 2")
```

Zwróci nam, zgodnie z oczekiwaniami:

```
Linijka 1
Linijka 2
```

Znakiem końca linii jest zazwyczaj `\n`. Czyli w rzeczywistości, zamiast wyświetlić tylko `„Linijka 1"`, Python wyświetli `„Linijka 1\n"`.

Czy to wszystko proste? Anty klimatyczne? Tak. Przynajmniej pozornie. Bo pod spodem, dzieje się tu wiele bardzo, bardzo ciekawych rzeczy, o których na razie nie masz pojęcia.

To, że dziś, za pomocą jednej linijki kodu, jesteś w stanie wypisać sobie w konsoli jakiś tekst, to efekt kilkudziesięciu lat pracy i budowania fundamentów przez ojców informatyki. Wiem, że może brzmieć to śmiesznie, ale tak jest. Popatrz np. na kod Assemblera, czyli języka, w którym wszyscy kiedyś pisali.

Pokazany niżej fragment kodu to Assembler, bardzo niskopoziomowy język, który w bardzo dużym szczególe pozwala na interakcję bezpośrednią z pamięcią komputera, z procesorem, ze wszystkim w sumie. Dzięki temu programista może zarządzać i optymalizować niemal wszystko, jednakże ma to swoją cenę w postaci tego, iż skoro trzeba zarządzać wszystkim samemu, to... No właśnie. Trzeba to robić samemu. Ma to wady i zalety. Nie przejmujmy się tym jednak zbytnio na razie, wspominam tylko po to, by gdzieś ci to tam w głowie krążyło. Assembler = szybki, niskopoziomowy język, gdzie dużo rzeczy trzeba robić samemu, który jest bardzo blisko samego procesora/pamięci, o niskim poziomie abstrakcji.

```
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

; wyjście z programu
        mov     eax, 1
        xor     ebx, ebx
        int     0x80
```

Wait... What? Także tak. Doceń to, co masz teraz.

Tylko od razu zaznaczam – nie przejmuj się jeśli kompletnie nic nie rozumiesz z tego kodu. Spokojnie. Ja też nie za dużo. To nie ma znaczenia. Chodzi tylko i wyłącznie o to, by pokazać ci ciekawe, stare drogi.

A wiesz, co jest jeszcze ciekawsze?

Fakt, że obecnie, niżej, pod spodem, Python właśnie tak wygląda. To znaczy nie sam Python – bo Python to tylko język - zbiór zasad, definicji, natomiast chodzi o CPythona – czyli implementacji interpretera Pythona, a one już mogą być dowolne, pisałem o tym wcześniej, możesz poczytać. Warto to zapamiętać, że domyślną implementacją Pythona jest CPython.

Różnica między Pythonem a CPythonem jest taka, że Python to po prostu język, czyli zbiór instrukcji i opis tego, jakie ten języka ma funkcje i jak ma się zachowywać w danych sytuacjach.

A CPython to już konkretne implementacja tego – przetłumaczenie na zachowania komputera, konkretny program wykonujący polecenia w konkretny sposób. Także CPython != Python.

Czyli jeszcze raz powtórzmy. Python to język. Interpreter Pythona to już jakiś program, który interpretuje kod napisany w języku Python i wykonuje określone polecenia.

Zazwyczaj, kiedy mówimy o interpreterze Pythona, mamy na myśli jego domyślną implementację, czyli CPythona – interpreter Pythona napisany w C, ale są też inne. Pamiętaj. Pamiętaj też o tym, że szczegóły implementacji interpreterów Pythona różnią się pomiędzy sobą. Autorzy języka pozwolili na to by, pewne decyzje co do określonych zachowań, podejmowały osoby implementujące interpreter. Zatem CPython może, ale nie musi, zachowywać się czasami inaczej niż Jython. Zatem dobrą praktyką jest, by nie polegać na szczegółach implementacyjnych interpretera a samej specyfikacji języka. Wrócmy jednak do tematu.

Dlaczego trzeba doceniać obecne abstrakcje?

Zacznijmy od tego, że jeśli korzystasz z jakiegokolwiek w miarę współczesnego komputera, to jestem prawie pewien, że widzisz polskie znaki w wypisanym tekście, bez żadnych problemów. W ogóle widzisz ten tekst. Prawda?

Nie zawsze było to takie proste i oczywiste.

Dlaczego?

Wynika to z tego, jak działa komputer.

7.2 Język binarny – jedyne, co rozumie komputer

Nie wiem jak dla ciebie, ale dla mnie zawsze interesujące było to, jak działa komputer. Jak to się dzieje, że po wciśnięciu jakiegoś magicznego guziczka, energia elektryczna zaczyna 'przepływać' przez tę cudowną maszynę, na ekranie pojawiają się różne znaczki i wszystko jest takie piękne, fajne.

Otóż sprawa jest prosta. U samiutkich podstaw tego, jak działają komputery, leży nic innego, jak dwie proste rzeczy: Prawda i Fałsz, 0 i 1, tak i nie. Chodzi mi o system binarny, język maszynowy.

Co przez to rozumiem? Otóż komputer, to nic innego, jak taka ogromna, ogromna grupa zlepionych razem przewodników/półprzewodników, takich jakby 'przełączników', które mogą być 'włączone' lub 'wyłączone' - mają dwa stany. Stany te reguluje się za pomocą napięcia prądu, jaki przez przewodniki przepływa, to ono świadczy o tym, czy któryś jest włączony, czy wyłączony.

W zależności od tego, jaką mamy kombinację, które 'przewodniki' mamy włączone, które wyłączone, komputer będzie robił różne rzeczy. To tak jak z pralką – zależnie od tego, jakie przyciski na niej pozostawisz wciśnięte, zrobi ona coś innego.

Z racji tego, że mamy tu dwa stany, cokolwiek co ma z tym do czynienia, często okraszamy przymiotnikiem 'binarny'. System liczb binarnych. Wybór binarny. Drzewo binarne. Ludzie binarni żyjący w świecie probabilistycznym i tak dalej.

W każdym razie. Ustalamy zatem jedno. Nasz komputer operuje tylko na dwóch wartościach – 0 i 1, czyli brak/niskie napięcie i wysokie napięcie. To wszystko. W dużym uproszczeniu to właśnie to jest kompletną podstawą całego komputera i nic więcej.

Zakładam, że moi czytelnicy to sprytni ludzie. Powinno zatem pojawić się zaraz pytanie – Ale jak to? Skoro komputer rozumie tylko 0 i 1, to jak to się dzieje, że mogę tu wpisywać różne litery, czytać je potem, mogę poruszać myszką, wpisywać inne liczby niż 0 i 1. Co? Pan coś tu ściemnia, panie Górski.

Otóż nic bardziej mylnego. Wasz komputer naprawdę rozumie tylko 0 i 1. Wszystko inne to efekt różnego rodzaju kalkulacji, przeliczeń i kodowania innych wartości do właśnie

tychże 0 i 1.

Doskonałym przykładem jest tutaj ten tekst.

7.3 Jak komputer widzi litery – system binarny

Wyobraź sobie, że wszystkie literki, które tutaj widzisz, tak naprawdę, pod spodem są niczym innym jak liczbą, zapisaną w systemie binarnym.

Dla ścisłości – czym jest liczba zapisana w systemie binarnym? To nic innego jak normalna liczba, tylko wyrażona za pomocą jedynie 0 i 1. My, jako ludzie, obraliśmy sobie jako podstawowy, system dziesiętny. Prawdopodobnie dlatego, że tyle mamy palców, ale kto tam dokładnie wie.

W każdym razie – operujemy na założeniu takim, iż mamy 10 cyfr, każda rząd wielkości może mieć 9 lub 10 możliwych stanów, lub może go nie być wcale, a rzędy wielkości opierają się na potęgach dziesiątki.

W binarnym systemie jest tak naprawdę podobnie, tyle że zamiast 10 cyfr, mamy tylko dwie i dwie wartości. Dodatkowo kolejne rzędy wielkości przeliczamy sobie nie na podstawie potęg dziesiątki a na podstawie potęg dwójki.

Weźmy, dla przykładu, liczbę 123. Jak liczymy jej wartość? Otóż.

1 – liczba setek, trzecia cyfra 2 – liczba dziesiątek, druga cyfra 3 – liczba jedności, pierwsza cyfra

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

Jak łatwo zauważyć, wykładnikiem potęgi jest liczba, równa numerowi cyfry, licząc od prawej strony, pomniejszona o 1.

Brzmi to wszystko skomplikowanie, bo my tak o tym nie myślimy – robimy pewne rzeczy naturalne z racji doświadczenia, więc trochę praktyki może wymagać przestawienie się na taki sposób myślenia o tym.

Sytuacja w binarnym zapisie będzie analogiczna. Jak obliczamy wartość danej liczby w zapisie binarnym? Załóżmy, że chcemy dowiedzieć się, jaką wartość ma liczba 101101.

```
1 - 6. cyfra liczby
0 - 5. cyfra liczby
1 - 4. cyfra liczby
1 - 3. cyfra liczby
0 - 2. cyfra liczby
1 - 1. cyfra liczby
```

A zatem:

$1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$

$1 * 32 + 0 * 16 + 1 * 8 + 1 * 4 + 0 * 2 + 1 * 1$

$32 + 0 + 8 + 4 + 0 + 1$=45

Czyli 101101 to nic innego jak odpowiednik 45 w systemie dziesiętnym. A jak przeliczyć z systemu dziesiętnego na binarny? Bardzo prosto. Otóż dzielisz sobie liczbę przez dwa i za każdym razem wypisujesz resztę.

$45/2 = 22$ reszty 1

$22/2 = 11$ reszty 0

$11/2 = 5$ reszty 1

5/2 = 2 reszty 1
2/2 = 1 reszty 0
1/2 = 0 reszty 1

Teraz, czytamy sobie reszty od DOŁU do góry: 101101. Zgadza się? Yep.

Przeanalizuj to sobie na razie dokładnie i powoli. To nic, jeśli na początku coś nie jest jasne, to tylko sposób przeliczania z jednego systemu na drugi.

Przydaje się tutaj znajomość wartości potęg dwójki. Gdy w pamięci masz jakąś ich część, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384 itd., to niektóre przeliczenia staną się łatwiejsze i szybsze.

W każdym razie. Wróćmy do tematu. Czyli... A tak. Literki to też liczby.

7.4 Kodowanie znaków – ASCII

No właśnie. Skoro komputer rozumie tylko i wyłącznie liczby, te w systemie binarnym, czyli w zasadzie zera i jedynki, to jak mu powiedzieć, że ma przechowywać jakąś literkę?

Wyobrażacie sobie świat, w którym zamiast czytać moje słowa na ekranie za pomocą liter, czytalibyście każdą literkę zapisaną w jakiś sposób za pomocą systemu binarnego, ręcznie na kartce sobie to tłumaczyli na ludzki? Ja nie, ale mniej więcej tak to wygląda w rzeczywistości, tylko robi to za nas komputer.

Otóż kilku mądrych panów zebrało się kiedyś i stwierdzili oni, że w sumie dobrym pomysłem będzie, by niejako stworzyć tłumaczenie swego rodzaju – tłumaczenie, mapowanie liter alfabetu do... liczb.

Każda litera alfabetu otrzymała swój unikalny kod w postaci jakiejś liczby. Dlaczego? Bo jak powiedzieć komputerowi, że 'k', to 'k'? Nie da się. Komputer jest głupi, nie rozumie pojęcia litery. Przynajmniej póki co, za kilka albo kilkadziesiąt lat, kto wie.

Liczby w systemie dziesiętnym zaś bez problemu przekonwertujemy do liczby w systemie binarnym, czyli do czegoś, co komputer już zrozumie.

Ustalono sobie zatem coś, co nazywa się standard ASCII, czyli skrót od ang. American Standard Code for Information Interchange. To taka swego rodzaju tablica, która zawiera mapowanie znaków alfabetu angielskiego, znaków interpunkcyjnych na liczby. Taki słownik niejako, która literka, jakiej liczbie odpowiada.

Dla przykładu 'A' - duża A, zostało oznaczone jako 0100 0001, czyli 65. 'a' to zaś 97. Znak nowej linii to 0000 1010, czyli 10. Dlaczego? Bo tak i już. Tak sobie amerykańskie mędrki wymyśliły i koniec. Standard został ustalony, stosujcie się do niego. Taka kwestia umowna to w sumie jest.

Mała nota, zazwyczaj liczby zapisane w systemie binarnym piszemy z przedrostkiem `0b`, żeby było wiadomo, że mamy do czynienia akurat z binarnym. Bo jak tu odróżnić 10 w dziesiętnym od 10 w binarnym? Zapis ten sam a wartości różne. To jak w tym żarcie, że na świecie jest tylko 10 rodzajów ludzi, ci, którzy rozumieją język binarny i cała reszta. Hehehe, programistyczny sucharek, sory.

B w kodowaniu ASCII to z kolei 0b0100 0010, czyli 66. C będzie miało liczbę 67. I tak dalej. Zgadnij jaką będzie miało D?

Każda literka, którą tutaj widzisz, jest tłumaczona w podobny sposób i zapisywana na dysku twojego komputera jako ciąg jedynek i zer. Następnie, przy odczytywaniu, komputer, po zinterpretowaniu jaką literą jest dana liczba, wyświetla określoną literkę.

Kompletnie on jednak nie rozumie tego, że ta 'k' to jest jakaś literka, a nie kawałek kodu binarnego. Po prostu.

Co wyróżnia ASCII? ASCII jest kodowaniem, gdzie każdy element można wyrazić za pomocą 7 bitów. Co to znaczy, za pomocą 7 bitów? Bit to nic innego jak 'cyfra' w systemie binarnym.

Przynajmniej pierwotnie było takie założenie. Pozwala ono zatem na translację 128 znaków, na liczby z przedziału od 0 do 127, włącznie.

Wiele osób, przynajmniej te, które miały trochę styczności z programowaniem lub informatyką, się zdziwi. Jak to – ASCII 7-bitowe? Wszystkich uczą często, że 8-bitowe przecież. Otóż nie, pierwotnie ASCII było projektowane jako 7-bitowy system, posiadając 128 znaków.

To, że dziś często myślimy o ASCII w kontekście 8-bitowym, wynika z tego, że podczas tworzenia ASCII, 8-bitów nie było jeszcze takim standardem. Wiele systemów kodowań było 7-bitowych, a 8 bit wykorzystywały sobie na jakieś tam swój różne dziwne potrzeby.

Co to znaczy, że 8 bitów jest 'standardem'? Otóż obecnie mamy coś takiego jak bajt. Bajt to z kolei zbiór 8 bitów. Czyli np. 1111 0000 czy 1000 0000.

To taka bardzo mała jednostka pamięci twojego komputera, z której możesz skorzystać. Czy to w RAM, czy w pamięci dyskowej.

Jest to jasno określone i proste. Ale... Bajt nie zawsze był określony jako 8 bitów. Istniały systemy, gdzie 1 bajt, podstawowa jednostka, były zdefiniowane zupełnie inaczej – na 2 bity, na 7 bitów, na 6 bitów. Pick your poison. Wolna amerykanka.

W ogóle dużo rzeczy w informatyce, czy programowaniu, podobnie jak w matematyce, jest umownych. Przyzwyczaj się do tego, że czasem robimy tak coś w konkretny sposób, bo tak, a nie inaczej. Informatycy to ogółem od matematyków się wywodzą w prostej linii w zasadzie, zatem dziwni z nas ludzie.

Dlatego też obecnie, mimo tego, że ASCII oryginalnie jest 128 znakowym systemem 7-bitowym, to zapisuje się go za pomocą 8 bitów, dając mu 256 możliwych znaków, oficjalnie, 8-bitowe ASCII nazywamy rozszerzonym ASCII, ale potocznie, zazwyczaj jak mówimy ASCII, to mamy na myśli to 256-znakowe – z kilkoma dodatkowymi znakami.

Wybrany fragment tabelki znaków ASCII:

Decymalny	Znak	Binarny	Decymalny	Znak	Binarny
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

Decymalny	Znak	Binarny	Decymalny	Znak	Binarny

Po prostu wcześniej był chaos. A z chaosu wyłonił się ład. I powstała informatyka. W miejscu tym ład i chaos współistniały w harmonii byś mógł pokazać mi swoje towary. Dobrze, starczy nerdowskich nawiązań do Gothica.

No, w każdym razie. Bo wątek zgubiłem.

Mamy sobie to kodowanie, w tym wypadku ASCII, i jest elegancko. Chociaż nie do końca, bo pojawia się problem. ASCII teraz używamy na 8 bitach, tak? No tak. 8 bitów, to 8 zer lub jedynek, tak? Tak. Za pomocą 8 zer lub jedynek jesteśmy w stanie wyrazić 256 liczb.

Biorąc pod uwagę, ze 1 liczba = 1 litera, to szybko wyjdzie nam, że mamy do wykorzystania maksymalnie 256 znaków. Trochę mało, jak na to, żeby pomieścić wszystkie alfabety świata, prawda? Owszem. Weźmy tu jeszcze pod uwagę, że 'A' i 'a' to dwie różne litery jak dla komputera – mały i duży znak to nie to samo. Doliczmy kropki, przecinki i inne znaki interpunkcyjne.

Szybko okazuje się, że na same literki za dużo miejsca nie zostało.

Dlatego też kodowanie ASCII zawiera tylko litery alfabetu łacińskiego. Po polsku już sobie nie napiszesz w ASCII, bo znaków brakuje. Kurła i co teraz?

A no widzicie, żeby Polacy i inne nacje, typu nie wiem, Chińczycy czy Japończycy, którzy mają tych znaczków dość sporo, bo każdy wyraz potrafi być innym znaczkiem, czyli tak jakby u nas inną literą, nie czuli się pokrzywdzeni, zaczęły powstawać nowe kodowania. Dużo kodowań. O wiele za dużo. Nie będziemy o nich mówić. Od razu przeskoczymy dalej, do czasów jasności.

7.5 I wtedy UNICODE I UTF-8 wchodzą całe na biało

Obecnie jednak takim standardem jest coś zwanego UTF-8. Jest to system kodowania znaków UNICODE, który do zapisu wykorzystuje od 1 do 4 bajtów. O, ważna informacja. Do 4 bajtów. 4 bajty, ile to było? 1 bajt, 8 bitów, 4 bajty, 32 bity. A co to znaczy, że UTF-8 to system kodowania znaków UNICODE? UNICODE to to nasze całe mapowanie, które literka/znaczek do jakiej liczby się sprowadza, a UTF-8 to sposób zapisu tego, konwersji na bity, bo jak masz nieco więcej niż 1 bajt, to sprawa robi się mniej oczywista, stąd można te znaki/mapowania, różnie przedstawiać. UTF-8 jest jednym ze sposobów na to. Wracając do tematu bajtów...

32 bity, to z kolei 32 jedynki lub zera, czyli za ich pomocą można zapisać wiele różnych liczb, zatem też i różnych znaków. No tutaj to już sporo się robi, bo to nam daje jakieś 4 294 967 296 możliwych znaków. Sporo, prawda? Wciąż jest to mniej niż 70 milionów jakie Sasin zmarnował na wybory, które się nie odbyły, ale to nic. Nawet jak te wszystkie azjatyckie znaczki wrzucimy, to i tak sporo miejsca zostanie. Pięknie, idylla. Marzenie.

Marzenie, bo w rzeczywistości, od wprowadzenia RFC 3629, UTF-8 obsługuje co najwyżej 2 097 152 znaków. To przez różne zaszłości historyczne, szczegóły implementacyjne i inne dziwne rzeczy, którymi nie musisz się martwić ani Ty, drogi czytelniku, ani ja, tylko raczej grube mózgi typu Ken Thompson i spółka, jakoś tak jednak wyszło, że niektóre bity są zarezerwowane na specjalne cele, niektóre bajty muszą mieć określony format, by wiadomo było różne przydatne rzeczy i tak dalej.

A o co chodzi z tym RFC całym? Ogółem to takie standardy, które pewne mózgi wyznaczają. Na jakiej podstawie? Na jakiej uznają. Podobnie jak z ASCII – bo tak i tak. Ogółem upraszczam i zasadniczo to podczas podejmowania różnych decyzji, osoby decyzyjne kierują się bardziej racjonalnymi argumentami.

Tyle znaków nam raczej wystarczy na co dzień. Obecnie w UNICODE mamy zdefiniowanych tak standardowo około 143 859 znaków. Czyli mamy nawet jeszcze trochę zapasu w razie czego, żeby dodawać potem nowe znaki.

Czy też raczej `codepointy`, ale my sobie uprośćmy, nie wnikajmy i mówmy po prostu znaków. Czym jest codepoint? Zazwyczaj jak się dyskutuje o różnych kodowaniach, to zamiast 'znak', używa się pojęcia codepoint. Drobna różnica. Z twojej perspektywy nie ma jakoś to bardzo znaczenia.

Czyli tak: skojarzcie, że jak UTF-8, to ogółem też i Unicode. Te dwa terminy trzymajcie w pamięci razem, ale UNICODE nie jest tym samym co UTF-8 i to bardzo ważne.

Do tego kolejna notka. Jest też coś takiego jak UTF-16. Czym się różni od UTF-8? Długością słowa. Czyli w UTF-8 jedno słowo ma 8 bitów, w UTF-16 ma 16 bitów. I tyle. Jest i UTF-32. Tutaj już jest zawsze jedno słowo, takie 32 bitowe.

Żeby nie było – bajtów mają w górnym limicie po tyle samo, czyli maksymalnie 4, a w przypadku UTF-16, minimalnie 2 (no bo 16 bitów). Czym jest słowo? Bynajmniej nie jest to słowo jak ze słownika. Słowo, czyli słowo maszynowe, to taki trochę bajt, ale nie do końca. Sposób na pogrupowanie bitów w X sztuk po prostu.

W UTF-8 niektóre znaki mogą zostać wyrażone 1 słowem, czyli 1 bajtem, w UTF-16 najmniejszy rozmiar to 2 bajty a w UTF-32 zaś to już 4 bajty. Czyli niezależnie od tego, jakiego znaku użyjemy, np. `A`, czyli 65, czyli coś, co mieści się w 1 bajcie, przy UTF-32, komputer i tak zapisze wszystko w 4, marnując sporo pamięci. W UTF-16 zapisze w 2, w UTF-8 w jednym. Oszczędniej. Za to UTF-32 łatwiej w pamięci znaleźć, bo wiesz, że każdy znak to 4 bajty i koniec. A w UTF-8 są różne długości, jeden znak będzie miał 1 bajt, inny 4, no i weź tu zgaduj człowieku, co jest czym, co się kiedy i gdzie kończy, ale jakoś dajemy z tym radę.

Nie musisz się tym zbytnio przejmować, ale gdzieś tam może zapamiętaj.

UTF-8 jest w 100% kompatybilny z ASCII – tekst w ASCII jest poprawnym UTF-8, ale UTF-8 już NIE musi być poprawnym ASCII. To bardzo ważne! Dlaczego? Gdyż UTF-8 zawiera głównie znaki, których nie zmieścimy w jednym bajcie przez co są one nieobecne w ASCII. Zatem ASCII to podzbiór UNICODE.

Swoją drogą, ciekawostka - jak piszesz jakieś wiadomości na fejsie i emotki wysyłasz, to one też są często zapisane w UNICODE, mają swoje konkretne liczby!

Czyli popatrzcie, samo zaczęcie jakiejkolwiek rozmowy o tym, jak działa podstawowa funkcja w Pythonie, rzucenie chociaż trochę światła na to, co leży pod jej przykrywką, zajęło mi tutaj kawał tekstu.

A to dopiero początek – ledwo co stópki zamoczyliśmy w całym temacie, jakby mi przyszło opisać wszystko o tej jednej prostej funkcji, to pewnie by mi tu książki nie starczyło.

Kiedyś tę wiedzę – o wszystkich niskopoziomowych rzeczach, faktycznie trzeba było posiadać, by cokolwiek napisać. Dziś?

Dziś mamy takie czasy, że bazując na dekadach pracy tytanów intelektu informatyki, możemy sobie stworzyć tak abstrakcyjne języki, że nic z tego nie musimy znać. Wystarczy

wpisać print(“xd”) i działa. Kiedyś to były czasy, tera to nie ma czasów. Programiści też kiedyś byli, teraz to nie ma.

To naprawdę coś niesamowitego, mimo tego, że nam wydaje się banalne. To jest właśnie piękno nauki, informatyki. Bazując na pracy innych, możemy tworzyć nowe, niepojęte rzeczy. Wyśmienicie.

W każdym razie.

7.6 Podsumowanie

Zróbmy podsumowanie tego, co udało nam się zrobić i dowiedzieć.

Mamy ogółem w pythonie taką funkcję jak **print**, która, uwaga, drukuje nam tekst na ekranie. Ten tekst w rzeczywistości, to nie jest tekst dla komputera, tylko nic innego jak ciąg zer i jedynek, bo komputer nie rozumie nic innego, za sprawą tego, jak jest zbudowany – napięcie lub brak/niskie napięcie – to jedyne co on tak naprawdę rozumie.

W związku z tym powstało coś takiego jak system binarny. To taki system liczenia, troszkę inny od dziesiętnego, w którym liczby wyrażamy za pomocą dwóch cyfr i tylko tyle. Jest on nieco bardziej rozlazły w porównaniu do dziesiętnego – zapisanie tej samej liczby, co w dziesiętnym, zajmuje więcej miejsca, można by rzec, ale ogółem nie jest to jakieś skomplikowane pojęcie, da się ogarnąć.

No i teraz mając już coś, co komputer jest w stanie zrozumieć, czyli **system binarny**, bazujący na dwóch wartościach, możemy na tym coś budować.

My, jako sprytni ludzie, zbudowaliśmy sobie coś, co się nazywa **kodowaniem znaków**. Otóż wykminiliśmy sobie, że w określonych wypadkach, dana liczba w systemie binarnym, będzie znaczyła nie konkretną liczbę a np. znak właśnie.

I tak oto powstało jedne z pierwszych popularniejszych kodowań, czyli **ASCII**. ASCII było spoko, ale miało tę wadę, że mało znaków twórcy tam przewidzieli, powiedzmy. Wiadomo, wujek sam, pępek świata, nie pomyśleli o innych nacjach, ino o znakach ze swojego alfabetu.

Zatem przyszło coś nowego, czego używam do dziś, co jest nieco lepsze – **UTF-8**. UTF-8 to sposób kodowania znaków **UNICODE**. Unicode to swego rodzaju ‘mapowanie’ liczb na dane znaki.. Żeby nie było problemów z kompatybilnością wsteczną, to znaczy, żeby stare teksty i programy działały na nowych komputerach, UTF-8 jest **kompatybilny wstecznie** z ASCII, to znaczy tekst w ASCII jest też poprawnym tekstem UTF-8. W drugą stronę już niekoniecznie nie każdy UTF-8 jest poprawnym ASCII.

Pewnie cię trochę nudzę, co? Ostrzegałem na początku – będzie też trochę teorii i innych rzeczy, bo to nie będzie tylko książka o Pythonie. Chociaż przyznam szczerze, że mnie akurat te wszystkie tematy bardzo jarają, interesują.

To dla mnie niesamowite, co stworzyliśmy jako ludzkość i jak te wszystkie procesy zachodzą. Piękna sprawa. Mam nadzieję, że chociaż po części będzie mi się udawało zaciekawić cię, czytelniczko, czytelniku, podczas tej lektury, takimi tematami – nie tylko samym Pythonem, ale informatyką, nauką ogółem.

Z drugiej strony, uważam, że takie podejście, które tu prezentuje – omawiając szerszy zakres, trochę historii i teorii, a nie same suche powiedzenie, „O tu masz printa i to drukuje tekst.” jest o wiele lepsze. Daje ci ono wgląd w fundamentalne teorie, które leżą u stóp tego, czym się będziesz posługiwać na co dzień. Poznasz narzędzie i jego budowę, zastosowanie,

będziesz świadom. Moim zdaniem to konieczne do bycia dobrym programistą.

Zaraz przejdziemy do zadań/pytań. Oprócz nich, chciałbym, byś po każdym rozdziale bawił czy bawiła się sama nieco tym, o czym piszę – mówimy o print, poprintuj sobie trochę. Ja wiem, że wydaje się to nudne, ale zrób to. Proszę. Do tego, możesz poguglować trochę więcej i zgłębić tematy, o których tu mowa. To pomoże.

7.7 Zadania i pytania

Niektóre będą mega banalne, ale i tak odpowiedz. No. Jak nie do końca znasz odpowiedź, to się nie przejmuj, przeczytaj jakiś kawałek jeszcze raz ewentualnie, spróbuj pomyśleć.

Najlepiej to weź kawałek papieru i na nim spisuj swoje odpowiedzi na pytania, które nie wymagają programowania. To sprawi, że lepiej zapamiętasz. Sformułuj odpowiedź na podstawie tekstu. Później podam ci odpowiedzi.

1. Jaka funkcja w Pythonie służy do drukowania tekstu na ekran?
2. Czy ta funkcja drukuje coś poza tekstem, który został wpisany, czy nie? Podpowiedź: co się stanie, jak ponownie jej użyjesz, do wypisania czegoś nowego na ekran? Czy tekst będzie w tej samej linii?
3. Skąd komputer wie, kiedy zacząć drukować w nowej linii?
4. Czym jest system binarny?
5. Czym jest bit? A czym bajt?
6. Jakiej długości obecnie jest bajt? Czy zawsze bajty były tej długości?
7. Jakie wartości rozumie komputer tak kompletnie u podstaw? Dlaczego?
8. O co chodzi z ASCII? Co to jest?
9. Jak komputer sobie wewnętrznie reprezentuje tekst, który wpisujesz?
10. A ten cały UTF-8 - co to?
11. Ile plus minus (rząd wielkości) znaków można przedstawić za pomocą dwóch kodowań, o których mówiliśmy w tym rozdziale?
12. Przelicz następujące liczby z dziesiętnego na binarny: 5, 10, 32, 127, 256.
13. Teraz w drugą stronę, z binarnego na dziesiętny: 0000 1101, 1000 0000, 0010 0100.
14. Czy te dwa systemy znaków, które omawialiśmy, są ze sobą kompatybilne wstecznie? Czy w obie strony? To znaczy A z B i B z A? Czy może tylko w jedną?
15. Jaka jest różnica między UTF-8 a UNICODE? Co jest czym?
16. Różnice między UTF-8, UTF-16 i UTF-32. Który zużywa najmniej pamięci zazwyczaj? Który najwięcej? Dlaczego czasami warto wybrać ten mniej optymalny pamięciowo wariant?

Odpowiedzi znajdziesz na następnej stronie.

7.8 Odpowiedzi

1. Funkcja print.
2. Tak, drukuje ona dodatkowo znak nowej linii na końcu naszego tekstu, co sprawia, że jeśli coś nowego wyprintujemy, to będzie to w nowej linii.
3. Komputer wie, że trzeba teraz tekst od nowej linii wypisać, jeśli napotka specjalny znak, znany jako znak nowej linii. Nam go nie wyświetla, ale sam go interpretuje.

4. To system zapisu liczb za pomocą 2 cyfr – 1 i 0.
5. Bit to podstawowa i najmniejsza jednostka informacji, której używamy do zapisu wartości binarnych , czyli taka cyfra w systemie binarnym– to taka podstawowa cząsteczka, która przyjmuje jedną z dwóch wartości. Bajt zaś to nic innego jak 8 bitów.
6. Obecnie przyjmuje się raczej, że bajt to 8 bitów. Nie zawsze tak było, przed rozpowszechnieniem konwencji 8-bitowej, można było spotkać bajty zupełnie innej długości.
7. Komputer, u samych podstaw, rozumie tylko dwie wartości. 1 i 0. Nic innego. Wszystko powyżej to już ludzka abstrakcja. Wynika to z tego, jak jest zbudowany – system binarny bazuje na tym, że komputer operuje na dwóch wartościach: napięcie i brak/niskie napięcie.
8. ASCII to system kodowania znaków. Takie tłumaczenie swego rodzaju, gdzie odpowiednie liczby przypisujemy do konkretnych znaków. Coś w rodzaju szyfrów które tworzyło się w dzieciństwie. Po prostu umawiamy się, że X znaczy Y.
9. Tak samo, jak wszystko inne – za pomocą 1 i 0, czyli liczb. Te liczby potem tłumaczy na konkretne znaki.
10. UTF-8 to system kodowania UNICODE. Czyli trochę takie ASCII, ale nowsze. Pozwala przedstawić więcej znaków i tak dalej.
11. ASCII – 256, UTF-8 – 2 097 152
12. Tego mi się nie chce robić.
13. Tego też nie.
14. Tak, UTF-8 jest kompatybilne wstecznie z ASCII. ASCII nie jest kompatybilne z UTF-8, czyli każdy poprawny tekst zakodowany w ASCII będzie poprawny w UTF-8/Unicode, natomiast nie każdy tekst w UTF-8 będzie poprawnym ASCII.

8 Zmienne, wprowadzenie

Dobrze, trochę sobie poprintowaliśmy, jest okej. Nie jest to jednak coś bardzo ekscytującego. Czy Python potrafi coś innego w ogóle? W innym wypadku to taki słaby z niego język w sumie. Oczywiście, że umie dużo więcej.

8.1 Zapamiętywanie wartości

Następnym pojęciem, jakie chciałbym przedstawić, jest idea zmiennych.

O co tu chodzi? Już tłumaczę.

Mamy sobie ten nasz komputer. Ma on jakąś tam pamięć, czy to RAM, czy dyskową, prawda? Prawda. Jest ona dość pojemna, szybka, czasem trwała nawet. Fajnie by zatem było móc z niej korzystać w jakiś sposób podczas programowania. Nasze mózgi to myślenia to nadają się całkiem dobrze, ale do pamiętania różnych rzeczy już gorzej, zwłaszcza trwałego.

Takim podstawowym narzędziem, z którego korzystamy, żeby coś sobie zapisać lub wyciągnąć z pamięci komputera, są właśnie zmienne/stałe.

To sposób, by do pamięci komputera wrzucić jakąś wartość i przypisać jej swego rodzaju identyfikator, by później można było z niej normalnie korzystać. Jak to wygląda w Pythonie? Prosta sprawa.

```
nazwa_zmiennej = wartość
```

Gdzie wartość jest praktycznie dowolna.

Nazwa za to już nie jest – są pewne zasady, których musimy się trzymać podczas nazywania zmiennych jak chociażby to, że nie może się ona zaczynać od cyfry, musi od litery czy znaku _.

8.2 Nazwy zmiennych

W nazwach zmiennych możemy stosować również polskie znaki, ale nie róbmy tego. Bo nie.

Zmienne nazywamy po angielsku, korzystając przy tym ze snake case - to taka praktyka, gdzie poszczególne wyrazy w nazwie zmiennej dzielimy od siebie za pomocą znaku _. Trzymaj się tego, bo to ważne, bardzo ważne.

Teoretycznie rzecz ujmując, jeśli pracujesz w 100% polskim zespole, gdzie masz pewność, że w przyszłości NA PEWNO nikt, kto polskiego nie zna, nie będzie czytał tego kodu (czyli nigdy), to okej. Teoretycznie można by pisać kod po polsku, ale... Nie jest to ogółem dobra praktyka, proszę, nie rób tego o ile tylko możesz. Czasem mogą cię przymusić na przykład przy projektach z sektoru publicznego, realizowanych przez pewne duże korporacje, ale to nie do końca są projekty, w których chcesz się znaleźć. Zazwyczaj.

Czyli sprawa ma się tak: zmienne i wszystko w naszym kodzie nazywamy opisowo, tak by od razu było wiadomo, co dany kawałek kodu robi, co znajduje się w zmiennej. Nie przesadzajmy jednak w drugą stronę – nazwą zmiennej nie powinien być cały poemat. Do tego w nazwach raczej używamy tylko liter, cyfr(rzadziej), podkreślenia. Tutaj konserwatywnie i bez szału. Zwięzłe, trafne nazwy.

Dlaczego? Poprawne nazywanie zmiennych, funkcji, klas i wszystkiego w twoim kodzie, sprawia, że jest on czytelny, że jest on zrozumiały. Po prostu. Musisz to robić. Tak, od samego początku. Wyrobi to w tobie dobry nawyk, który jest krytycznie ważny.

Pozwól, że rzucę ci przykładem.

```
def redirect_logged_in_user(self, request, *args, **kwargs):
    if self.redirect_authenticated_user:
        redirect_to = resolve_url(settings.REDIRECT_URL)
        if redirect_to == request.path:
            raise ValueError(
                "Redirection loop detected. Check that your"
                "REDIRECT_URL doesn't point to a login page."
            )
        return HttpResponseRedirect(redirect_to)
    return super().dispatch(request, *args, **kwargs)
```

Nawet nie znając zbytnio języka, a jedynie angielski, wychodzi na to, że dość szybko idzie się domyślić, co ten kod robi. Żeby nie było – to jest autentyczny kawałeczek z kodu produkcyjnego. Trochę zmieniony, ale sens zachowany.

Konkretnie chodzi o to, że to jakaś funkcja/metoda czy coś tam, co bierze jakieś żądanie, czyli pewnie jakaś web aplikacja, sprawdza, czy powinno się przekierowywać zalogowanych użytkowników i wtedy, jeśli użytkownik jest zalogowany a przekierowywanie jest włączone, to go przekierowuje gdzieś, a gdzie go ma przekierować, to zależy od jakiejś zmiennej REDIRECT_URL z settingsów, czyli ustawień, czyli pewnie jakaś konfiguracja.

Jeśli nie znacie angielskiego, to nic, spróbujcie to przetłumaczyć na własny rachunek. Googlujcie nawet słowo po słowie, a okaże się, że naprawdę bardzo szybko można dojść do tego, co dany kod robi. Wystarczy trochę znajomości języka i niewielki kontekst informatyczny odnośnie tego, jakie zwyczaje mają programiści w nazywaniu pewnych rzeczy, konkretnych terminów i ich znaczeń.

No i przy okazji sprawdza się, czy to `gdzieś`, w które mamy przekierować użytkownika, nie jest przypadkiem miejscem, w którym się znajdujemy, bo wtedy nam się nieskończona pętla stworzy. Ciągłe przekierowania. Nieskończona pętla.

Teraz, dla kontr przykładu, kod z **nieco** mniej opisowymi nazwami.

```
    def rdr_lg_usr(self, r, *args, **kwargs):
        if self.rau and r.u.ath:
            to = rslv(stings.TO)
            if to == r.pt:
                raise VEr(
                    "Redirection loop detected. Check that your"
                    "REDIRECT_URL doesn't point to a login page."
                )
            return HttpRR(to)
        return super().dp(r, *args, **kwargs)
```

Nie wiem jak wam, ale mnie ten kod nic nie mówi w zasadzie. No dobra, na podstawie

pewnych informacji, mogę się domyślać niektórych rzeczy, wywnioskować je z kontekstu, ale...

To nie tak powinno wyglądać. Absolutnie. Za każdym razem, kiedy widzę jakikolwiek produkcyjnie wypuszczony kod, który wygląda jakoś podobnie, dostaje raka. Potem mój rak dostaje raka.

I tak sobie obaj siedzimy, ja i mój rak, i płaczemy, bo obaj mamy raka. I po co było czytać taki kod? Jeszcze gorzej – czasami trzeba z czymś takim pracować, bo jakiś jełop zdecydował, że jak sobie skróci `redirect` do `rr`, to te 6 literek co sobie zaoszczędził, zbawi jego świat, jego piękne palce, codebase i wszystko inne. A idźże mnie pan z tym.

Czasami, naprawdę, bardzo, ale to bardzo rzadko trafia się taka okoliczność, gdzie faktycznie można coś tam skrócić. Są to jednak zdecydowanie wyjątki od reguły. To takie miejsca, gdzie nawet jak rzucisz skrótem, to każdy będzie wiedział o co chodzi. Ewentualnie jak masz adnotacje typów, to też czasem ułatwia sprawę i umożliwia pewne ustępstwa.

Trzeba doliczyć tu też fakt, że ja oczywiście tutaj mocno przejaskrawiam przykład, ale chodzi o to, by pokazać pewien fakt.

Zatem jak widzisz – nazwy są krytycznie ważne, a każdemu, co tworzy kod jak ten w drugim przykładzie, trzeba zasądzić a) rentę z racji braku mózgu b) wyrok 15 lat kodzenia w legacy code napisanym w C++, jako gratis.

Także tak.

8.3 No po co mi to wszystko?

Znowu – Riedel zadał dobre pytanie. Już mówiłem – komputer jest lepszy w pamiętaniu rzeczy niż ty. To po pierwsze. Po drugie dochodzi tutaj inna kwestia – lenistwa. Załóżmy sobie, że mamy jakieś imię, które chcemy zapamiętać.

Wchodzi nam do pokoju Prorok i chce, żebyśmy mu wyprintowali kilka część jego przemowy:

`Dobry wieczór. Coś się... coś się popsuło i nie było mnie słychać, to powtórzę jeszcze raz(...)`

Dlaczego? Nie pytaj, zrób to. Szybko, szybko, zanim zdamy sobie sprawę, że to bez sensu.

```
print("Dobry wieczór. Coś się... coś się popsuło...")
print("Dobry wieczór. Coś się... coś się popsuło...")
print("Dobry wieczór. Coś się... coś się popsuło...")
print("Dobry wieczór. Coś się... coś się popsuło...")
```

i done, prawda? Tylko tak. To jest albo bardzo dużo pisania, albo dużo kopiego pasty (metoda kopiego pasty polega na kopiowaniu i wklejaniu czegoś - przypominam skromnie dla tych mniej obeznanych). Oba rozwiązania nie są za dobre.

I tu zmienne wchodzą całe na biało, jak UTF-8 w poprzednim rozdziale.

```
prophet_lyrics = "Dobry wieczór. Coś się... coś się popsuło..."
print(prophet_lyrics)
print(prophet_lyrics)
```

```
print(prophet_lyrics)
print(prophet_lyrics)
```

Inny przykład – załóżmy, że chcemy sobie jakieś dokładne obliczenia matematyczne porobić, do których potrzebujemy liczby PI z dużą dokładnością. I co, za każdym razem będziemy pisać `3.14159265359...` czy może po prostu zrobimy tak:

```
PI = 3.14159265359
print(PI)
```

Krócej, prawda? Tak mi się wydaje. To oczywiście dość słaby przykład, ale obrazuje to, co chcę przedstawić. A jeszcze taka notka: normalnie to tak nie róbcie, bo Python ma już sam w sobie zdefiniowaną PI, wystarczy ją zaimportować. Co to dokładnie znaczy, omówimy później, ale jakby w następnych rozdziałach PI była wam do czegoś potrzebna, to zróbcie tak:

```
from math import pi
print(pi)
```

I gotowe!

8.4 Znowu trochę teorii

Wróćmy do tego, co lubię. Czyli zgłębiania tego dlaczego, co i jak.

Te nasze zmienne całe. Na czym one polegają? Jak komputer je rozumie? A no dość prosto, zatem już tłumaczę.

Najpierw omówimy taki ogólny model tego, jak komputer widzi zmienne.

Otóż sprawa ma się tak, że za każdym razem, kiedy tworzymy nową zmienną, nasz komputer sprytnie sobie działa i robi coś na takiej zasadzie, że asocjuje niejako daną zmienną, a raczej jej nazwę, z jakimś konkretnym adresem w pamięci.

W zasadzie to nawet nie komputer a kompilator/interpreter, ale to tam nie wnikajmy na razie.

Co to znaczy w praktyce i z czego wynika?

Jak wcześnie już ustaliliśmy, komputer rozumie tylko zera i jedynki. Nic więcej. Musi sobie zatem wszystko tłumaczyć na rzeczy zrozumiałe dla niego.

Nie inaczej jest w przypadku zmiennych. Kiedy w kodzie zapisujemy coś pokroju:

```
new_variable = "TEXT"
```

Pod spodem interpreter Pythona robi sobie taki myk, który asocjuje (kojarzy) w prosty sposób kawałek tekstu, czyli `new_variable` z jakimś adresem w pamięci, jakąś lokalizacją. Bo nie wiem, czy pamiętasz, ale chwilkę temu mówiłem, że zmienne są przechowywane w pamięci. No właśnie. Zatem, żeby komputer wiedział, gdzie ma konkretnie szukać jakiejś wartości, podaje mu się adres, pod którym ta wartość się znajduje.

A jak wygląda ta pamięć komputera? Nie inaczej niż taka bardzo długa linijka z ponumerowanymi komóreczkami. Wyobraź sobie niesamowicie długi rząd komórek ustawionych obok siebie. W tych komórkach mogą znajdować się dwie wartości – 0 albo 1. Tak właśnie wygląda pamięć komputera.

Teraz w tych komórkach zapisujemy sobie nasze zmienne, dane. Tak jak mówiłem, żeby później móc ich znowu używać, żeby komputer wiedział, skąd ma zaczytać raz zapisane już dane, potrzebujemy adresu tych danych. Adres jest niczym innym, jak tak zwanym przesunięciem. To liczba bitów/bajtów (zależy od notacji), jaką należy się przesunąć od początku pamięci, by znaleźć daną wartość. Wtedy nasz sprzęt sobie tam skoczy, pod konkretny adres. Przeczyta, co musi i zwróci nam to, żebyśmy my nie musieli pamiętać.

8.5 Heksadecymalne liczby

Teraz małe wtrącenie – pamiętasz, jak mówiłem o tym, że system binarny jest nieco rozlazły? Otóż z racji tego, że chociażby pamięć adresowa w komputerze zazwyczaj ma bardzo dużo możliwych adresów, mamy coś takiego jak system szesnastkowy, czyli heksadecymalny. Podobna idea co system dwójkowy, ale zamiast dwóch cyfr, czy dziesięciu jak w dziesiętnym, mamy tutaj szesnaście.

Dlaczego szesnaście? Łatwo się przelicza pomiędzy nim a dwójkowym i jest zwięzły, bo duże liczby można wyrazić za pomocą małej liczby cyfr, gdyż bazujemy tu na potęgach szesnastki, a jeśli potęgi wchodzą w grę to wzrost/spadek jest wykładniczy, nie inaczej jest jeśli o długość zapisu idzie.

Do tego liczby w binarnym ładnie się tłumaczą na heksa.

Wszystko jest analogiczne do teorii z systemu binarnego, więc przypomnij sobie, jak to tam wyglądało np. z przeliczaniem i po prostu zrób analogicznie w systemie heksadecymalnym. Jak samodzielnie się nie uda, to pogoogluj. Jakie są tam 'cyfry'? A no takie:

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
DEC	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Czyli tak w skrócie: cyfry z dziesiętnego + 6 pierwszych liter alfabetu.

W każdym razie. Pamiętaj zatem, że pamięć zazwyczaj adresujemy za pomocą systemu heksadecymalnego, a liczby w nim zapisane zazwyczaj oznaczamy przedrostkiem 0x, analogicznie jak w przypadku binarnego, gdzie było 0b.

(Tak, wiem piękne bazgroły. Nie umiem into profesjonalne ilustracje, więc są ręczne.)

Wracając do tematu. Przeciętny komputer ma obecnie jakieś minimum 4 GB RAMU. To około 4 miliardy bajtów. Czyli 32 miliardy bitów. Sporo. Dlatego też adresujemy hexem. Adresy będą krótsze w prezentacji. Zdecydowanie krótsze.

To teraz wyobraź sobie takie 32 miliardy bitów, każdy jako jedna komóreczka, te komóreczki są koło siebie w linii ciągłej. To, w dużym uproszczeniu, tak wygląda pamięć twojego komputera.

Mała uwaga. Z racji tego, że bit to taka malutka jednostka, to obecnie raczej adresujemy za pomocą bajtów. Czyli adres w pamięci – nic innego jak liczba, jest liczbą bajtów, o które trzeba się przesunąć od początku pamięci, by dorwać się do danej wartości.

Zatem jeśli interpreter kojarzy nam, że `new_variable` to ogółem adres `0x123`, to za każdym razem, kiedy będziemy się odwoływać do `new_variable`, nasz interpreter przesunie się o `0x123` bajty od początku pamięci i stamtąd sobie weźmie wartość.

Tylko moment, chwila...

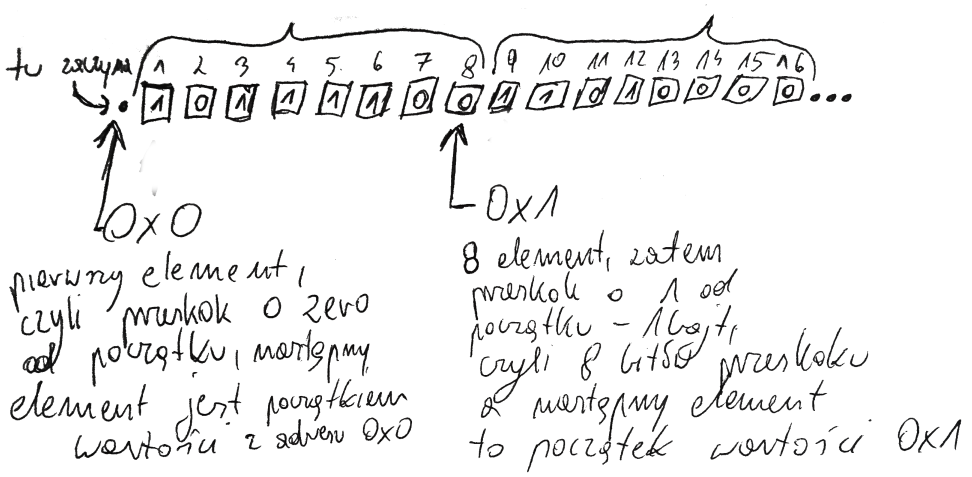

Figure 1: Przedstawienie pamięci

8.6 Kiedy przestać czytać?

Bo adres początkowy ma, ale końcowego tak niezbyt. Co teraz? Już mówię. Znowu wrócimy sobie do C i innych archaicznych rzeczy.

Zacznijmy od takiego kodu:

```
#include <stdio.h>

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

Spróbuj przeanalizować ten kod samodzielnie i pomyśleć co tu się stało. Nie jest aż taki trudny. Generalnie zadeklarowaliśmy sobie kilka zmiennych - `a, b, c, h, z` i wydrukowaliśmy z oraz c.

Resztę omówimy niżej, ale spróbuj najpierw wywnioskować coś samodzielnie!

I teraz uwaga, zrobimy mały trik. Otóż spojrzymy na to, jaki kod wygeneruje z tego kompilator. W tym wypadku `x86-64 gcc 9.3`. Popatrzmy jakie instrukcja dla procesora nasz kompilator wypluł, jaki kod `Assembly` powstał. Z racji rozmiaru kod na następnej stronie.

```
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. O co tu chodzi? Spokojnie, już omawiamy. Kawałek po kawałku.

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

To pomińmy - nie interesuje nas w tym konkretnym przykładzie.

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

To już nas interesuje. To odpowiednik naszego: `short a = 1;` I co tu się dzieje? Magiczne ilustracje na pomoc!

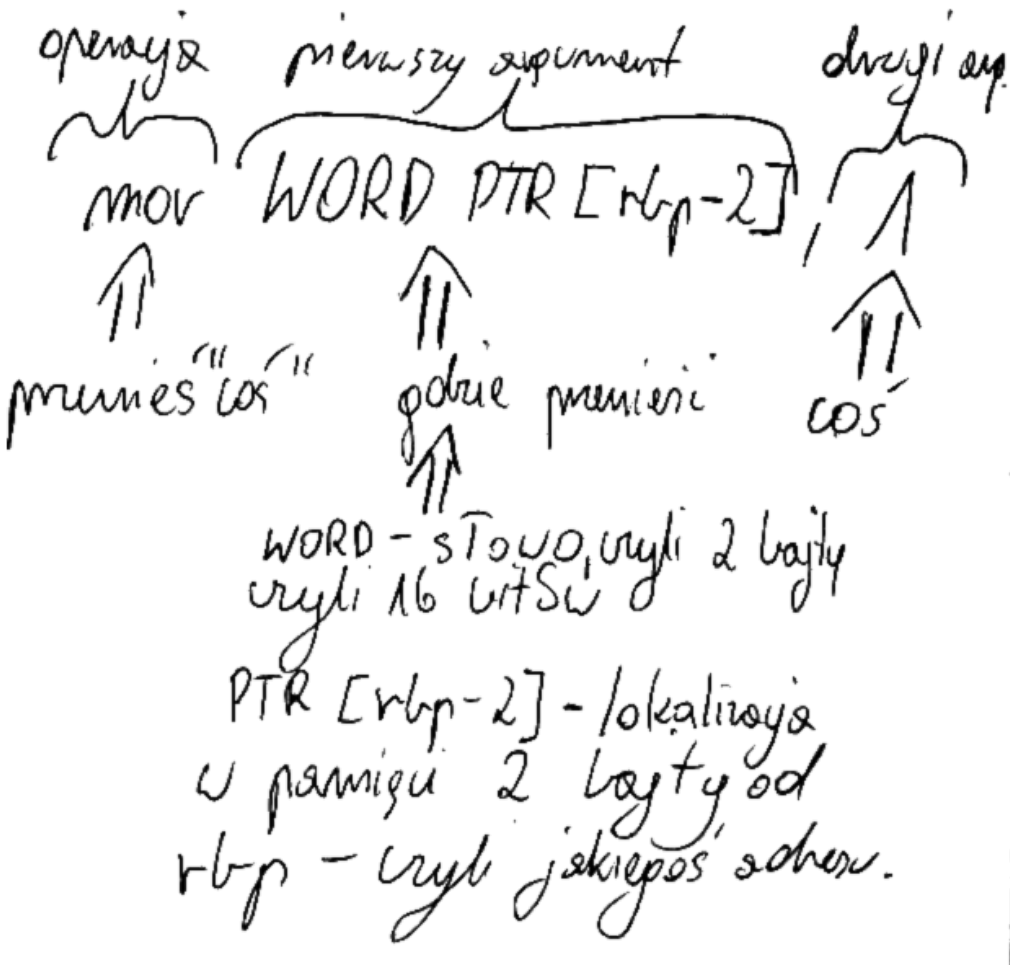

Figure 2: Rozpisane polecenie wyżej

Okej, trochę to rozpisałem na ilustracji, teraz przejdźmy do głębszych wyjaśnień.

Nasza deklaracja `short a = 1;` oznacza po prostu stworzenie zmiennej `a`, typu `short` i przypisaniu jej wartości `1`. Short to nieduża liczba całkowita, w tej implementacji ma akurat 16 bitów, czyli 2 bajty. W Pythonie podobny (na jakimś poziomie) zapis wyglądałby po prostu `a = 1`. Czyli tworzymy zmienną `a` o wartości 1. Tak to wygląda w C. Pora zejść poziom niżej. A co jest poziom niżej? Jaki język? Przypomnij sobie z poprzednich rozdziałów albo Googluj.

`mov` mówi procesorowi, żeby przenieść wartość drugiego argumentu w miejsce sprecyzowane w pierwszym argumencie.

Pierwszym argumentem jest całe to wyrażenie `WORD PTR [rbp-2]`. Oznacza ono ni mniej, ni więcej, że pod (PTR) adresem, konkretniej adresem [rbp-2], czyli `rbp` minus dwa bajty, mamy `WORD`, czyli słowo, a słowo w tej implementacji akurat to 2 bajty, czyli 16 bitów. Czym jest `rbp` - nie zajmujmy się tym teraz, niezbyt ważne. Wyobraź sobie, że to jakiś tam adres, odniesienie w pamięci, wskaźnik na coś, cokolwiek.

A jako drugi argument mamy wartość, którą trzeba tam wstawić, czyli `1`.

Przeanalizuj to sobie na spokojnie, nie jest to aż tak skomplikowane. Zwróć uwagę na to, że typ `short`, który ma rozmiar 2 bajtów w tej implementacji, czyli 16 bitów, jakoś brzmi tak podobnie do rozmiaru, którego kompilator użył przy pierwszym argumencie, tam jest `WORD` - też 2 bajty i 16 bitów. Przypadek?

Do tego wróć się trochę i popatrz na dalszą część tego kodu, zwłaszcza deklaracje następnych zmiennych, ich typy i fragmenty z X PTR [rbp-XD].

Po tym wszystkim powinna ci się lampka zaświecić. Odpowiadając na pytanie z początku rozdziału - skąd program wie, kiedy przestać czytać? Otóż w procesie kompilacji znika coś takiego jak `a`. Jego wystąpienia zastępowane są czymś pokroju `WORD PTR [rbp-2]`. Mając to z kolei, program doskonale wie, kiedy przestać czytać i kiedy zacząć, bo mamy i adres, i ilość bajtów jaką odczytać.

Spędź chwilę czasu nad tym, pomyśl. Nie musi od razu zaskoczyć. Przeanalizuj najpierw cały ten kod. Spokojnie. Dopiero potem przejdź dalej.

8.7 Wszystko fajnie, ale w Pythonie inaczej

Teraz tak - muszę zrobić tutaj bardzo ważną adnotację. Otóż cały czas w przykładach z tego rozdziału używam kodu z języka C i opisuję proces, który tam zachodzi przy kompilacji i tak dalej.

W Pythonie sprawa wygląda jednak trochę inaczej, jak już mówiłem. Wynika to z różnych natur tych dwóch języków, statycznie kompilowany C i dynamicznie interpretowany Python. Ta wiedza, którą ci tu przekazuje, jest jednak uniwersalna, bo o ile na wyższym poziomie, czyli tym, co jest płytko pod powierzchnią, Python tak nie wygląda, to na najniższym poziomie już jak najbardziej - w końcu CPython napisany jest w.... C.

W Pythonie mechanizm deklaracji zmiennych wygląda troszkę inaczej (jak, to porozmawiamy innym razem) i tak dalej, ale logika jest, powiedzmy, gdzieś tam zachowana. Prościej jednak jest mi tłumaczyć na przykładzie z C i Assembly. Dlaczego? Bo w Pythonie, podczas interpretacji, zastosowanych zostaje milion różnych trików optymalizacyjnych i różne dziwne rzeczy się tam dzieją. Dla przykładu - podczas samego startu Pythona inicjalizowanych jest około 50 tysięcy różnych obiektów, zmiennych, innych rzeczy! Nieźle, co? Dlatego na razie operujemy na "prostszym" C. Później pewnie wejdziemy trochę w te

optymalizacyjne Pythonowe wątki, ale na razie nie trzeba. Także jak czasami słyszysz kompilator zamiast interpreter, wydaje ci się, że coś się pomyliło i tak dalej, to nie obawiaj się.

Do tego nie przejmuj się naprawdę, jeśli powyższe nie jest dla ciebie oczywiste, ten fragment z kodem. Na spokojnie. posiedź troszkę nad nim, to nie jest takie proste! Zastanów się, przeanalizuj samodzielnie, poguglaj nawet. To, że po sekundzie nie rozumiesz, co się tam dzieje, nie sprawia, że jesteś nierozumna czy głupi. Także do przodu!

8.8 Inna inszość

Jak już o inności Pythona mówimy, to powiedzmy trochę, dosłownie dwa zdania, o tym, jak Python, jako język interpretowany, jest inny od języków kompilowanych, ale tylko pozornie.

Otóż w tradycyjnym modelu, o czym już mówiliśmy, mamy kod, następnie ten kod kompilowany jest do kodu maszynowego, potem uruchamiany. W Pythonie sprawa ma się tak, że mamy sobie zaimplementowany interpreter, który wykonuje/interpretuje nasz kod. Tak się sprawa ma, przynajmniej pozornie.

Nieco głębiej jak zajrzymy, okaże się, że... Python jest interpretowany, ale tu też zachodzi kompilacja? Ale jak to można by zapytać. O co chodzi?

Otóż kod Pythonowy, który napiszesz też jest kompilowany, ale nie do kodu maszynowego. Python kompiluje się do `bytecodu` zrozumiałego dla interpretera Pythona, coś jak maszyna wirtualna w Javie (JVM), ale inaczej.

Następnie, nasz skompilowany bytecode jest wykonywany przez interpreter Pythona, zaś interpreter Pythona to nic innego jak inny kod skompilowany do kodu maszynowego, czyli zwykły program. Mówiąc w skrócie Python to kompilowany język interpretowany tak jakby.

Zastanawiałeś się kiedyś czym są i dlaczego się tworzą pliki *.pyc po tym, jak uruchomisz swój kod? To swego rodzaju forma optymalizacji i zapamiętania przez Pythona pośredniego kroku kompilacji. Python patrzy na plik źródłowy, na jego podstawie liczy sobie jakąś tam sumę z tegoż pliku, czy 'liczbę', w końcu każdy plik pod spodem to nic innego jak jakaś tam bardzo długa binarna nawijka. Binarne ciągi zaś można przetłumaczyć, przeliczyć na zwykłą liczbę, w uproszczeniu. Zatem Python pod spodem sobie to robi, patrzy, czy istnieje plik* .pyc, jeśli nie, to go tworzy. Następnie sprawdza, czy ta liczba, która jest unikalna dla każdego pliku źródłowego, jest taka sama. Jeśli tak, to nie wykonuje kroku kompilacji ponownie, od razu przeskakuje do interpretowania.

Jeśli dokonamy jakiejkolwiek zmiany w kodzie to Python wyłapie zmianę, gdyż zmieni się ta 'liczba' i przed interpretowaniem przeprowadzi proces 'kompilacji' ponownie.

8.9 Zarządzanie pamięcią

Można sobie by zadać pytanie, skąd Python wie, kiedy 'skasować' z pamięci zmienne? Bo zaalokować, to wiemy kiedy. Skąd Python wie, kiedy zmienna nie jest nam już potrzebna i można zwolnić zaalokowaną pamięć? Gdyby Python nie wiedział i nie robił tego ręcznie, nie przeprowadzał tak zwanego `Garbage Collection`, to każde uruchomienie naszego programu zapychałoby permanentnie, przynajmniej aż do czasu ponownego uruchomienia, RAM naszego komputera. Jak to się dzieje, że to nie zachodzi w większości przypadków?

Otóż Garbage Collector w Pythonie, czyli to, co sprząta, nam już niepotrzebne zmienne i uwalnia pamięć do ponownego użycia prowadzi sobie taki spis. W tymże spisie trzyma liczbę referencji dla każdego obiektu. Jeśli ta liczba osiągnie 0 to znaczy, iż dany obiekt nie jest już potrzebny, bo nic się do niego nie odnosi, zatem można uwolnić pamięć.

Co w sytuacji gdy jeden obiekt odnosi się do drugiego, ale nigdzie indziej? Mamy wtedy pętlę zależności tak jakby. Czyli teoretycznie liczba referencji jest większa od 0, ale obiekt nie jest używany.

Tutaj wchodzi cały na biało system wykrywania pętli referencji. Python jest w stanie to wykryć i wtedy również uwolni pamięć. Linked lista i takie tam. Doczytaj samodzielnie.

Warto też dodać, że Python sprytna bestia. Ma taki mechanizm optymalizacyjny, który najczęściej sprawdza świeżo utworzone obiekty. Otóż jeśli obiekt jest świeży, to duża szansa, że zaraz nie będzie potrzebny. Stare obiekty, które przetrwały do tej pory mają dużą szansę na to, by przetrwać jeszcze dalej. Ci, co mają, zostanie im dodane, ci, którzy nie mają nic, zostanie im zabrane. Czy jakoś tak. To jak z modą i trendami. Nowe przemijają, ale stary rdzeń trwa dalej.

8.10 Podsumowanie

Z racji tego, że w tym rozdziale dość sporo informacji spłynęło na wasze głowy, moi drodzy, postanowiłem tutaj go zakończyć, nieco wcześniej niż oryginalnie planowałem.

Przypomnijmy sobie o tym, o czym pisałem w tym rozdziale.

Nasz komputer jest całkiem dobry w zapamiętywaniu rzeczy, znacznie lepszy, niż nasze mózgi, zatem warto z tego korzystać. Zazwyczaj podczas programowania do zapamiętywania na jakiś czas pewnych rzeczy wykorzystujemy **RAM**.

Przy okazji notka - wyrażenie `pamięć RAM` to pleonazm językowy, gdyż RAM znaczy `Random Access Memory` zatem pisanie pamięć RAM to jakby pisać masło maślane. Wróćmy jednak do tematu - wykorzystujemy RAM. Kiedy?

Robimy to chociażby używając **zmiennych/stałych**, które są niczym innym jak jakąś **nazwą, aliasem**, który tworzymy dla tego, co chcemy zapamiętać, przez co komputer wie, gdzie w swojej pamięci, pod jakim adresem, dokładniej mówiąc, danej rzeczy szukać, a nam jest łatwiej wpisać i zapamiętać `first_name` jako nazwę zmiennej/odniesienie/alias zamiast `0xA1FBA`.

Tworząc te `aliasy` czy też **nazywając nasze zmienne, musimy kierować się określonymi zasadami**, jak chociażby tym, że nie powinniśmy ich zaczynać od cyfr. Nazwy zmiennych powinny być opisowe, ale krótkie, podobnie z całym kodem. Ma to zasadnicze znaczenie jeśli idzie o czytelność kodu, jaki tworzymy i jego jakość.

Oprócz systemu binarnego, mamy też coś takiego jak **system heksadecymalny**, którego używamy po to, żeby zwięźlej zapisywać to, co w binarnym zajęłoby nam o wiele dłużej. Przelicza się to wszystko na podobnej zasadzie co z dziesiętnego na dwójkowy.

Komputer kojarzy sobie **adres**, rozmiar zmiennej zależnie od tego, co tam w środku jest. Na podstawie kodu Assemblera/C, plus minus dowiedzieliśmy się, jak to tam pod spodem wygląda. Tylko no właśnie, my dokładniej zbadaliśmy proces w C, w **Pythonie to wszystko wygląda nieco inaczej, bo jest interpretowany**, ale ogólna mechanika gdzieś tam pozostaje podobna, stąd dobrze ją znać.

8.11 Zadania i pytania

Teraz pora na pytania. Pamiętaj, niektóre z nich będą wymagały poszukania informacji w Internecie czy wiedzy ogólnej. To nic złego!

1. Jakie rodzaje pamięci posiada twój komputer, o których mówiliśmy? Wymień ich charakterystykę i która używana jest do przechowywania jakiego rodzaju danych?
2. Jak zadeklarować zmienną w Pythonie? Zadeklaruj ich kilka w interpreterze.
3. Jakie znasz zasady co do nazw, jakie możemy nadawać zmiennym, w Pythonie?
4. Polskie znaki w nazwach zmiennych to dobry pomysł. Czy zgadasz się z tym stwierdzeniem? Jeśli tak, to dlaczego? Jeśli nie, to dlaczego?
5. W rozdziale znajduje się kawałek kodu z funkcją o nazwie `redirect_logged_in_user`. Przetłumacz cały ten fragment linijka po linijce i swoimi słowami opisz, co on robi, co robi każdy kawałek tego kodu, albo co sądzisz, że robi. W tekście rozdziału już masz nieco opisane, ale zrób to teraz samodzielnie i wejdź głębiej w szczegóły.
6. Jak wygląda pamięć komputera, obrazowo mówiąc? Jak komputer się po niej porusza, upraszczając?
7. Co to jest system szesnastkowy?
8. Przelicz kilka liczb z systemu dziesiętnego na szesnastkowy. Jakich? 2, 8, 19, 32, 111.
9. Ile RAMu ma twój komputer? Ile to bajtów? Zwróć uwagę czy mówimy o bitach czy bajtach ;) Jak wyrazić tę liczbę w systemie szesnastkowym? A jak w binarnym?
10. Skąd komputer wie, gdzie szukać wartości, jaką zapisaliśmy dla danej zmiennej?
11. Skąd komputer wie, kiedy przestać czytać wartość pod danym adresem?
12. W kodzie z podpunktu `8.6` znajduje się fragment owiany tajemnicą - wartość dla zmiennej `z` nie jest wyrażona. Analizując kod asemblera znajdujący się poniżej, spróbuj zgadnąć, jaka wartość została tam przypisana. Podpowiem tylko, że typ `char` to nic innego jak jakiś `character`, czyli znak. Podpowiedź: przypomnij sobie trochę o tym jak przedstawiamy/kodujemy znaki/tekst.

8.12 Odpowiedzi

1. Chodzi mi o RAM i pamięć dysku twardego. Ich charakterystyki z kolei to już sobie wyszukajcie. W skrócie RAM jest zazwyczaj (uogólniając) szybszy, ale mniej persystentny, dysk na odwrót.
2. `nazwa_zmiennej = "wartość"` dla przykładu.
3. Póki co mówiliśmy tylko o tym, by nie zaczynać od cyfr na przykład a od liter lub podkreślenia.
4. Nie jest to dobry pomysł. Dlaczego? Niekoniecznie powinniśmy pisać kod po polsku, poza naprawdę nielicznymi wyjątkami.
5. Tutaj szczegółów już nie dorzucam, niech to będzie małe wyzwanie ;)
6. Plus minus można o tym myśleć jako ciągu położonych w linii prostej kolejnych komórek, zawierających jedynki lub 0.
7. System zapisu liczb bazujący na 16 jako podstawie.
8. Znowu - samodzielnie.
9. Patrz wyżej.

10. Po adresie, który sobie ogarnia `pod spodem` i stamtąd odczytuje - w tym konkretnym miejscu w pamięci.
11. O tym mówiliśmy w `8.6`.
12. Jeśli przeanalizujemy linijkę `BYTE PTR [rbp-5], 102` możemy dojść do wniosku, że ta definicja `chara`, mówi coś o liczbie 102. 102 w ASCII/UNICODE to nic innego jak `f`.

Pamiętaj, że Twoje odpowiedzi możesz wrzucić na GH o tutaj - https://github.com/grski/junior-python-exercises, a sprawdzę twoje rozwiązania i dam feedback. Więcej o tym podrozdziale 'Część interaktywna'.

9 Typy danych

W poprzednim rozdziale mówiliśmy sobie o zmiennych. Będąc przy tym, siłą rzeczy warto wspomnieć o typach danych jakie nasze zmienne mogą przechowywać w Pythonie. Porozmawiamy sobie o tym konkretnym języku, ale podobny podział występuje w większości języków.

Jak już w poprzednich rozdziałach pisałem, Python nie wymaga od nas definiowania typów dla naszych zmiennych, nie ma w nim statycznego typowania. Przypomnij sobie co to oznaczało i odpowiedz na pytanie - co oznacza brak statycznego typowania? Czym jest dynamiczne typowanie? Jakie są tego wady/zalety. Poszukaj w książce, może w odpowiedziach do poprzednich rozdziałów. Zrób to teraz.

Mimo tego dobrze wiedzieć, na jakie typy zazwyczaj dzielimy sobie różne zmienne. Dlaczego? Bo Python też ich używa, tylko sam niejako zgaduje jakiego typu użyliśmy. Zależnie od typu na zmiennej można wykonywać różne operacje. Jak sobie o tym pomyślimy, to jest to logiczne, bo mimo tego, że pod spodem to wszystko to samo - kod binarny, to na pewnych fragmentach, które interpretujemy jako X, chcemy wykonywać tylko operacje ze zbioru Z, a gdy interpretujemy G, to tylko z F. Mówiąc prościej jak coś oznaczamy jako tekst, to inaczej to potraktujemy, czy inne modyfikacje będziemy mogli zastosować, niż w sytuacji, gdy coś jest liczbą. Na liczbach możemy wykonywać operacje arytmetyczne dla odmiany a w tekście chociażby szukać swojego imienia. Zależnie od typu inne operacje. Logiczne, prawda?

A więc w Pythonie wyróżniamy między innymi następujące podstawowe typy danych:

1. Liczby
2. Napisy
3. Bajty
4. Typ logiczny/boolowski

Z bardziej złożonych mamy:

1. Listy
2. Krotki
3. Słowniki
4. Zbiory

Zaraz je sobie wszystkie po kolei omówimy. Zacznijmy od liczb, bo będzie najdłużej prawdopodobnie.

9.1 Liczby

9.1.1 Krótka charakterystyka

No to tak moi drodzy, w Pythonie wyróżniamy sobie trzy główne typy liczb: liczby całkowite, liczby zmiennoprzecinkowe i liczby złożone, czyli po kolei tak jakby inty, floaty, complex.

Co to znaczy?

9.1.2 Liczby całkowite

Tu sprawa jest chyba prosta, prawda? `1, -1, 5, 0, 938, -24861` to przykłady liczb całkowitych, czyli po prostu `zwykłe` liczby bez żadnych przecinków, udziwnień i wynalazków. Więcej na ich temat rozpisywał się jakoś zbytnio nie będę bo i nie ma po co.

Czy na pewno?

Zwróć uwagę na to, że w przykładach wyżej pojawia nam się liczba ujemna, mniejsza od zera. W przypadku tekstu zapisać jest ją prosto, stawiamy minus przed i gotowe. Jako ludzie nauczeni jesteśmy, by interpretować to jako liczbę ujemną. Jak natomiast robi to komputer?

9.1.3 Przykładowy sposób reprezentacji liczb ujemnych

Przypomnijmy sobie poprzednie rozdziały i to, o czym tam mówiliśmy. Pokazywałem między innymi to, jak komputer zapisuje sobie liczby w pamięci i jak je reprezentuje, a mianowicie za pomocą systemu binarnego, bity, bajty, te sprawy. Teraz tak, pojawia się pytanie: jak w takim razie komputer oznacza, że dana liczba jest ujemna? Przecież `minusa` sobie nie postawi w pamięci w jakiś sposób.

Otóż przedstawię wam, jak to wygląda, znowu, w C. Sposobów i pomysłów jak rozwiązać ten problem było wiele, dalej jest, ale my omówmy tylko jeden. Załóżmy, że operujemy sobie jakimś typem, który jest akurat wielkości 1 bajta. Znaczy to tyle, że ma 8 bitów długości. Ile w takim razie obsługuje maksymalnie wartości/liczb? Odpowiedz na to pytanie proszę, przelicz.

Dobrze, mając 8 bitów, mamy do dyspozycji 8 zer i/lub jedynek. Czyli możemy maksymalnie przedstawić 256 wartości, prawda? Czyli na przykład liczby od 0, do 255. A no nie do końca!

W domyślnym przypadku będziemy mieć do dyspozycji 256 wartości, prawda, ale z innego zakresu: od **-128** do **127**. Można to określić wzorem: od -2^{n-1} do $2^{n-1}-1$, gdzie n to liczba bitów, czyli dla 8 bitów: od -2^7 do 2^7-1

Skąd ta zmiana? A no stąd, że zabieramy sobie jeden bit by oznaczyć czy dana liczba jest dodatnia, czy ujemna, tak w dużym skrócie. W C, jeśli wiemy, że nie interesują nas wartości ujemne, możemy powiedzieć kompilatorowi, żeby przesunął zakres ujemny na dodatki. Zmienne ze znakiem vs zmienne bez. Signed variables vs unsigned variables.

Swoją drogą jak już jesteśmy przy tym to dorzucę jeszcze jedną ciekawostkę. Wiesz, że nawet sposób zapisu kolejności bitów w pamięci jest umowny? Co to znaczy? Otóż niektórzy ludzie nie byli w stanie dogadać się co jest lepsze, zapisywanie bita o najwyższej wartości pierwszego czy ostatniego. Stąd też mamy dwa standardy: big endian (grubokońcowość) i little endian (cienkokońcowość). Co to znaczy i jak wygląda w praktyce? Prosta rzecz.

Załóżmy, że rozmawiamy o Big Endian. Chcemy pod adresem 100 wpisać wartość np. 0x4A3B2C1D. Wyglądałoby to tak.

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

A Little Endian?

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

Czyli odwrotnie. Chodzi generalnie o to, który zapisać gdzie. Robi nam to różnicę przy przeliczaniu/odczytywaniu tych wartości. Która lepsza? Łatwiejsze do ogarnięcia będzie pewnie Big Endian, gdyż jest analogiczna do zapisu jakiego używamy na co dzień w systemie dziesiętnym.

Różne procesory mają różne konwencje, całe szczęście ty nie musisz się tym martwić w swoim kodzie - interpreter Pythona zrobi to za ciebie.

9.1.4 Liczby zmiennoprzecinkowe i niedokładność ich reprezentacji

Z czym do czynienia mamy tutaj? Nic innego aniżeli liczby z **częścią po przecinku** mówiąc prostym językiem. I w zasadzie tyle. Jeśli widzimy gdzieś w Pythonie liczbę, gdzie występuje sobie kropeczka - np. `1.0`, to musimy wiedzieć, że mamy do czynienia z liczbą zmiennoprzecinkową. Dlaczego to ważna wiedza? Otóż...

Tutaj, w przeciwieństwie do liczb całkowitych, udziwnienia są i to duże, ale pod spodem. Dłuższy temat, ale generalnie sprowadza się to do tak zwanej niedokładności reprezentacji liczb zmiennoprzecinkowych w systemie binarnym. Tak tak. Jasne, prawda? Powiem tylko tajemniczo, żebyście pamiętali zawsze i wszędzie, że używanie zwykłych floatów/doublów do dokładnych obliczeń czy przechowywania informacji o pieniądzach, to niezbyt dobry pomysł, gdyż czasami `0.1+0.2 != 0.3`. Dlaczego? Bo spróbuj za pomocą potęg dwójki dokładnie przedstawić np. 1/3. Ciężko, co? Ale o co mi w sumie chodzi?

Rozważmy prosty program w C(sprawa dotyczy praktycznie każdego języka):

```c
#include <stdio.h>

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

Prosty kod, prawda? Myślę, że każdy powinien go zrozumieć, jeśli zna choćby podstawy programowania. Oczekiwanym przez sporą część wynikiem działania tego kodu byłoby wydrukowanie 'Equal' w konsoli, racja? Ja też oczywiście tak myślałem na początku. Sprawdźcie jednak sami, co się stanie gdy kod skompilujecie i uruchomicie.

O dziwo "Equal" się nie wyświetliło. Dlaczego? Coś się pomyliło? Liczby pozornie te same, no bo i tu 0.1 i tu 0.1, co jest? Hm, może zmienną nam źle zapisało. Wypiszmy ją sobie i zobaczmy.

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

Dodajcie sobie tę linijkę kodu po zakończonym ifie. Uruchomcie kod... I co? Oto wynik:

```
0.100000
```

Chwila. Czyli jednak coś źle działa w naszym programie, prawda? No bo `example_float` jest przecież równy 0.1, prawda? A no nie.

Tutaj tego nie widać, bo precyzja jest zbyt niska, ale poprawny to, zmuśmy funkcję `printf` do wyświetlenia naszego floata z większą dokładnością niż domyślna, bo jak widzicie, printf domyślnie wyświetla tylko 6 cyfr po przecinku.

```c
#include <stdio.h>

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

Daje nam

```
0.1000000014901161
```

Lekka modyfikacja naszego kodu i wszystko jasne. Nasz `example_float` nie jest równy dokładnie 0.1, tylko troszkę więcej. Dlaczego?

Wszystko wynika stąd, że komputer 'operuje' na języku binarnym. Oznacza to, że przy tworzeniu liczb dostępne są jedynie potęgi dwójki, mnożone odpowiednio przez 1 lub 0, które można sumować(tak w dużym uproszczeniu, mówiliśmy o tym już). Nic zatem dziwnego, że nasz float tak wygląda. No bo spróbujcie z takich liczb `{..., 1/128, 1/64, 1/32, 1/16, 1/8, 1/4, 1/2, 0, 1, 2, 4, 8, 16, ...}` zbudować dokładnie 0.1. Nie da się tego zazwyczaj zrobić idealnie. Teoretycznie w wyimaginowanym świecie, gdzie mielibyśmy nieskończoną ilość pamięci do dyspozycji i nieskończoną ilość czasu, to moglibyśmy zbliżyć się nieskończenie blisko, nawet ją osiągnąć czasem, do dowolnej liczby. Ale o tym jak chcecie więcej, to poczytajcie sobie o limitach albo przypomnijcie z liceum i z matematyki.

Stąd ta niedokładność - wynika ona jedynie z tego jak reprezentowane są liczby zmiennoprzecinkowe w pamięci komputera. O ile w większości przypadków, za pomocą skończonej ilości pamięci można uzyskać zadowalającą dokładność, tak są takie przypadki, gdzie niestety ta dokładność nie będzie wystarczająca.

Do takich przypadków mamy specjalne biblioteki czy też może specjalne podejście, które inaczej zajmuje się tematem, niemniej jednak warto o tym wiedzieć. Dlatego też, jeśli piszemy jakiś program, który cokolwiek ma wspólnego z pieniędzmi, warto zastanowić

się dwa razy zanim użyjemy floata czy doubla. Może lepiej złotówki trzymać w oddzielnym incie, a grosze w oddzielnym? Who knows.

Rozwiązanie wielu problemów związanych z liczbami zmiennoprzecinkowymi znajdziemy w modułach `decimal` i `fractions`.

Do poczytania:

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

9.1.5 Liczby złożone

Bardzo rzadko spotykane, ale czasem jak ktoś robi obliczenia naukowe jakieś albo inne dziwne rzeczy, to może mu się ta wiedza przydać - otóż są to liczby składające się z części rzeczywistej i urojonej. Matematyczne tematy. Jak nie wiesz o co chodzi, to nie przejmuj się.

Definiujemy je tak: `test = 21 +3j` albo `some_complex_number = complex(32, 3)`. I możemy wykonywać na nich takie same operacje jak na liczbach - dzielenie, mnożenie, dodawanie i tak dalej. Czasami przydatne.

I tyle. Na razie tyle potrzeba ci wiedzieć jeśli idzie o rzeczy, którymi kwalifikujemy jako liczby w Pythonie. Są jeszcze np. decimale czy rationalsy, ale o nich innych razem!

9.1.6 Operacje na liczbach

Jak wspomniałem przy okazji liczb złożonych, na liczbach możemy wykonywać wszelakie podstawowe operacje matematyczne i zadziałają one plus minus tak, jak byśmy tego oczekiwali. Mówię tutaj o dodawaniu, odejmowaniu, mnożeniu, dzieleniu, dzieleniu całkowitym, operacji modulo i innych podstawowych operacjach arytmetycznych.

Operatory, jakie Python rozumie, to:

Operator	Akcja
*	Mnożenie
**	Potęgowanie, czyli 2 ** 3, znaczy dwa do potęgi trzeciej.
/	Zwykłe dzielenie zmiennoprzecinkowe
+	Dodawanie
-	Odejmowanie
//	Dzielenie bez reszty, czyli 5 // 2 równa się 2, -11 // 3 równa się -4 (zwróć uwagę na negatywne liczby)
%	Reszta z, czyli np. 5 % 2, to nic innego jak reszta z dzielenia 5 przez 2, czyli 1

Warto też pamiętać o tym, że jeśli w jednym wyrażeniu mamy kilka różnych typów liczb, to Python zrzuci wynik całego wyrażenia do najbardziej złożonego typu. Co to znaczy? Wynikiem `1.0 + 1` będzie nie `2` a `2.0`, wynikiem `1 + 1.0 + 3 +0j` będzie `5+0j`. Kolejność złożoności jest zatem taka:

Całkowite -> Zmiennoprzecinkowe -> Złożone.

Warto o tym pamiętać, gdyż niesie to za sobą pewne konsekwencje - po prostu uważaj na te kropeczki, bo czasami można się naciąć.

Dodatkowo wspomnę jeszcze o lukrze składniowym, który pozwala nam na skrócenie zapisu popularnych operacji. Wygląda to tak:

```
foo = 2
foo = foo + 2 # ta linijka będzie równoważna z tą niżej
foo += 2
```

9.1.7 Konwersje liczbowe

Liczby można pomiędzy sobą 'konwertować', czy też dokonywać swego rodzaju `rzutowania typów` jakbyśmy powiedzieli w innych silnie typowanych statycznych językach. Jak? Niżej przykłady. Przeanalizuj je sobie kapkę sam.

```
>>> 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'
```

Pobaw się funkcjami `int`, `float`, `complex`. Następnie opisz co każda z nich robi. Jakie przyjmuje wartości, co powoduje błędy. Ostatni przykład `int("1.3")` spowodował błąd. Przetłumacz go i spróbuj wyjaśnić co tu się stało.

Warto zaznaczyć jedną ważną rzecz, którą Python dostarcza od ręki, a niekoniecznie każdy język tak ma. Otóż wszystkie te funkcje rozumieją, że `"1"` to 1. W innych językach bywa inaczej. Dlaczego. Otóż podczas rzutowania typów następuje operacja często bezpośrednio na wartości w pamięci. Przypomnijmy sobie ten fragment o tym, jak komputer przechowuje tekst w pamięci. ASCII, UTF-8, Unicode i tak dalej. Wróć do poprzednich rozdziałów jak potrzebujesz.

No właśnie, zatem jak? Jako liczby odpowiednio później mapowane. A zatem, w innych językach bywa tak, że zamiast interpretować `"1"` jako 1, to często rzutuje się to do numerycznej wartości która kryje się za znakiem `1` w danym zestawie znaków/kodowaniu. W naszym przypadku `"1"`, czyli jedynka w tekście jest oznaczana nie jako `0b1` ale jako `49` czyli `0b110001` lub `0x31`.

Przy okazji - mała notka. Do szybkiego przeliczania mogą zainteresować cię funkcje, które widać w kawałku niżej.

```
>>> ord('1')
49
>>> bin(49)
'0b110001'
>>> hex(49)
'0x31'
>>> oct(49)
'0o61'
>>> chr(49)
'1'
```

Czyli `ord`, `bin`, `hex`, `oct`, `chr`. Pobaw się i poczytaj. Gdzie? W dokumentacji Pythona(https://docs.python.org/3/). Najlepiej po angielsku. Swoje wnioski i efekty zabawy podsumuj pisząc artykuł, w którym opisujesz o co chodzi z którą funkcją, po krótce scharakteryzujesz każdy z typów. Podasz przykłady dla których funkcje nie działają i domysły dlaczego. Ewentualnie skorzystaj z funkcji `help` np. `help(int)`

Do tego pokażę ci mały trik:

```
>>> dir(float)  # alternatywnie: dir(1.0)
['__abs__', '__add__', '__bool__', '__ceil__', '__class__',
 '__delattr__', '__divmod__', '__doc__', '__eq__', '__float__',
 '__floor__', '__floordiv__', '__format__', '__ge__',
 '__getattribute__', '__getnewargs__', '__getstate__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__int__', '__le__',
 '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__',
 '__pos__', '__pow__', '__radd__', '__rdivmod__', '__reduce__',
 '__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__', '__rmul__',
 '__round__', '__rpow__', '__rsub__', '__rtruediv__', '__setattr__',
 '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__',
 '__trunc__', 'as_integer_ratio', 'conjugate', 'fromhex', 'hex',
 'imag', 'is_integer', 'real']
>>> dir(str)  # alternatywnie: dir("text")
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__getnewargs__', '__getstate__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__',
 '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier',
 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind',
 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
 'translate', 'upper', 'zfill']
```

Otóż funkcję `dir`. Funkcja dir to funkcja, która zwraca wszystkie dostępne metody/atrybuty danego obiektu.

Póki co nie zajmuj się tymi, które zaczynają się od `__` czy `_` a skup na tych, które zaczynają się od normalnych liter. Czym jednak one są? Metody zaczynające się od `__` to tak zwane Python Magic Methods/Dunder Methods/Metody Magiczne. To coś o czym porozmawiamy później, ale to takie specjalne rodzaje metod/funkcji danego obiektu, które mają spełniać określone role. Te, które zaczynają się od pojedynczego podkreślenia `_`, są metodami prywatnymi.

W Pythonie nie ma enkapsulacji, co znaczy, że generalnie jak dodamy jakiś atrybut/metodę do klasy/obiektu, to nie możemy jakoś bardzo skutecznie zabronić innym wołać, nawet jeśli chcemy by użytkownik nie miał możliwości tego zrobić, gdyż np. dana metoda jest tylko pomocnicza, **prywatna**. Konwencja zatem mówi, byśmy dawali podkreślenie przed prywatnymi zmiennymi, metodami a my jako programiści nie powinniśmy używać takowych o ile nie jest to wewnątrz definicji. Porozmawiamy o tym jeszcze później. W międzyczasie możesz sobie pogooglować o tej całej enkapsulacji.

Podsumowując: za pomocą `dir` możesz sprawdzić, co na danym obiekcie można robić, jakie ma metody/funkcje etc. Przydatne.

Pobaw się tym teraz znanych ci typach.

9.1.8 Przykłady podstawowych operacji na liczbach

```
>>> 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)
```

9.2 Łańcuch znaków

9.2.1 Krótka charakterystyka

Napisy/tekst, łańcuch znaków, czyli tak zwane stringi. To po prostu tekst. Zazwyczaj przynajmniej, bo mogą to też być jakieś tam zbiory bajtów tak jakby, ale o tym też przy innej okazji.

W Pythonie3 dla zwykłych stringów króluje UNICODE i zazwyczaj utf-8. Oczywiście można używać innych kodowań korzystając z odpowiednich metod, ale póki co chyba nie mamy się co tym martwić. W Pythonie 2 było nieco inaczej, więc tylko i wyłącznie dla wiedzy historycznej o tym wspomnę i nie będę się zbytnio rozwodził nad tym jak proces tam wyglądał, gdyż jest to już nieco archaiczne podejście a wszystkie nowoczesne języki programowania zakładają wykorzystanie UNICODE i utf-8.

Stringi w pythonie deklarujemy w następujący sposób

```
some_text = "foo"  # pierwszy sposób
another_text = 'bar'  # drugi sposób z użyciem ' zamiast "
longer_text = """hahah
another line """
```

A zatem jest to po prostu tekst otoczony " lub '. Python zezwala na użycie zarówno podwójnego jak i pojedynczego 'ciapka'. Moją praktyką jest, by preferować ". Technicznie rzecz ujmując standard pozwala na to by używać zarówno jednego jak i drugiego, byle nie mieszać ich ze sobą w jednym projekcie. Co to znaczy?

Jeśli mamy już jakiś codebase/kod/projekt, i postanowimy używać ' zamiast " to okej, niechże tak będzie, chociaż nie jest to moja preferencja, ale nie mieszajmy stylów. Konwencją jest, by trzymać się jednego i mieć ujednolicony styl w całym kodzie. To znaczy, że jak gdzieś zaczniemy używać pojedynczych ciapków, to używajmy ich już wszędzie w danym projekcie. Jak podwójnych to podwójnych. Wspominałem już, że preferuje podwójne i uważam je za lepsze? Tak samo uważają twórcy narzędzia do formatowania kodu - `black` plus obiektywnie podwójne ciapki mają swoje zalety typu lepsza czytelność czy też łatwiejsze używanie razem z językiem angielskim gdzie mamy sporo znaków ' w tekście. Nie trzeba dodawać znaku ucieczki.

A no właśnie, znak ucieczki. Porozmawiajmy sobie o znakach specjalnych. Co jeśli w naszym tekście, który zdefiniowaliśmy za pomocą ", będziemy potrzebowali użyć też tego znaku? Spróbujmy.

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

Podany kod nie zadziała. Dlaczego? Python nie potrafi domyśleć się, że chcesz akurat tutaj cytować i ten znak powinien traktowany być specjalnie, a nie tak jak zwykle. Trzeba mu o tym powiedzieć. Jak? Prosta rzecz.

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

Zwyczajnie wystarczy dodać \ przed danym znakiem, który chcemy traktować w specjalny sposób. Teraz może ci kapkę zaświtać dlaczego " > '. Otóż w języku angielskim często występuje pojedynczy ciapek. Jeśli używamy go do definiowania stringów to pojawia się problem w postaci tego, że musimy często stosować znak ucieczki. Jak użyjemy podwójnego to już rzadziej. A zatem powodem jest lenistwo i czytelność kodu, czyli w sumie też lenistwo. Yay!

Anyway, wróćmy do głównego tematu.

Tak definiowane stringi muszą się mieścić w jednej linijce. Jeśli chcemy, by nasz tekst był wielolinijkowy w kodzie, to musimy użyć Potrójnego znaku - zatem albo """ albo ''' zamiast jednego. To spowoduje, ze python będzie czytał nie tylko do końca linii ale aż do znalezienia znaku zamykającego, który może być już w innej linii.

Warto też zaznaczyć, że do komentowania w kodzie używamy do jednolinijkowych komentarzy używamy # a wielolinijkowych stringów jako wielolinijkowych komentarzy.

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

Przykład tego wyżej.

Pamiętaj o tym, by użyć `dir` na stringach i przejrzeć sobie podstawowe metody jakie Python posiada w bibliotece standardowej, które pozwalają nam manipulować stringami.

9.2.2 Pojedyncze znaki

Hm, skoro stringi to łańcuchy znaków to co z pojedynczymi ogniwami tychże łańcuchów? Otóż wyobraź sobie, że po łańcuchach tekstów można iterować jak po liście, plus mamy dostęp do jej poszczególnych elementów, do slicingu etc. NEAT!

9.2.3 Zmienne w tekście

Oprócz prostych stringów, które po prostu zawierają hardcodowany tekst, czyli dla przykładu:

```
name = "Aryo"
```

Istnieje możliwość przeprowadzania operacji na stringach, które umożliwiają nam wstawianie do tekstu zmiennych, dodawania stringów etc. Istnieje kilka sposobów by to osiągnąć. Zamiast się rozpisywać, po prostu zaprezentuję.

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

Pierwszy sposób nazywamy f-stringami. Są eleganckie. Piękne. Kozackie. Prawilne.

Druga opcja, to funkcja format.

Trzecia zaś to tak zwana konkatenacja.

Którego używać i kiedy? F-stringi ftw. Format spoko, jak nie można inaczej to konkatenacja.

9.2.4 Używanie zmiennych w tekście - wydajność

Jestem dość dużym zwolennikiem f-stringów w Pythonie. Podobają mi się one, są eleganckie, czytelne i proste w użyciu. Ciekawiło mnie jednak, jak wypadają jeśli chodzi o wydajność pod spodem, gdyż cóż, ta elegancja pewnie musi mieć jakiś ukryty koszt. Nic w życiu nie ma za darmo, prawda? Postanowiłem to sprawdzić i zestawić ze sobą różne metody manipulacji stringów w Pythonie pod względem wydajności.

W konkurencji znalazły się: f-stringi, konkatenacja (dodawanie) stringów, metoda join(), metoda format(). W zestawieniu nie znalazł się operator %. Dlaczego? Nie przepadam za nim tak szczerze. Moja osobista preferencja. Uważam, że powinno się go raczej unikać z pewnych względów. Relikt przeszłości, mamy dziś lepsze rozwiązania.

9.2.4.1 Metodyka testowania Testował będę za pomocą modułu timeit wbudowanego w Pythona, wywołując polecenia z terminala. Wszystkie zmienne wykorzystywane w modyfikowanym stringu, będą definiowane i ładowane zanim rozpocznie się mierzenie czasu. Każde polecenie będzie uruchamiane pętlach o 1000000 iteracjach,

każda taka pętla będzie uruchomiona 3 razy. Z tejże pętli wyłoniony zostanie najkrótszy przebieg pojedynczej iteracji. Przejdźmy do samego testowania.

9.2.4.2 Porównanie. Zaczynajmy zatem. Poniżej kod, jakiego użyłem. Wybaczcie prymitywne nazwy zmiennych, ale pisałem go kompletnie na kolanie.

```
python3 -m timeit -s "x = 'f'; y = 'z'" "f'{x} {y}'" # f-string
python3 -m timeit -s "x = 'f'; y = 'z'" "x + ' ' + y" # konkatenacja
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
```

Wszystko raczej proste. Rozważyłem dwie opcje, jedną standardową drugą taką, gdzie znalezienie metody nastąpi w setupie, a w czasie mierzonym tylko i wyłącznie jej wywołanie.

Co mam na myśli, kiedy mówię, że metoda zostanie znaleziona za pomocą operatora .? Otóż Python tam pod spodem robi sobie tak, że atrybuty danej klasy/nazwy metod i tak dalej, trzyma sobie zhashowane w słowniku. Zatem gdy piszemy `obiekt.atrybut`, pod spodem leci sobie szukanie w słowniku/dictionary lookup czy coś takiego jest w tej klasie. To oczywiście dodaje to czasu wykonania bo same instrukcje lookupu zajmują czas, co prawda niedużo, prawie nic, ale wciąż, do tego dochodzi jeszcze czas potrzebny na zaalokowanie pamięci i dodanie elementów do dicta pod spodem przy konstruowaniu instancji. Dla pewności zatem przebadałem różne przypadki. Zaznaczę jednak, że w przypadku produkcyjnego kodu raczej wzbraniaj się przed tego typu optymalizacjami, dobrze? Na juniorskim poziomie to raczej rzadko zdarzy się, iż będziesz przetwarzał tak duże zbiory danych i twój kod będzie wymagał takiej wydajności, by takowe rzeczy odwalać. Tak ku przestrodze. Anyway.

Podobnie zrobiłem z joinem i formatem. Tu rozważyłem dwie opcje - normalne wywołanie z lookupem i to bez niego.

A oto wyniki:

```
f-string: 10000000 loops, best of 3: 0.0791 usec per loop
konkatenacja: 10000000 loops, best of 3: 0.0985 usec per loop
join bez lookupu: 10000000 loops, best of 3: 0.112 usec per loop
join: 10000000 loops, best of 3: 0.144 usec per loop
format bez lookupu: 1000000 loops, best of 3: 0.232 usec per loop
format: 1000000 loops, best of 3: 0.264 usec per loop
```

9.2.4.3 Zaskoczenie Powiem szczerze, że nie spodziewałem się tego, że f-stringi są nie tylko eleganckim rozwiązaniem, ale i najszybszym! Bardzo mnie to cieszy. Na drugim miejscu uplasowała się konkatenacja, join bez lookupu, join, format bez lookupu, format, a na samym końcu template string. Z racji tego, że optymalizacja, którą poczyniłem, jest dość niepraktyczna i raczej w kodzie nikt takich potworów nie będzie tworzył poza pewnymi wyjątkami, które być może powinny być napisane w C a nie w Pythonie, to nie umieszczam wyników bez lookupów w rankingu, który wygląda tak:

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

9.2.4.4 Nieco bardziej skomplikowany przykład Pokazałem prosty przykład - wstawienie dwóch zmiennych oddzielonych spacją. Co jeśli zmiennych mamy nieco więcej niż 2? Załóżmy przypadek z 13 zmiennymi, które chcemy połączyć spacją. Kod:

```
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}"
# konkatenacja
"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))
```

Kod wyżej uruchomiłem analogicznie do poprzedniego razu.
Ciekawi mnie jak tutaj sytuacja będzie wyglądała.
Wyniki:

```
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
```

Bazując na poprzednich wynikach, nie zdziwiły mnie one za bardzo. Dlaczego? Zacznijmy od tego, co się zmieniło. Join wskoczył z 3. miejsca na 1. Konkatenacja spadła z 2. na przedostatnie. Format na 3. z czwartego. W sumie dość zasadne, dlaczego.

Pierwsze miejsce join w takiej sytuacji jest oczywiste - popatrzmy co tam robimy - joinujemy tak jakby ze sobą wiele stringów ze wspólnym stringiem, czyli dokładnie to, do czego join został stworzony. Jestem niemalże pewnym, iż pod spodem na poziomie implementacji metody czy nawet interpretera są zrobione pod to optymalizacje, dzięki czemu join świetnie poradzi sobie z dużą ilością argumentów. Cieszy mnie to - ponownie rozwiązanie, które w tym przypadku wygląda najbardziej elegancko, wypada pierwsze.

Drugie miejsce f-string. Tutaj też się nie zdziwiłem. Dlaczego? Otóż f-stringi, pierwotnie co prawda były wolne, bardzo wolne, - w pierwszej implementacji były one "kompilowane" na nic innego jak zbiór odpowiednich joinów albo formatów, nie pamiętam. Niemniej jednak w kolejnej implementacji f-stringi doczekały się własnego, zoptymalizowanego OPCODE w CPythonie, co pozwoliło poczynić znaczne oszczędności i lepiej dostosować kod C, który jest pod spodem.

Dlaczego format wyprzedził konkatenacje? Cóż, domyślam się. Wydaje mi się, iż chodzi o ewaluację. Być może Python, z racji tego, że stringi są niemutowalne w Pythonie, za każdym razem, kiedy wykonywał operacje dodania na dwóch stringach, musiał zaalokować nowy kawałek pamięci, który pomieści X znaków, gdzie X to suma długości dwóch stringów, potem je tam przekopiować, by otrzymać finalną wartość. Z racji doświadczenia tego, jak python działa, to założę się, że w naszym wypadku, kiedy mieliśmy kod w postacie a + ' ' + b + ..., Python wykonywał każdą operacje dodawania oddzielnie. To znaczy, prawdopodobnie instrukcje pod spodem wyglądały tak:

1. Zaalokuj pamięć, która pomieści zmienną a oraz string ' '.
2. Przekopiuj wartość a
3. Przekopiuj wartość ' '
4. Otrzymany wynik dodaj do zmiennej b.
5. Zaalokuj pamięć, która pomieści poprzedni wynik oraz zmienną b.
6. ...

I tak dalej. A to wszystko kosztuje czas - nowe alokacje, kopiowanie. Tak mi się przynajmniej wydaje, że to zadziałało w ten sposób, nie jestem jednak pewien, czy developerzy pythona nie poczynili jakiś optymalizacji na ten przypadek i może robią to inaczej? Nie wiem, aż tak głęboko nie zaglądałem, ale patrząc po wynikach, nie sądzę.

9.2.4.5 Podsumowanie - o wydajności W pythonie mechanizmy, które zdają się wyglądać elegancko w danej sytuacji, zazwyczaj są pod takową zoptymalizowane i przygotowane, stąd warto ich używać. Piękny ten wąż po prostu. Elegancki kod.

Używajcie zatem f-stringów gdziekolwiek tylko możecie i cieszcie się z życia, tam gdzie dużo stringów do połączenia w przewidywalny sposób, join. Dzięki temu wasz kod będzie ładniejszy ale i szybszy!

9.2.5 Przykłady podstawowych operacji na stringach

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__getnewargs__', '__getstate__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__',
 '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier',
 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
 'partition', 'removeprefix', 'removesuffix', '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'
```

Zauważ, że stringi są niemutowalne. Znaczy, to, że nie możemy ich modyfikować a każda modyfikacja powoduje, że w efekcie powstaje nowy string. Pomówimy o mutowalności jeszcze nieco później kiedy będziemy omawiać listy i krotki.

9.3 Bajty

9.3.1 Krótka charakterystyka

Hmmm, bajty. To owszem, podstawowy i prosty typ w Pythonie, to po prostu ciąg bajtów jak sama nazwa wskazuje. Przydatne przy operacjach na plikach. Poczytaj samodzielnie.

9.4 Typ logiczny/Boolowski

9.4.1 Krótka charakterystyka

Tutaj sprawa wygląda prosto - typ logiczny to tak zwany bool - albo prawda, albo fałsz. Typ o długości 1 bita. Albo 0 albo 1, albo `True` albo `False`.

9.4.2 Wartości prawdziwe vs wartości fałszywe

Zazwyczaj używany przy warunkach, ustawianiu jakichś flag i tak dalej. Warto zaznaczyć, że w Pythonie typ logiczny jest nieco rozszerzony. To znaczy, że cokolwiek co można zewaluować do pewnych rzeczy, będzie traktowane jak typ logiczny. Mówiąc prościej, Fałsz jest zerem lub czymś pustym. Prawda jest zaś dowolną liczbą inną niż zero lub czymś niepustym. Pusty string to Fałsz, jakiś tekst to Prawda. Pusta lista to Fałsz. Niepusta przeciwnie.

```
>>> bool(1)
True
>>> bool(0)
False
>>> bool(0.1)
True
>>> bool([])
False
>>> bool(["f"])
True
>>> bool("String")
True
>>> bool("")
False
```

Mała notka. Funkcja `bool` to w Pythonie coś, co próbuje 'przekonwertować' zadaną wartość do `booleana` czyli typu logicznego/boolowskiego. Zasady Przedstawiłem ci wyżej. Inne typy również mają swoje odpowiedniki, pobaw się nimi.

9.4.3 Przykłady podstawowych operacji na typie boolowskim

9.5 Listy

9.5.1 Krótka charakterystyka

Czym jest lista? Lista to nic innego jak mutowalny zbiór elementów. Swego rodzaju 'tablica'. Możesz sobie pomyśleć o tym w kategoriach stringa. Dlaczego? A bo czymże jest string, aniżeli listą znaków? To po prostu pewien obszar w pamięci, który jest mutowalny. Co to znaczy? Otóż oznacza to, że lista jest zbiorem elementów, który po zadeklarowaniu można dowolnie modyfikować. Listę można zmniejszać, kasować z niej elementy. Można dodawać do niej elementy. Można znowu je usunąć. Rozmiar listy można zmienić. Poszczególne elementy można nadpisać. Co chcesz, to masz.

Lista może składać się z praktycznie dowolnych elementów, oznacza to, że do listy można włożyć prawie każdy obiekt z Pythona. Nie zawsze tak jest. W innych językach programowania często jest tak, że jak definiujemy tablicę, to po pierwsze z góry znana jest jej długość, chyba, że używamy tablic dynamicznych, dwa, jej elementy często są ograniczone do jednego tylko typu z różnych powodów. Dlaczego?

9.5.2 Lista od strony niskopoziomowej

Jednym z powodów będzie coś, co nawiązuje do samego początku tej książki, gdzie opisywaliśmy sobie zagadnienie związane z tym, kiedy program wie, gdzie przestać czytać pamięć w przypadku odczytu zmiennej spod danego adresu.

Otóż tutaj zastosowanie ma podobna analogia. Np. w C deklarując tablicę/listę, podajesz jej długość i typ. Po co? Po to by kompilator/program mógł sobie ogarnąć ile pamięci zaalokować i jak poradzić sobie z adresami, gdzie przestać czytać etc.

Zatem 10 elementowa tablica charów zaalokuje nam w C pamięć o rozmiarze `10 * size(char)`. Komputer będzie wiedział gdzie, i co, i jak czytać, kiedy skończyć.

Załóżmy, że nasza tablica znajduje się pod adresem `0x1` i w jej środku znajdują się 4 elementy, każdy z nich o wielkości jednego bajta:

Bit (hex)	0x01	0x02	0x03	0x04
Wartość	A	B	C	D

Tutaj przypomnij sobie jak wygląda pamięć, ile bitów używamy standardowo do zapisu pojedynczego znaku, załóżmy ASCII w tym wypadku i jak liczyć z szesnastkowego na dziesiętny/binarny.

Mamy już zobrazowane nieco jak to wygląda w przypadku np. C i tablicy/listy o jednakowym typie i znanej długości. Co w przypadku Pythona?

9.5.3 Referencje i wartości

W Pythonie jest nieco inaczej, ale podobnie. Otóż można by się zapytać: to skąd Pythonie wie kiedy przestać czytać dany adres, skoro tam pod spodem to też najczęściej jest C, w przypadku CPythona przynajmniej? Otóż lista w pythonie tak naprawdę nie jest listą wartości z danymi typami a listą referencji niejako. Cóż to takie? Otóż Python tak naprawdę, kiedy tworzymy listę, to przechowuje on sobie zbiór referencji do danych wartości, a nie same wartości. Zatem wracając do naszej wcześniejszej analogii i porównania z C, gdzie musieliśmy zadeklarować typ wartości w tablicy, nagle wszystko ma sens. Okazuje się, że w Pythonie, tam pod spodem, też mamy w pewnym sensie jeden rodzaj wartości - referencje. Referencje to, w uproszczeniu, odniesienia do jakichś obiektów. Obiekt może być np. inną listą czy instancją jakiejś klasy.

Czyli w Pythonie, w pamięci nasza tablica będzie wyglądała +/- w taki sposób:

Bit (hex)	0x01	0x09	0x11	0x19
Wartość	Referencja1	Referencja3	Referencja2	Referencja4

Kolejność losowa. Teraz kiedy odczytujemy sobie wartość z listy python przejdzie sobie do tego co skrywa np. `Referencja1` i weźmie sobie co potrzebuje, niezależnie od typu. Takim oto sposobem pozornie różne typy mogą być w pythonie w jednej liście i nie powoduje to problemów.

Jedna zawiłość wyjaśniona. Pora na drugą -> dynamiczny rozmiar.

9.5.4 Dynamiczny rozmiar list

Okej, pora wyjaśnić jak to się magicznie dzieje, że w listy w Pythonie mają dynamiczny rozmiar, mimo tego, że np. w C, póki co, pisałem o tym, że wypada podać rozmiar. Otóż sprawa ma się tak, że w Pythonie, pod spodem, jest tak samo. Interpreter Pythona patrzy na to z iloma elementami utworzyliśmy listę i następnie sam się 'domyśla' ile elementową tablicę zaalokować.

Co się dzieje gdy zmieniamy rozmiar listy/dodajemy nowe elementy?

9.5.5 Alokowanie ponad potrzeby - chciwy i przebiegły wąż

Generalnie przypadek tutaj mamy taki, że Python podczas deklaracji jakiejś listy alokuje więcej pamięci niż nam potrzeba! CZO?! JAK TO?

Ano tak. Generalnie zazwyczaj jest to około $2n$ zakładając, że $2n >= 2$ gdzie n to liczba elementów. Zatem nawet kiedy inicjalizujemy pustą listę Python pod spodem alokuje sobie miejsce na przynajmniej dwa elementy, albo więcej. Nie pamiętam już nawet. Po co? A no po to, że interpreter spodziewa się tego, iż będziemy dodawać kolejne elementy. Jeśli osiągniemy zadaną wielkość i spróbujemy dodać kolejny element nastąpi wtedy alokacja nowej pamięci, znowu zachodzi proces opisany przed chwilą i nadalokacja pamięci, po czym wartości ze starego miejsca w pamięci/starej listy są kopiowane w nowe miejsce gdzie mamy zaalokowany większy fragment pamięci, a całość jest dynamicznie podmieniana gdzie trzeba w taki sposób, że my, jako programista/użytkownik końcowy, nawet nie zauważymy.

Jak widzisz Python robi za nas wiele rzeczy byśmy nie musieli się tym martwić. Oczywiście, ma to swój koszt w postaci wydajności czy to komputacyjnej czy pamięciowej, ale niestety, coś za coś. Python szybki nie jest, ale jest wystarczająco szybki, ale tę rozprawkę już czytałeś - w rozdziale o wadach i zaletach Pythona.

Opisuję tutaj cały proces w bardzo dużym uproszczeniu, ale plus minus tak to wygląda.

9.5.6 Dostęp do elementów listy

Dostęp do elementów listy zachodzi za pomocą podania indeksu elementu jaki chcemy dosięgnąć. Z racji tego, że znamy rozmiar referencji to wiemy o ile zrobić offset w pamięci. Mówiliśmy już o tym. W kodzie wygląda to tak:

```
war_crimes = ["Russia on Ukraine", "Israel on Palestine"]
>>> war_crimes[0]
'Russia on Ukraine'
>>> war_crimes[1]
'Israel on Palestine'
>>> war_crimes[2]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Dosyć logiczne. Przekroczenie długości tablicy spowoduje błąd. Jedna ważna notatka. W programowaniu zazwyczaj indeksujemy od zera. Dlaczego? Podstawy odpowiedzi znajdziesz w rozdziale 4. i 5., ale dla pewności wytłumaczę. Oczywiście są wyjątki od reguły, niektóre języki indeksują od 1, bo tak i cześć, ale my o prawilnych językach rozmawiamy jak C, Python czy Rust.

9.5.7 Negatywne indeksy

O ile zazwyczaj w innych językach niekoniecznie istnieje taka instytucja jak negatywne indeksy w przypadku list, tak w przypadku Pythona jak najbardziej tak. Używanie negatywnych indeksów powoduje, że Python zaczyna liczyć od końca listy w górę. Dość prosty koncept.

```
>>> numbers = [1,2,3,4]
>>> numbers[-1]
4
>>> numbers[-2]
3
```

9.5.8 Cięcie list

Czyli tak zwany slicing. O cóż chodzi? A no mając sobie jakąś listę może najść nas potrzeba, by operować na jej kawałku czy coś. Kto wie, dziwne potrzeby ludzie mają. No i cóż teraz? Kopiować, iterować samemu po danej liście? Absolutnie nie. Python posiada, jak zawsze, wbudowane rozwiązanie, które pozwala nam w prosty i przyjemny sposób osiągnąć tenże efekt. Slicing się to po angielskiemu nazywa. Jak działa?

```
>>> numbers = [1,2,3,4, 5, 6, 7, 8]
>>> numbers[3:6]
[4, 5, 6]
# od pierwszego do piątego
>>> numbers[:4]
[1, 2, 3, 4]
# od czwartego do końca
>>> numbers[3:]
[4, 5, 6, 7, 8]
# od pierwszego do ostatniego
>>> numbers[::]
[1, 2, 3, 4, 5, 6, 7, 8]
# od pierwszego elementu listy, zaznaczając co drugi element -> 2n+1
>>> numbers[::2]
[1, 3, 5, 7]
# od drugiego elementu listy, zaznaczając potem co drugi element -> 2n
```

```
>>> numbers[1::2]
[2, 4, 6, 8]
 # od ostatniego elementu listy do końca zaznaczając po tem co drugi element
>>> numbers[-1::2]
[8]
 # od ostatniego elementu listy do końca czyli po prostu ostatni element
>>> numbers[-1:]
[8]
>>> numbers[-4:]
[5, 6, 7, 8]
>>> numbers[:-4]
[1, 2, 3, 4]
>>> numbers[::-1]  # łatwy sposób odwrócenia listy
[8, 7, 6, 5, 4, 3, 2, 1]
```

Slicing ma następującą składnię

```
zadana_lista[początkowy_element:końcowy_element:skok]
```

Początkowy czy końcowy element to raczej znany koncept. Skok to po prostu informacja, co który element mamy zabierać.

Ważna informacja: slicing tworzy kopie listy, świeżutką, nowiutką kopię listy. Jest to jednak płytka kopia, a nie głęboka. Co to znaczy, opiszę nieco dalej. Dowód:

```
>>> numbers is numbers[:]
False
```

Jak widać slicing powoduje skopiowanie i utworzenie nowej listy pod spodem. Albo inaczej. Pamięć zostanie skopiowana w drugie miejsce i zduplikowana w ten sposób, natomiast referencje w środku będę te same. Jeśli lista zawiera referencje do obiektu, który jest pass-by-reference, to modyfikacja tego obiektu spowoduje zmianę wartości w obu listach, tej skopiowanej też, natomiast jeśli coś jest pass-by-value (niemutowalne typy w większości przypadków), to modyfikacja będzie dotyczyła tylko tej listy której element modyfikowaliśmy.

```
>>> 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  # dwie różne listy
False
# pierwszym elementem listy jest inna lista
# listy są mutowalne i przekazujemy je przez referencje
>>> pass_by_reference[0]
[1, 2, 3]
```

```
# modyfikujemy tutaj drugi element tej wewnętrznej listy z oryginału
>>> pass_by_reference[0][1] = "test"
>>> pass_by_reference[0]
[1, 'test', 3]
# jakimś sposobem lista w kopii też uległa zmianie
>>> new_pass_by_reference
[[1, 'test', 3], 1, 2, 3]
>>> new_pass_by_reference[0]
[1, 'test', 3]
```

Pomyśl o tym. Pogadamy pewnie jeszcze na ten temat przy omawianiu kluczy w słownikach.

9.5.9 Dlaczego indeksujemy od zera

Stwierdziłem kiedyś gdzieś, że indeksowanie od zera jest logiczne i ma swoje powody, to naturalne i jest tak, jak w RHC przykazali. Padł jednak komentarz, że jest w zasadzie na odwrót a my, programiści, indeksujemy sobie od 0 tak po prostu, bo się przyzwyczailiśmy.

Otóż nie. Mimo tego, że sam kiedyś podobnie myślałem, to indeksowanie od 0 jest logiczne i ma swoją zasadność. Jaką?

```
#include <stdio.h>

int main(void) {
    int numbers[] = {1,2,3,4};
    printf("numbers in general: %p -- %p\n", &numbers, numbers+0);
    for (int i = 0; i < sizeof(numbers)/sizeof(numbers[0]); i++) {
        printf("number no. %i: %p -- %p -- value: %d\n",
               i, &numbers[i], numbers+i, *(numbers+i));
    }
    printf("int size: %d\n", sizeof(int));
    return 0;
}
```

Kod wyżej skompilowany i uruchomiony da nam:

```
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
```

Przeanalizujmy troszkę o cóż tu chodzi.

Zanim to zrobimy, zaznaczę tylko, że Ty, jeśli uruchomiłeś ten kod u siebie, mogłeś dostać nieco inne wyniki. To normalne.

Dla większości osób nieznajomych z C/C++ ten kod może wydawać się nieco kryptyczny, ale w gruncie rzeczy jest dość prosty.

Zacznijmy może od linijki

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

Zakładam, że pierwsza część printa jest zrozumiała dla każdego, może poza `%p` - to po prostu nam mówi, że argument do wyprintowania będzie specyficznym typem danych.

A co to to całe `&numbers` - operator & mówi, że chcę otrzymać adres danej zmiennej - czyli jej lokację w pamięci. Bo jak dobrze wiemy, zmienne alokowane są w pamięci, w pewnym miejscu wybranym przez komputer. Ponownie - mówiliśmy już o tym w rozdziałach 4. i 5.

To miejsce zazwyczaj opisuje się jako 'adres' - czyli liczba bajtów/bitów od 'początku' pamięci, którą procesor musi 'przeskoczyć', by dotrzeć do danej zmiennej.

Nasza tablica (czyli taka jakby lista z Pythona, ale nie do końca), znajduje się pod adresem: 0x7ffc9f728f20 (zapis szesnastkowy), i jest to tym samym adres naszego pierwszego elementu.

Kompilator musi musi jednak wiedzieć, pod jakim adresem znajduje się następny element naszej tablicy. Skąd? To już wyjaśniałem i liczę na twoją odpowiedź. Moją znajdziesz poniżej.

Zadeklarowaliśmy, że elementy naszej tablicy będą typu `int`. Typ `int` na komputerze, z którego korzystam, jest akurat 4 bajtowy, czyli 32 bitowy. Jest to w zasadzie standard (chociaż oficjalnie standard mówi o tym, że int ma być po prostu przynajmniej 16 bitowy, nie specyfikuje jego rozmiaru dokładnie), ale czasami są odstępstwa od reguły, zależnie od architektury, stąd też ten `sizeof(int)` w kodzie - zwraca on rozmiar danego typu w obecnym środowisku.

Dlatego też, jeśli 0x7ffc9f728f20 jest adresem pierwszego elementu, który zajmuje w pamięci 4 bajty o adresach:

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

to możemy wnioskować, że następny element tej tablicy będzie po nim, pod adresem `0x7ffc9f728f24`, czyli 4 bajty dalej. Następny znowu kolejne 4 bajty i tak dalej, aż do ostatniego elementu. Można stąd wyekstrahować prosty wzór zatem.

Adres konkretnego elementu tablicy można określić wzorem:

$adres\_pierwszego\_elementu + (index * rozmiar\_typu)$.

Z takiego też wzoru korzysta komputer - za każdym razem, gdy piszesz `numbers[index]` *kompilator tłumaczy to sobie wewnętrznie na*

$(numbers + (index * rozmiar\_typu))$.

Co znaczy dla kompilatora `*`? Nic innego, jak 'idź pod dany adres i weź wartość znajdującą się pod tym adresem.'

Zatem gdy napiszemy numbers[0], to nasz kompilator przetłumaczy to na

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

co z kolei znaczy: weź wartość z tego adresu i wstaw ją tutaj.

W przypadku np. numbers[1], będzie to

$0x7ffc9f728f20 + (1 * sizeof(int)) = 0x7ffc9f728f20 + (1 * 4)$
$0x7ffc9f728f20 + (1 * 4) = 0x7ffc9f728f20 + 4$
czyli
`0x7ffc9f728f24`.

Jasne? Jak dla mnie tak. Jeśli masz problem ze zrozumieniem tego konceptu, nie przejmuj się, wiele osób nie do końca rozumie wskaźniki, adres i pamięć. Ja też miałem z tym problem. Przynajmniej na początku.

Możesz wspomóc się filmikami Gynvaela - Gynvael's Code: Pointery #1 czy wykładami z CS50 - kursu z Harvardu oni, jako osoby o znacznie większej wiedzy, tłumaczą całe zagadnienie znacznie lepiej niż ja.

9.5.10 Jakby to wyglądało, gdybyśmy indeksowali od 1?

Załóżmy, że indeksujemy od 1. Wtedy wzór musiałby ulec modyfikacji - i wyglądałby on tak:

$adres\_pierwszego\_elementu + ((index - 1) * rozmiar\_typu)$

Innym rozwiązaniem byłoby przesunięcie lokacji pierwszego elementu tablicy o 4 bajty do przodu względem adresu samej tablicy, ale wtedy nasza tablica zajmowała by dodatkowe miejsce w pamięci i to niepotrzebnie, gdyż te pierwsze x bajtów, gdzie x to rozmiar danego typu danych, byłoby po prostu puste. To raz, dwa, że trzeba by pamiętać, że adres tablicy nie jest adresem pierwszego jej elementu.

Oba te rozwiązania są bezsensowne, bo o ile nie jest to niby dużo - kilka bajtów na każdej tablicy, czy jedna operacja odejmowania, to gdy pomnożymy to sobie przez ilość takich zmiennych, które mamy w pamięci, to już wyjdzie całkiem pokaźna sumka bajtów/operacji, które w istocie rzeczy, są zbędnie zajmowane.

Dodatkowo ileż kodu bazuje już na indeksowaniu od 0. Niemożliwym by było to wszystko zmienić.

Oczywiście, są również inne argumenty, by indeksować czy liczyć elementy od zera, jak chociażby te, głoszone przez Dijsktrę - Why numbering should start at zero]. To taki raczej znany i ważny Pan, dla tych, którzy nie kojarzą ;)

Ogółem użyłem tutaj trochę uproszczeń i skrótów myślowych, ale generalny koncept przekazany.

9.5.11 Przykłady podstawowych operacji na listach

9.6 Krotki/Tuple

9.6.1 Krótka charakterystyka

Krotka to tak naprawdę nic innego niż niemutowalna lista. Co to znaczy? Mniej więcej tyle, że po zadeklarowaniu krotki nie można już modyfikować. Deklarujemy raz i koniec. Niesie to ze sobą wiele konsekwencji o których zaraz opowiem.

Jedyne co z krotką można zrobić to odczytać ją, skopiować czy też ponownie zadeklarować zmienną o danej nazwie. Przykłady niżej.

Po co nam te krotki/tuple? Ogółem niemutowalnośc danych często sprawia, że z pewnymi rzeczami trudniej sobie strzelić w stopę. Do tego jest to podejście bardziej podobne do programowania funkcyjnego, powiedzmy. Niemutowalność jest całkiem spoko.

Oprócz tego mamy tutaj jeszcze jedną zaletę. Wydajność.

9.6.2 Wydajna bestia

Otóż jeśli jest to niemutowalna struktura danych to interpreter pythona wie dokładnie ile pamięci zaalokować plus z pewnych względów proces ten zachodzi szybciej. Czyli tutaj alokacja ponad potrzeby nie ma miejsca plus do tego instrukcja wykonuje się niejako szybciej, python wie jakie typy są użyte, zna konkretne dane które wykorzystaliśmy etc.

Jako anegdotkę przytoczę historię, gdy zastosowanie tupli zmniejszyło nam zużycie pamięci z 4 GB do ~2.1 GB w pewnej niedużej webappce. W innych wypadkach redukcja bywała nawet bardziej drastyczna.

9.6.3 Przykłady podstawowych operacji na krotkach

```
dir(tuple)
['__add__', '__class__', '__class_getitem__', '__contains__',
 '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
 '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__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
```

9.7 Słowniki

9.7.1 Krótka charakterystyka

Czymże jest magicznie brzmiący słownik zwany także Dictem/HashMapą? Otóż jest to nic innego jak swego rodzaju słownik/mapowanie. Tak jak w zwykłym słowniku mamy pewnego rodzaju mapowanie **kluczy** do **wartości** w postaci słowa i jego znaczenia, tak podobnie jest w pythonowym słowniku/hashmapie.

Do rzeczy.

Słownik w Pythonie to struktura danych, która pozwala nam przechowywać dowolne wartości pod określonymi kluczami. Wyobraź sobie listę, ale zamiast indeksu w postaci liczbowej masz indeks w postaci określonego klucza.

W praktyce wygląda to tak:

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

Przewidywalne. Reszta działa również podobnie jak w liście.

Co jest różnego od działania listy jest to, że o ile w liście zapewniona jest gwarancja tego, że elementy zawsze będą w tej kolejności w jakiej je do listy włożyliśmy. Tak hashmapa z definicji takiego czegoś nie przewiduje. Obecna implementacja CPythona, od wersji 3.7, mimo wszystko zapewnia coś takiego dodatkowo, czyli ze zwykłego `Dicta` zrobił nam się `OrderedDict`, natomiast lepiej się nie nastawiać na to, gdyż wersja pythona typu 3.6 jest dość stara, ale wciąż jest mnóstwo projektów w niej napisanych. Co z tego? A no to, że kod, który będziesz pisał prawdopodobnie może być odpalany na wersji Pythona, która nie bierze pod uwagę i nie gwarantuje zachowania kolejności insercji elementów, zatem lepiej na tym zbytnio nie polegać, bo w większości przypadków ta kolejność będzie zachowana mimo wszystko, ale nie jest ona gwarantowana implementacyjnie, czyli zawsze znajdzie się ten 1%, gdzie jednak coś pójdzie nie tak. Potem weź takiego buga dostań do inwestygacji.

Oczywiście jeśli jesteś świadom i wiesz co robisz, plus masz gwarancje tego, na jakich wersjach pythona twój kod będzie banglał, to śmiało. Natomiast pamiętaj, W najnowszej wersji pythona -> spoko, poniżej 3.8 albo 3.7 już niekoniecznie. Sprawdź dokładnie w której wersji wprowadzono `OrderedDict` jako domyślny.

9.7.2 Jak przebiega proces dodawania elementów do dicta?

Otóż generalnie tak jak w przypadku listy mieliśmy numeryczny indeks, za pomocą którego Python liczył sobie offset w pamięci, tak w przypadku dicta mamy coś takiego jak funkcja hashująca. Ta funkcja bierze sobie za argument klucz jakiego używamy i na jego podstawie próbuje generować w miarę unikalny hash. Potem na podstawie hashu, zazwyczaj poprzez operacje modolu, ogarniamy sobie adres/offset gdzie trzymana jest dana wartość.

Logiczne? Czyli tak, za każdym razem jak ktoś wpisuje `some_dict["key"]` to pod spodem dzieje się coś takiego, że interpreter Pythona, by uzyskać adres z jakiego odczytać ma wartość dla danego klucza, bierze tenże klucz, wrzuca go w funkcję hashującą, nie wiem, załóżmy `hash("key")`, ta funkcja zaś zwraca nam jakiś tam możliwie unikalny hash wygenerowany/obliczony na podstawie danego klucza. Z hasha wyczarowujemy sobie adres/offset. Jakoś tak.

Dlaczego w miarę unikalny?

9.7.3 Kolizja hashy

Hash collision to coś co się czasami zdarza. Dlaczego? Otóż funkcja hashująca nie może być kompletnie losowa. Musi być stabilna i powtarzalna. To znaczy, dla zadanego argumentu musi zawsze zwracać to samo, generacja hashu musi odbywać się w sposób przewidywalny.

Dlaczego? Otóż gdyby było inaczej, a dla jednego klucza dałoby się wygenerować kilka hashy, to powstałby problem w postaci takiej, że nigdy nie moglibyśmy, albo czasem byśmy nie mogli, trafić do dokładnego adresu, gdzie pierwotnie przypisaliśmy wartość. Co to oznacza?

Brak kompletnej losowości sprawia to, że algorytmy hashujące są w jakimś tam stopniu ograniczone. Ograniczone są też wydajnością i czasem jaki komputer może poświęcić na hashowanie, które dzieje się dość często jednak, bez kosztów dla użytkownika. Trzeba było zatem znaleźć kompromis pomiędzy skomplikowaniem funkcji hashującej i jej zasobożernością, czasem wykonywania a unikalnością dostarczanych hashy dla różnych kluczy.

Obecnie mądre głowy jakiś złoty środek wymyśliły, natomiast w dzisiejszych czasach zdarza się operować na zbiorach danych tak dużych, że kolizja hashu się zdarza i funkcja hashująca wygeneruje taki sam hash dla dwóch różnych kluczy, przez co jeden klucz nadpisze drugi. Bardzo, bardzo rzadki przypadek. Natomiast jeśli masz do przetworzenia milion trylionów rekordów, to nagle bardzo rzadkie przypadki mają jakieś 100% szansy się pojawić.

9.7.4 Co może być kluczem?

Kluczem w słowniku może być dowolna wartość/zmienna/obiekt, który jest hashowalny. Co to znaczy? A no to, że w swojej definicji zawiera implementację metody `__hash__`. Tak w dużym skrócie. Czyli te obiekty, dla których zaimplementowana została funkcja hashująca, mogą być kluczami. W sumie logiczne, bo jeśli nie mamy metody hashującej dany obiekt to nie policzymy hashu. Nie policzymy hashu z klucza to nie ogarniemy adresu/offsetu. Bez tego niewiadomo gdzie w pamięci przechować wartość. A zatem musimy mieć tę metodę zaimplementowaną. Logiczne.

9.7.5 Pass by value & Pass by reference

O co tutaj chodzi? O przekazywanie zawartości poprzez referencje lub poprzez wartość. Dokładniej mówiąc chodzi o to, że niektóre obiekty Python skopiuje, tak jak ma to miejsce w przypadku slicingu listy i otrzymania jej kopii w sposób, który wewnętrzne elementy tegoż obiektu uwspólni dla kopii jak i dla oryginału. Troszkę rozpisywałem się już o tym podczas pisania o listach.

Wydaje mi się, że zostało to tam w miarę jasno wytłumaczone. Teraz tak - dlaczego wspominam o tym w kontekście dictów? Otóż mutowalne typy danych często przekazywane są przez referencje, czyli zamiast samego obiektu, dostajemy referencję doń. Z tego też powodu np. lista nie może być kluczem w słowniku - jest ona mutowalna, przekazywana przez referencję a z tego też powodu nie implementuje metody `__hash__` przez co jest niehashowalna, a zatem implementacja słownika w pythonie, przy próbie ustanowienia nowego klucza będącego listą, wyrzuci błąd.

Przypomnijmy sobie kod, którym ilustrowałem przekazywanie przez referencje vs przez wartość.

```
>>> 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  # dwie różne listy
False
# pierwszym elementem listy jest inna lista
# listy są mutowalne i przekazujemy je przez referencje
>>> pass_by_reference[0]
[1, 2, 3]
# modyfikujemy tutaj drugi element tej wewnętrznej listy z oryginału
>>> pass_by_reference[0][1] = "test"
>>> pass_by_reference[0]
[1, 'test', 3]
# jakimś sposobem lista w kopii też uległa zmianie
>>> new_pass_by_reference
[[1, 'test', 3], 1, 2, 3]
>>> new_pass_by_reference[0]
[1, 'test', 3]
```

Warto na to uważać, zwłaszcza przy np. wybieraniu argumentów dla funkcji czy ustanawianiu wartości domyślnych. Pass by reference -> zwracamy adres i tam następuje modyfikacja danego obiektu. Pass by value -> zwracamy samą wartość i następuje jej 'skopiowanie' na świeżo zamiast modyfikować zmienną pod danym adresem, otrzymujemy nową. Kolejny przykład:

```
student = {"Putler": 10}
def test(student):
    new = {"Wołodia": 20, "CzłowiekMałpa": 21}
    student.update(new)
    print("Wewnątrz funkcji", student)
test(student)
print("Poza funkcją:", student)
```

Jak widać wyżej dict przekazywany jest poprzez referencje, czyli adres tak jakby. Python zatem idzie pod dany adres i modyfikuje obiekt przez co zmiany są rozpowszechnione w miejscach, gdzie niewprawiony programista mógłby się nie spodziewać. W przypadku przekazania wartości sprawa ma się inaczej. Oryginalny obiekt nie jest modyfikowany, jedynie jego kopia.

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

```
('Wewnątrz: ', 'NAFO <3')
>>> student_name
'NAFO'
```

Nie wiem, czy ma to sens, może wrócimy do tego jeszcze. Przeanalizuj i poszukaj troszeczkę w necie samodzielnie na ten temat by dodatkowo rozjaśnić sytuację.

9.7.6 Kopia płytka i kopia głęboka a klucze w słowniku

Mamy te całe przekazywanie referencji, wartości etc. Pomówmy teraz zatem o kopiach płytkich i głębokich. Krótko bo krótko, ale warto wspomnieć.

Kiedy użyliśmy slicingu jako metody kopiowania listy, otrzymaliśmy tak zwaną płytką kopię tejże listy. Co to znaczy płytką? Mianowicie skopiowany został tylko początkowy obiekt, obiekt z samej góry. Wszystko co w środku a co było przekazywane przez referencje, nie zostało zduplikowane. Skopiowane zostały jedynie referencje. To jest płytka kopia.

Głęboka kopia to kopia, gdzie interpreter 'wchodzi' do obiektu, który kopiujemy i kopiuje wszystko przez wartość, nie przez referencje. Sprawia to, że otrzymujemy faktyczny, samodzielny i niezależny duplikat danego obiektu a nie tylko jego 'top levelu' jak w przypadku kopii płytkiej.

Czasami potrzebne. Warto wiedzieć, gdyż w niektórych przypadkach myślimy, że mamy dwa różne obiekty po skopiowaniu używając kopii płytkiej, modyfikujemy jeden obiekt a tu bam, zmiany w obu. Potrafi to spowodować naprawdę brzydkie do debugowania błędy. Nie polecam.

9.7.7 dict.values() keys() items()

Zwróć uwagę na te trzy metody. Pobaw się nimi i podsumuj swoje wnioski. Ja zwrócę tylko uwagę na jedną rzecz.

Jakiego typu obiekty zwracają funkcje dict.keys(), dict.values(), dict.items()? dict.items() - oczywiste. Lista tupli. Ale tak nie do końca. Bo jak się bliżej przyjrzeć to jest to klasa dict_items, która nie do końca jest listą – trochę to taka rozszerzona klasa, bo umożliwia nam wykonywanie na niej operacji takich, jak na setach. Podobnie z keys() i values(). Czyli na obiektach zwracanych przez te funkcje można wykonywać operacje sumy, różnicy czy części wspólnej ze zbiorów. Tldr – te funkcje zwracają iterable set-like object.

9.7.8 Przykłady podstawowych operacji na słownikach

```
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 to set-like obiekt, na którym
można wykonywać takie operacje jak na zbiorach"""
In [19]: some_dict.values()
Out[19]: dict_values(['OK', 'NOT OK'])
# dict_keys podobnie
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')])
```

9.8 Zbiory

9.8.1 Krótka charakterystyka

Czym są zbiory? Analogicznie jak w matematyce. To taka jakby lista, ale bez powtórzeń. Przynajmniej pozornie. Pod spodem jest nieco inaczej, bo pod spodem zbiorom/setom bliżej do hash mapy. W sumie to jest niejako hashmapa. Po co na co i dlaczego? Otóż zadajmy sobie pytanie, jakie są atrybuty zbiorów. Każdy element występuje tylko raz. Niekoniecznie zachowana kolejność insercji. Zaczyna brzmieć znajomo? Yup. Sety to tak jakby hashmapy gdzie wartości są też i kluczami niejako.

Jaka jest zaleta zbioru? Pierwsze to deduplikacja elementów -> każdy występuje dokładnie raz. Możemy wyciągnąć 'statystyki' z danego elementu, ile razy został dodany do seta, ale w samym secie pojawi się on tylko raz. Druga jest wydajnościowa.

9.8.2 Przeszukanie szybsze niż na warszawskiej Woli

Wyszukiwanie/przeszukiwanie w zbiorze ma złożoność obliczeniową na poziomie O(1) - czas stały. Co to znaczy? Niezależnie od rozmiaru zbioru by sprawdzić przynależność danego elementu do zbioru wykonujemy operację, która cechuje się stałym czasem wykonania niezależnym od rozmiaru. Czyli nawet dla bardzo bardzo dużych zbiorów, jeśli zmieszczą się w pamięci, mega szybko możemy stwierdzić, czy znajdują się one w danym zbiorze.

W przypadku listy nie ma aż tak łatwo, zwłaszcza jeśli dane są nieposortowane.

Dlaczego tak jest? Otóż przez to, że pod spodem jest niejako hashmapa, by sprawdzić czy element przynależy do zbioru wystarczy jedynie policzyć hash tegoż elementu a potem sprawdzić czy wszystko się zgadza. Stąd O(1) niezależnie od wielkości zbioru.

To zaś wymusza nam ograniczenia odnośnie tego, co możemy do zbioru wrzucić. Jakie? Takie same jak przy kluczach w słownikach.

Do tego zbiór pythonowy obsługuje też podobne operacje jak zbiór matematyczny. Koniunkcje, alternatywę, różnicę. Zwykła lista nie wszystko z tego potrafi ogarnąć.

9.8.3 Przykłady podstawowych operacji na zbiorach

```
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}
```

9.9 Podsumowanie

Uf, nareszcie. Kawał tekstu, co? A to tylko wybrane z Pythonowych typów.

Warto je dobrze znać, pobawić się z nimi i oswoić. Dlaczego? Otóż w standardowej bibliotece Pythona jest już tyle cudeniek, tyle różnych rzeczy, które ułatwiają życie, że głowa mała. Szkoda wymyślać koło na nowo i implementować coś samemu, skoro język udostępnia swoją wersję.

Mało tego, implementacja od zera jest często też bezsensowna z pewnego bardzo ważnego względu. Otóż twoja własna implementacja może być dziurawa, bo sprawdzasz ją ty, może ludzie na Code Review i tyle. Natomiast jeśli idzie o kod, który znajduje się w standardowej bibliotece Pythona i implementacje kontrybutorów, to sprawa ma się tak, że jest to kod przetestowany i przejrzany przez tysiące osób. Wpadki się zdarzają, to prawda. Natomiast gdzie prędzej znajdziemy buga? W kodzie, który przejrzało tysiące osób, który jest przetestowany w milionach produkcyjnych aplikacji jak i pokryty wieloma testami? Czy w kodzie, który przejrzałeś ty i być może twój zespół? Nie ma porównania.

Do tego dam sobie rękę uciąć, wierząc w to, że dzięki wysiłkowi tysięcy kontrybutorów kod z biblioteki standardowej Pythona będzie lepiej zoptymalizowany. Korzystaj z tego, co zbudowano i nie wynajduj koła na nowo tworząc własne naiwne implementacje algorytmu sortowania czy coś. Czasami zdarza się taka potrzeba, prawda, ale wątpię, byś ty takowe

miewał jako junior wannabe.

Dlatego koniecznością jest dobra znajomość biblioteki standardowej Pythona. Własne implementacje zostaw na cele związane z nauką czy zabawą, by zrozumieć jak coś działa. W kodzie produkcyjnym starajmy się tego unikać zaś na rzecz sprawdzonych rozwiązań z biblioteki standardowej.

To nie tylko ułatwia zadanie, ale sprawia, że kod będzie solidniejszy, bardziej zoptymalizowany i prawdopodobnie szybciej dostarczony. Łatwiej poskładać coś z gotowych klocków niż samemu budować dom zaczynając od wydobycia gliny i wypalenia cegieł.

9.10 Pytania i zadania

1. Czy w Pythonie jest dynamiczne typowanie? A może statyczne?
2. Co to oznacza? Jakie są tego wady, jakie zalety.
3. Wymień podstawowe typy danych w Pythonie.
4. A te bardziej złożone?
5. Jakie są różnice pomiędzy podstawowymi typami liczbowymi?
6. Co to znaczy, że wartość jest `truthy` albo `falsy` w Pythonie? Podaj przykłady z rozróżnieniem co jest czym.
7. Jakie są sposoby deklarowania tekstu w Pythonie?
8. Co to jest znak ucieczki i dlaczego go używamy?
9. Jakie funkcje ułatwiające przeliczanie znaków na liczby, na inne kodowania, o których wspomniałem?
10. Przygotuj artykuł w którym opiszesz o co chodzi z którą funkcją, po krótce zcharakteryzujesz każdy z podstawowych typów. Podasz przykłady dla których funkcje nie działają i domysły dlaczego. Docelową audiencją są inne osoby początkujące. Niech będzie to rozbudowany artykuł pełny twoich osobistych uwag a nie tylko Copi Pasta z dokumentacji. Podaj żywe przykłady.
11. Przygotuj podsumowującą listę dla znanych ci podstawowych typów danych w których przedstawisz i po krótce opiszesz jakie poszczególne typy mają metody/atrybuty i co dana metoda robi. Pomiń te, które zaczynają się od `__`. Jakiego polecenia musisz tu użyć?
12. To samo co w 11., ale dla typów złożonych. Bardzo dokładnie opisz. Artykuły możesz połączyć. Skup się na pokazaniu praktycznego użycia. Okomentuj kod i pokaż przykłady inne niż tutaj.
13. Artykuły z 12. i 11. podeślij na olafgorski@pm.me a ja je sprawdzę i dam feedback :)
14. Przygotuj tabelkę porównawczą: lista vs krotka vs zbiór.
15. Poeksperymentuj z naprawdę dużymi liczbami. Opisz swoje wnioski.
16. Stwórz kod, który pobierze liczbę od użytkownika a następnie powie, czy jest parzysta, czy nie. Jak pobrać coś od usera? Wygooluj jeśli nie pamiętasz.
17. Utwórz stringa a potem zamień jego ostatnią literę z pierwszą.

Pamiętaj, że Twoje odpowiedzi możesz wrzucić na GH o tutaj - https://github.com/grski/junior-python-exercises, a sprawdzę twoje rozwiązania i dam feedback. Więcej o tym podrozdziale ‘Część interaktywna’.

10 Pętle i iteracja

Opowiemy sobie trochę o pętlach w Pythonie, o iteracji też słów kilka. Co to, po co to i na co to komu. Porozmawiamy o różnych pętlach, `for`, czyli pętli krokowej, `while`, czyli zwykłej pętli. O iteracji i obiektach iterowalnych.

Co to jednak w ogóle znaczy?

Pętle to koncept w programowaniu służący wykonaniu jakieś czynności, jakiegoś kawałka kodu, zadaną ilość razy. Otóż wyobraź sobie, że musisz przetworzyć 10 elementów z tablicy. Na każdej z nich wykonać jakieś operacje. Załóżmy, że te operacje to nie jest jedna linijka a skomplikowane przetwarzanie. Chociaż nie, dla naszego przykładu nawet mnożenie weźmy. I co teraz? Naiwne rozwiązanie wyglądałoby tak:

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

Koszmar. Teraz pomyślcie, że tych elementów jest więcej. Na przykład milion, bo tylu użytkowników masz. Ręcznie tego nie idzie wyedytować, chyba, że zatrudnimy skończoną liczbę studentów albo Hindusów. Natomiast wciąż, potrwa to niezmiernie długo. No i tutaj właśnie wchodzą pętle całe na biało.

10.1 Pętla krokowa

10.1.1 Krótka charakterystyka

Zacznijmy od czegoś, co nazywa się pętla krokowa. Jest to pętla, która pozwala nam 'przejść się' po elementach zadanego obiektu i umożliwia, krok po kroku, czyli element po elemencie, przetwarzanie danego elementu. To znaczy, że słowo kluczowe `for` umożliwia nam iterację po elementach obiektu iterowalnego.

10.1.2 Obiekt iterowalny

Czymże jest obiekt iterowalny? To taki obiekt, który po wrzuceniu do funkcji `iter()` zwróci nam iterator. A iterator to coś, na czym wołamy `next()`. Widziałeś już kilka przykładów iterowalnych obiektów. Listy, Dicty, Tuple.

To takie obiekty, które mają zaimplementowaną metodę `__iter__` i kolejno `__next__`, czyli w skrócie programista powiedział pythonowi jak ma brać kolejne elementy z danego obiektu/danej struktury i która zwraca obiekt iteratora.

W listach, dictach, tuplach mamy to domyślnie jako część języka. Jeśli sami tworzymy jakieś wyspecjalizowane klasy, to również możemy uczynić je iterowalnymi poprzez zaimplementowanie tejże metody. Czyli tak w skrócie to takie coś po czym można iterować.

Iterować znaczy przechodzić krok po kroku po elementach danej struktury. Zrozumiałe? Oby tak. Jak nie to googlaj albo patrz kod niżej.

Jak to wygląda w praktyce? Zmodyfikujmy nasz przykład wyżej.

```
>>> 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]
```

Zamiast miliona linii mamy dwie. Fajne, co? Ale chwila, co to za magiczna funkcja `enumerate`? To wbudowana funkcja Pythona, która pozwala nam na ponumerowanie elementów danego obiektu. Mówiąc prościej to taki wrapper, taka nakładka na nasz dany obiekt, np. listę, która oprócz danego elementu z obiektu zwróci też jego indeks.

Całość, przekuta na instrukcję w języku polskim, wygląda tak.

1. Weź obiekt elements. Elements to lista.
2. Przekaż obiekt elements jako argument dla funkcji enumerate.
3. Funkcja enumerate zwraca nowy obiekt. Obiekt ten jest czymś z kolei, co zwraca nam kolejne elementy oryginalnego obiektu i dokleja do nich niejako indeks/numerek

Następnie kiedy mamy ten nowo zwrócony obiekt, pętla for wchodzi do życia i jedzie z tematem.

1. Weź nowo zwrócony obiekt
2. Pod spodem zawołaj `__iter__()`, który zwróci nam iterator dla tej listy, chyba, że domyślnie mamy przekazany iterator.
3. Zawołaj `next()` jako argument podając otrzymany iterator.
4. Next zwróci następny element, jaki ma do przekazania iterator.
5. Kiedy nie będzie już elementów iterator rzuci wyjątkiem StopIteration, co jest poprawnym zachowaniem dla iteratorów, które nie mają już elementów do przekazania.

```
>>> 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
```

Tak to wygląda w praktyce i to plus minus pod spodem się dzieje, kiedy używamy pętli for i iterujemy sobie po danym obiekcie. W skrócie, tak jak pisałem, po prostu przechodzimy krokowo przez wszystkie elementy, otrzymując pojedynczy element do dyspozycji i mogąc go przetwarzać w jakiś sposób.

Ma to wiele zastosowań i pętla krokowa to taki chleb powszedni często w pythonie. Będziesz widział/widziała ją jeszcze nie raz. Oswój się z nią dość dobrze, bo to będzie prawdopodobnie twój przyjaciel.

10.2 Zwykła pętla

10.2.1 Krótka charakterystyka

Zwykła pętla jak to ją nazywam, to po prostu pętla `while`. O ile pętla krokowa wykonuje się przechodząc po elementach danego iterowalnego obiektu, tak pętla `while` wykonuje się dopóki zadany warunek jest prawdziwy. Koncept nieco podobny co w pętli krokowej, ale kapkę inny.

10.2.2 Przykłady użycia pętli

```
counter = 0
while True:
    counter += 1
    if counter >= 10:
        break
```

I chyba tyle. Reszta do samodzielnej analizy.

10.3 Składanie

Składanie, czy też `comprehensions` i struktury składane to nic innego jak swego rodzaju lukier składniowy, który sprawia, że pewne typowe zachowania można opisać w Pythonie krócej. Zachowania te dotyczą pętli for i tworzenia za jej pomocą nowych obiektów, list, tupli etc.

Przykład tego, jak używać 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)
```

Poczytaj kapkę więcej i poeksperymentuj sobie samodzielnie. Pamiętaj, że comprehensions można zagnieżdżać, czyli mieć listę składaną składaną z listy składanej. Book. Incepcja.

Wspomnę tylko dodatkowo o tym, że ja osobiście lubię dłuższe listy składane rozpisywać w następujący sposób:

```
list_comprehension = [
    value
    for value in range(10)
    if value % 2
]
```

Czyli na trzy linijki, w każdej mamy kolejne elementy. Jeśli comprehension jest krótkie, to tego nie robię, w przypadku bardziej złożonych już tak. Poprawia to moim zdaniem czytelność.

10.4 Generatory

Jak już o pętlach i iteracji mówimy, to wspomnę o generatorach. Cóż to takiego?

Generalnie są to funkcje, które coś `yield`ują zamiast zwracać. Co to znaczy w praktyce i o co chodzi, po co to?

Mówiąc prostym językiem, to zwyczajnie chodzi o to, że czasami mamy zbiory danych, które są za duże by jednorazowo załadować je do RAMu, gdyż ten jest skończony, duży, ale skończony i ograniczony. Co zrobić wtedy? Generator jest jedną ze strategii radzenia sobie w takiej sytuacji. Otóż generatory pozwalają nam przetwarzać duże zbiory danych krok po kroku.

Generatory to pod spodem tak naprawdę zwykłe pętle, opakowane w funkcje, które `yield`ują jakąś wartość. Co to znaczy? Zamiast zwracać dany element i kończyć egzekucje, generator yielduje daną wartość, 'zapisuje' czy 'zapamiętuje' sobie swój obecny stan i czeka na sygnał, by zwrócić kolejny element. W międzyczasie my możemy sobie przetwarzać i robić coś z danymi wartościami.

Generatory mogą działać na skończonych zbiorach, mogą być nieskończone. Wiele wariacji.

```
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>
```

Generatory mają też to do siebie, że są jednorazowe. To znaczy, że raz zainicjalizowany generator można przejść tylko raz. To plus fakt, iż nie ładujemy danych do pamięci na raz to główna różnica między generatorami a tradycyjnymi listami czy krotkami.

10.5 Walrus

Od Pythona 3.8 mamy do dyspozycji coś, co nazywane jest Walrusem lub operatorem przypisania w wyrażenia. Pozwala on nam na przypisywanie zmiennych nie tylko w stwierdzeniach, ale również i wyrażeniach, za pomocą operatora `NAME := expr`.

No dobrze, ale co to znaczy w praktyce. Spójrzmy na kod.

```
data = None
if our_function_getting_json(some_arg) is not None:
    data = our_function_getting_json(some_arg)
    data.do_stuff()
```

Raczej proste do zrozumienia, zasadne, racja? Przykład z pewnego kodu wzięty. Przykład brzydki. Powyższy kod ssie w tym kontekście. Jak można by go poprawić?

```
data = our_function_getting_json(some_arg)
if data is not None:
    data.do_stuff()
```

Możemy zrobić coś takiego, ale czy to najkrócej jak się da, najlepiej jak się da? Obecnie raczej tak, ale...

Fajnie by było, gdyby można było zadeklarować tą zmienną tam w tym ifie - zapisać po prostu wynik funkcji tam, gdzie jest ona pierwotnie używana. Ważne jest to tam, gdzie chcemy później np. wykonać jakieś operacje na wyniku wyrażenia, które wykonaliśmy, np. w warunku, ale przez to, że w wyrażeniach obecnie nie można zapisywać zmiennych, to musimy zapisać ją sobie sami, wcześniej. Czy to w pętlach, czy w list comprehensions, lambda functions czy w innych.

Chyba, że użyjemy walrusa.

```
if (data := our_function_gettin_json(some_arg)) is not None:
    data.do_stuff()
```

Inne przykłady użycia:

```
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]
# lub też coś takiego jak niżej
stuff = [[y := f(x), x/y] for x in range(5)]
y = y1 := f(x) # BŁĄD
bar(x = y := f(x)) # BŁĄD
bar(x = (y := f(x))) # OK
something := 'lalala' # BŁĄD
something2 = 'hey' # OK
```

10.6 Podsumowanie

Pętle, generatory, listy składane etc to przydatne narzędzia dla każdego programisty i podstawowe bloki budulcowe każdego kodu. Dobrze jest być z nimi świetnie zaznajomionym.

10.7 Pytania i zadania

1. Napisz przykłady pętli for, while. Po 3 różne.
2. Napisz artykuł porównujący pętlę for z pętlą while.
3. Do tego charakterystyka porównawcza pętli for i `składania`.
4. Napisz kod, który znajduje największą i najmniejszą liczbę na liście.
5. Napisz kod, który zliczy wyrazy w zadanym stringu.
6. Następnie litery.
7. I częstotliwość ich występowania.
8. Zbadaj czy na zadanej liście, znajdują się dwie liczby – a i b, których suma wynosi zadaną liczbę.
9. Napisz kod, który z dowolnej listy(same liczby) wyświetli tylko te, które są mniejsze od 5.
10. Poproś usera o liczbę a następnie wypisz wszystkie dzielniki tejże liczby.
11. Pobierz od usera dwie liczby a następnie zwróć kwadrat ich sumy.
12. Napisz kod, który wczyta od użytkownika jakiegoś stringa a następnie wyświetli trzy pierwsze litery tego stringa, kolejno po sobie, bez nowych linii. Np. “MelonTusk” -> “MelMelMel”
13. Pobierz od użytkownika stringa a następnie wyświetl go w odwróconej kolejności liter
14. Bazując na kodzie benchmarkującym f-stringi z poprzedniego rozdziału, przeprowadź analizę tego, co jest szybsze. list(), [] czy listy składane. wszystkich niech zawiera liczby naturalne od 1 do 10. Czy jest

Pamiętaj, że Twoje odpowiedzi możesz wrzucić na GH o tutaj - https://github.com/grski/junior-python-exercises, a sprawdzę twoje rozwiązania i dam feedback. Więcej o tym podrozdziale ‘Część interaktywna’.

11 Funkcje

Funkcje to podstawowy element programowania, który pozwala na grupowanie kodu w celu jego powtarzalnego wykorzystania. Są one również używane do dzielenia programu na mniejsze, bardziej zrozumiałe fragmenty, co ułatwia jego tworzenie i utrzymanie, czytelność. A o tym już wspominałem, że jest krytycznie ważne, prawda? W końcu kod piszesz raz, a czytasz dziesiątki razy czasami. Ty i inni ludzie.

Czym są funkcje i dlaczego są ważne w programowaniu? Zaraz pomówimy o tym szerzej. Zacznijmy od zwykłych funkcji.

11.1 Zwykłe funkcje/metody

Najpierw omówmy tradycyjne funkcje, czy jeśli mówimy o funkcjach definiowanych w klasach, metody.

11.1.0.1 Składnia definiowania funkcji w Pythonie. W Pythonie definicja funkcji rozpoczyna się od słowa kluczowego `def`, po którym następuje nazwa funkcji oraz nawiasy z argumentami. Kod wewnątrz funkcji jest wcięty. Przykład definicji funkcji:

```
def nazwa_funkcji(arg1, arg2):
    # kod funkcji
    return wynik
```

11.1.0.2 Przekazywanie argumentów do funkcji Aby wywołać funkcję, należy podać jej nazwę oraz odpowiednie argumenty. W zależności od tego, jakie argumenty są wymagane przez funkcję, należy podać odpowiednią liczbę argumentów.

11.1.0.3 Zwracanie wartości z funkcji Funkcje w Pythonie mogą zwracać wartości za pomocą słowa kluczowego `return`. Po wywołaniu funkcji, kod wewnątrz funkcji jest wykonywany, a następnie wartość jest zwracana i przypisywana do zmiennej lub wykorzystywana w inny sposób. Jeśli funkcja nie zwraca żadnej wartości, domyślnie zwraca `None`.

11.1.0.4 Użycie słowa kluczowego `return` do zwracania wartości z funkcji Aby zwrócić wartość z funkcji, należy użyć słowa kluczowego `return` wraz z wyrażeniem, które ma zostać zwrócone. Na przykład:

```
def powieksz(x):
    return x + 1

a = 5
b = powieksz(a)
print(b)  # Output: 6
```

11.1.0.5 Wincyj wartości do zwrotu W Pythonie możliwe jest zwracanie wielu wartości z funkcji za pomocą krotki lub słownika. Aby zwrócić wiele wartości w formie krotki, należy je po prostu oddzielić przecinkami. Przykład:

```
def min_max(x):
    return min(x), max(x), x

a = [1, 2, 3]
min_a, max_a, _ = min_max(a)
# alternatywnie:
result = min_max(a)
min_a = result[0]
max_a = result[1]
print(min_a)  # Output: 1
print(max_a)  # Output: 3
```

Przy okazji linijka z wywołania tejże funkcji, to przykład tak zwanego tuple unpacking - rozpakowywania krotki.

Przydatna i ważna rzecz.

Jeśli stwierdzamy, że drugi argument zwracany przez funkcję jest nam zbędny albo, że w funkcji, która zwraca trzy argumenty, chcemy tylko pierwszy, albo tylko drugi, lub może ostatni, to też się da. Jak? Przeanalizuj kod niżej.

```
def min_max(x):
    return min(x), max(x), x

a = [1, 2, 3]
min_a, max_a, _ = min_max(a)
# trzeci argument zostanie olany i nieprzypisany do
# żadnej zmiennej _ to placeholder, który python rozpoznaje
*_, last_argument = min_max(a)
first, *all_the_rest = min_max(a)
first, _, the_third = min_max(a)
first, *_, the_third = min_max(a)
```

Pobaw się kodem powyżej i zobacz jakie wnioski wyciągniesz. Dodatkowo polecam spróbować stworzyć funkcje z większą ilością argumentów, które zwraca.

Aby zwrócić wiele wartości w formie słownika, należy utworzyć słownik i zwrócić go za pomocą słowa kluczowego `return`. Nic odkrywczego.

```
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
```

11.1.0.6 Domyślne wartości argumentów W Pythonie możesz zdefiniować domyślne wartości argumentów funkcji, co oznacza, że argumenty te nie muszą być przekazywane do funkcji w momencie jej wywołania. Wartość domyślna jest używana w przypadku, gdy argument nie został przekazany do funkcji. Natomiast trzeba pamiętać tutaj o jednej ważnej rzeczy, przy definiowaniu wartości domyślnych, o czym zaraz.

Oto przykład funkcji z domyślnymi argumentami:

```
def greet(name, greeting='Hello'):
  print(f'{greeting}, {name}!')

greet('Alice')  # wypisze "Hello, Alice!"
greet('Bob', 'Hi')  # wypisze "Hi, Bob!"
```

W powyższym przykładzie argument `name` nie ma wartości domyślnej, więc musi być przekazany do funkcji podczas jej wywołania. Natomiast argument `greeting` ma wartość domyślną `'Hello'`, więc może być pominięty podczas wywołania funkcji. Jeśli jednak przekazujesz drugi argument do funkcji, to jego wartość zostanie użyta zamiast wartości domyślnej.

Drugi argument można przekazać na dwa sposoby.

```
greet('Bob', 'Hi')  # wypisze "Hi, Bob!"
greet('Bob', greeting='Hi')  # wypisze "Hi, Bob!"
```

Ten drugi sposób to używanie keyword arguments - argumentów z domyślnymi wartościami, które są opcjonalne. Argumenty bez domyślnych wartości też można przekazywać jako keyword arguments.

```
greet('Bob', 'Hi')  # wypisze "Hi, Bob!"
greet(name='Bob', greeting='Hi')  # wypisze "Hi, Bob!"
```

Nazwa argumentu jest używana jako klucz. Możesz używać keyword arguments, aby przekazywać argumenty do funkcji w dowolnej kolejności, niezależnie od kolejności, w jakiej zostały zdefiniowane w definicji funkcji, ale lepiej trzymać się konwencjonalnej kolejności. To, że można, nie znaczy, że wypada.

Oto przykład użycia keyword arguments:

```
def greet(name, greeting='Hello'):
  print(f'{greeting}, {name}!')

greet(name='Alice')  # wypisze "Hello, Alice!"
greet(greeting='Hi', name='Bob')  # wypisze "Hi, Bob!"
```

W powyższym przykładzie argument `name` jest przekazywany do funkcji za pomocą keyword argument `name`, a argument `greeting` jest przekazywany za pomocą keyword argument `greeting`. Możesz zamienić kolejność tych argumentów podczas wywołania funkcji, a funkcja nadal będzie działać poprawnie.

Uwaga: keyword arguments muszą być umieszczone po argumentach pozycyjnych (czyli takich, które nie są określone przez nazwę) podczas wywołania funkcji i podczas jej definicji.

```
def greet(name, greeting='Hello'):
  print(f'{greeting}, {name}!')

greet('Alice', greeting='Hi')  # poprawne
greet(greeting='Hi', 'Alice')  # błąd SyntaxError
def greet(greeting='Hello', name):
  print(f'{greeting}, {name}!')  # błąd
```

11.1.1 Śledzik na raz

Wartości domyślne argumentów są jak śledzik na raz - interpreter Pythona zerknie na nie tylko raz, chluśnie i więcej nie będzie się zajmował. Co to znaczy? Ano to, że interpreter działa w taki sposób, że jak może, inicjuje domyślne argumenty tylko raz. O ile przy stringach, liczbach to nie ma problemu, tak przy obiektach mutowalnych pojawia się problem. Jak sądzisz, jaki?

Otóż każde kolejne wywołanie funkcji, wbrew temu czego by się można spodziewać, nie będzie powodować stworzenia nowej instancji elementu domyślnego a po prostu użyje referencji do na początku zainicjowanego obiektu np. listy. Rozważ przykład niżej.

```
>>> 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]
```

Zajrzyj do tego kodu i przemyśl. Często pojawia się to na rekrutacjach jako haczyk :-).

11.1.1.1 Funkcje jako obiekty

W Pythonie istnieje możliwość przypisywania funkcji do zmiennych i przekazywania ich jako argumentów do innych funkcji.

Ponieważ funkcje są obiektami, można je przypisywać do zmiennych, tak jak przypisywana jest wartość zmiennej. Można również przekazywać funkcję jako argument do innej funkcji. Przykład:

```
def powieksz(x):
    return x + 1

def wykonaj_na_liscie(funkcja, l):
    return [funkcja(x) for x in l]

a = [1, 2, 3]
```

```
b = wykonaj_na_liscie(powieksz, a)
print(b)  # Output: [2, 3, 4]
```

11.1.1.2 Czysta funkcjowo Eur—- sieć Pure functions/czyste funkcje to funkcje, które zawsze zwracają tę samą wartość dla danych wejściowych i nie wpływają na stan programu poza swoim zasięgiem. Innymi słowy, pure functions nie modyfikują stanu globalnego, nie wywołują żadnych efektów ubocznych (takich jak zmiana zmiennej globalnej lub wyświetlenie czegoś na ekranie) i zawsze zwracają tę samą wartość dla danego zestawu wejściowych. To trochę takie odwzorowanie funkcji znanej nam z matematyki. Mapowanie wartości dla każdego argumentu.

Oto przykład czystej funkcji:

```
def add(x, y):
  return x + y

print(add(1, 2))  # wypisze 3
print(add(1, 2))  # wypisze 3
print(add(1, 2))  # wypisze 3
```

Powyższa funkcja `add` jest czysta, ponieważ zawsze zwraca tę samą wartość dla danego zestawu wejściowych (tutaj 1 i 2) i nie ma żadnych efektów ubocznych.

Pure functions są często używane w programowaniu funkcyjnym, ponieważ są one łatwe do testowania i łatwe do zrozumienia. Ponadto pure functions są często bardziej niezawodne niż funkcje, które wywołują efekty uboczne, ponieważ nie występuje ryzyko, że ich działanie będzie zależało od stanu globalnego lub innych niezdefiniowanych zmiennych.

Ja osobiście całkiem lubię. Pewne aspekty programowania funkcyjnego są mi dość bliskie, plus nawet używane w OOP, poprawiają jakość kodu. Kiedy staram się pisać w miarę czyste funkcje, zauważam często, że kod wychodzi lepszy. Oczywiście wszystko z głową, nie polecam bycia purystą i fanatykiem. Wiedz, kiedy zrobić wyjątek.

11.1.1.3 *args, **kwargs Słowa kluczowe `*args` i `**kwargs` służą do przekazywania dowolnej liczby argumentów do funkcji.

`*args` jest używane do przekazywania dowolnej liczby argumentów niekluczowych (tzw. "positional arguments") do funkcji. Argumenty te są przekazywane w postaci krotki.

Przykład:

```
def func(*args):
    print(args)

func(1, 2, 3)
# Output: (1, 2, 3)
```

`**kwargs` jest używane do przekazywania dowolnej liczby argumentów kluczowych (tzw. "keyword arguments") do funkcji. Argumenty te są przekazywane do funkcji w postaci słownika.

Przykład:

```
def func(**kwargs):
    print(kwargs)

func(a=1, b=2, c=3)
# Output: {'a': 1, 'b': 2, 'c': 3}
```

Słowa kluczowe `*args` i `**kwargs` są szczególnie przydatne, gdy chcemy przekazać dowolną liczbę argumentów do funkcji, bez konieczności znajomości ich dokładnej liczby lub nazw. Można również użyć obu słów kluczowych jednocześnie, jeśli chcemy przekazać zarówno argumenty pozycyjne, jak i kluczowe. Można używać ich razem - `*args, **kwargs`.

11.1.1.4 Wskazówki Aby napisać czytelny i efektywny kod z użyciem funkcji, należy pamiętać o kilku ważnych wskazówkach:

- Dziel kod na mniejsze fragmenty za pomocą funkcji. Dzięki temu łatwiej będzie go czytać i zrozumieć.
- Nazywaj funkcje tak, aby opisywały, co robią. Nazwy powinny być zrozumiałe dla odbiorców kodu.
- Staraj się unikać powtarzania kodu i metody copiego pasty. Twórz atomiczne, małe i reużywalne funkcje.
- Określaj dokładnie, jakie argumenty są wymagane przez funkcję i jakie wartości zwraca. Type Hinting i Docstringi twoimi przyjaciółmi.
- Idealnie jest kiedy funkcja ma jak najmniej efektów ubocznych - nie zależy od rzeczy z zewnątrz, poza nią.
- Staraj się ograniczyć liczbę argumentów przekazywanych do funkcji do minimum. Im mniej argumentów, tym łatwiej będzie ją zrozumieć.
- Funkcja z jednym argumentem jest ideolo. Dwa prawie tak samo dobra. Powyżej już się zastanów. Może warto opakować te argumenty w jakąś klasę/obiekt?
- Pamiętaj o testowaniu funkcji. Przetestuj każdą funkcję, aby upewnić się, że działa poprawnie. Małe, krótkie, czyste funkcje łatwo testować.

11.2 Funkcje anonimowe/lambda

Tutaj będzie naprawdę krótko. Funkcje anonimowe/lambda to po prostu jednolinijkowe funkcje, którym nie nadajemy nazwy, gdyż używamy ich tylko w określonym lokalnym miejscu lub do przekazywania funkcji jako argumentu do innej funkcji.

Nie jest dobrą praktyką, by ich nadużywać, natomiast w pewnych sytuacjach mają swoje zastosowanie. Zazwyczaj preferuję zwykłe funkcje, gdyż pozwalają mi bardziej rozlegle i opisowo zaznaczyć co dany kod robi, nadać nazwę. Są oczywiście trywialne przykłady i miejsca, gdzie lambdy mają sens. Warto jednak pamiętać, żeby nie tworzyć potworków, które posiadają milion zagnieżdżonych lambd czy skomplikowaną logikę.

Poniżej przykłady.

```
# Zwykła funkcja
def add(x, y):
```

```
    return x + y

# Funkcja lambda
add = lambda x, y: x + y
```

11.2.1 Gdzie używamy lambd najczęściej?

Funkcje lambda są często używane w połączeniu z funkcjami takimi jak `map()`, `filter()` i `reduce()`, które przyjmują funkcje jako argumenty, `sorted()` etc.

11.2.2 Przykłady

Oto przykład użycia funkcji lambda z `map()` jak i kilka innych przykładów:

```
numbers = [1, 2, 3, 4]
doubled = map(lambda x: x * 2, numbers)
# doubled jest teraz [2, 4, 6, 8]

# funkcja lambda zwracająca kwadrat danej liczby
square = lambda x: x**2
print(square(5))  # wypisze 25

# funkcja lambda zwracająca większą z dwóch liczb
max_number = lambda x, y: x if x > y else y
print(max_number(4, 5))  # wypisze 5

# funkcja lambda przyjmująca listę i zwracająca sumę jej elementów
sum_list = lambda lst: sum(lst)
print(sum_list([1, 2, 3]))  # wypisze 6

# funkcja lambda przyjmująca listę i zwracająca listę złożoną z elementów parzystych
even_elements = lambda lst: [x for x in lst if x % 2 == 0]
print(even_elements([1, 2, 3, 4, 5]))  # wypisze [2, 4]
```

W powyższym kodzie użyto funkcji lambda do stworzenia krótkich funkcji zwracających kwadrat danej liczby, większą z dwóch liczb, sumę elementów listy oraz listę złożoną z elementów parzystych.

Pamiętaj, że funkcje lambda są ograniczone do jednej linii i nie można ich używać do bardziej złożonych zadań. W takich przypadkach lepiej jest użyć zwykłej funkcji.

Przykład z nieco bardziej życiowego kodu:

```
def sort_timestamp(orders):
    return sorted(orders, key=lambda x: x.timestamp)
# inny przykład
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
            )
            (...)
```

Kod powyżej sortuje obiekty po jednym z jego atrybutów.
Do poczytania:

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

12 Klasy i OOP

12.1 Klasy

Można o klasach myśleć, w uproszczeniu, jak o po prostu pewnych zbieraninach funkcji. Funkcje tworzone wewnątrz klasy nazywają się nagle `metodami`.

Klasy można 'łączyć', co nazywa się dziedziczeniem. Jak jedna klasa dziedziczy po drugiej, to przyjmuje jej metody, o ile ich nie nadpiszemy. Przykład klasy:

```
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 max(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")
```

Do poczytania:

1. https://realpython.com/inheritance-composition-python/

12.1.1 super() i MRO

`super()` to nic innego jak sposób wywołania metody z klasy po której dziedziczymy. Tylko tyle i aż tyle. Czyli coś a'la jak matka krzyczy 'zawołaj starego'.

Jeśli dziedziczymy po kilku klasach, co w Pythonie jest dozwolone i częste w zasadzie, które implementują tę samą metodę, to to, która metoda zostanie użyta, jest decydowane przez MRO. Method resolution order.

Method Resolution Order (MRO) to sposób, w jaki Python odwzoruje dziedziczenie wielokrotne w klasach. MRO określa kolejność, w jakiej Python szuka metod w klasach podczas wywoływania metody na obiekcie i z której klasy odpala metodę.

Przykładowo, jeśli mamy następujące trzy klasy:

```
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")
```

a następnie tworzymy obiekt klasy `D`, która dziedziczy zarówno po klasie `B`, jak i `C`:

```
class D(B, C):
    pass
```

Jeśli teraz wywołamy metodę `method` na obiekcie `D`, to Python użyje MRO do określenia, która wersja metody zostanie wywołana. W tym przypadku, ponieważ klasa `D` dziedziczy po klasie `B` jako pierwszej, to Python wywoła wersję metody z klasy `B`. Jeśli klasa `B` nie miałaby tej metody, Python przeszukałby klasę `C`, a następnie klasę `A`.

MRO działa według algorytmu C3, który zapewnia spójne i łatwe do przewidzenia wyniki. Można zobaczyć MRO dla danej klasy, używając funkcji `__mro__`:

```
>>> D.__mro__
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A
```

MRO jest ważne, ponieważ umożliwia kontrolę nad tym, w jaki sposób dziedziczenie wielokrotne jest odwzorowywane w kodzie. Może to być szczególnie przydatne w przypadku klas, które dziedziczą po wielu klasach bazowych i chcą mieć pewność, że metody zostaną odpowiednio wywołane.

Algorytm C3 działa następująco:

1. Wszystkie klasy są umieszczane na liście, w kolejności, w jakiej są podane jako argumenty dziedziczenia. Na przykład, w klasie `D` zdefiniowanej jako `class D(B, C)`, klasa `B` znajduje się przed klasą `C`.

2. Dla każdej klasy na liście, dodaj jej klasę bazową do końca tej samej listy.

Czyli najpierw od góry do dołu potem lewo do prawa.
Resztę pominę.
Algorytm C3 jest stosowany w Pythonie od wersji 2.3. Jest on uważany za bardziej elegancki i prosty niż poprzedni algorytm stosowany w Pythonie (algorithmic depth-first search). Algorytm C3 zapewnia spójne i łatwe do przewidzenia wyniki dla MRO, co umożliwia lepszą kontrolę nad dziedziczeniem wielokrotnym w klasach i mniej niespodzianek na produkcji :)
Do poczytania: https://www.educative.io/answers/what-is-mro-in-python

12.2 Classmethods, staticmethods

Koncept, jaki warto kojarzyć, by tworzyć ładne interfejsy i sensowne klasy to metoda klasowa i metoda statyczna.

Co to znaczy? Metoda klasowa/classmethod to taka metoda, która nie potrzebuje instancji danej klasy, jedynie samej klasy. Znaczy to tyle, że nie będziemy mieli dostępu do zainicjalizowanego obiektu i jego atrybutów, które definiujemy w `__init__` a jedynie do zmiennych na poziomie samej klasy, czyli w jej scopie.

Metoda statyczna, to metoda, która nie potrzebuje nawet zmiennych z klasy i się do nich nie odnosi, nie odnosi się też do innych metod z danej klasy.

Czyli tak w skrócie: jeśli potrzebujemy stanu obiektu albo samego obiektu, zwykłe metody. Jeśli tylko rzeczy z danej klasy, to metoda klasowa, jeśli nic z powyższych to po prostu metoda statyczna. Przykłady na dole.

```
from datetime import date


class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    # zwykła metoda korzystająca z atrybutów instancji
    def print_name(self):
        print(self.name)

    # metoda klasowa tworząca instancje danej klasy
    # na podstawie roku
    @classmethod
    def from_birth_year(cls, name, year):
        return cls(name, date.today().year - year)

    # statyczna metoda do sprawdzania dorosłości
    @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))
```

Do poczytania: https://www.geeksforgeeks.org/class-method-vs-static-method-python/?ref=lbp

12.3 Menadżery kontekstu

Menadżery kontekstu to takie klasy, które definiują `__enter__` oraz `__exit__`. To te cosie, których używamy razem z klauzulą `with`. W skrócie, te klasy po prostu definiują magiczne metody, które są odpalane przy wejściu do bloku kodu z with oraz po ukończeniu przetwarzania tego bloku i wyjściu z niego. Pozwalają one nam, cóż, ustawić jakiś określony kontekst a potem po nim posprzątać.

Przykładem dobrym są tu operacje na plikach. Najpierw chcemy plik otworzyć, ustawić odpowiednio kursor etc a dopiero na nim pracować. Jak skończymy pracę na pliku to chcielibyśmy go zamknąć, żeby nic nie wisiało w pamięci. Zamiast robić to ręcznie za każdym razem, używamy kontekst menadżera, który wchodzi cały na biało.

```
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 jako 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!')
```

Do poczytania: https://realpython.com/python-with-statement/

12.4 Typehints

Type hinting to mechanizm w Pythonie, który pozwala niejako "podpowiedzieć" programiście jakiego typu danych oczekujemy w danym miejscu programu. W Pythonie nie ma konieczności deklarowania typów zmiennych, więc type hinting jest opcjonalnym narzędziem, które można wykorzystać w celu ułatwienia kodowania lub dokumentowania kodu.

Type hinting może być używany w kilku różnych miejscach kodu, takich jak deklaracje funkcji, zmiennych i metod, oraz w komentarzach.

Przykłady użycia type hinting:

```
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
```

Zaletą używania type hinting jest to, że może pomóc w dokumentowaniu kodu i ułatwić jego odczytanie dla innych programistów. Type hinting może również pomóc w detekcji błędów w czasie kompilacji, ponieważ interpreter może zgłaszać błędy, jeśli dane oczekiwanego typu nie zostaną przekazane do funkcji lub metody.

Wadą używania type hinting jest to, że może on być uciążliwy w implementacji, zwłaszcza w dużych projektach, gdzie konieczne jest ręczne dodawanie type hinting do wielu miejsc w kodzie. Ponadto, ponieważ type hinting nie jest obowiązkowy w Pythonie, niektórzy programiści mogą nie używać go w swoich projektach, co może utrudnić współpracę i odczytanie kodu przez innych, ale olać ich. Type hinting jest w pytę, ja polecam.

Do poczytania:

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

12.5 Docstrings

Docstringi (ang. "documentation strings") to ciągi znaków umieszczane w kodzie Python, które służą jako dokumentacja do kodu. Są one umieszczane bezpośrednio po deklaracji funkcji, metody, klasy itp. i są zwykle umieszczane w potrójnych cudzysłowach.

Docstringi są często używane do opisywania tego, co dana funkcja, metoda lub klasa robi, jakie argumenty przyjmuje i jakie wartości zwraca. Docstringi są później używane przez narzędzia do tworzenia dokumentacji (np. Sphinx) do automatycznego tworzenia dokumentacji kodu.

Oto przykład użycia docstringów w Pythonie:

```
def add(x, y):
  """
  Funkcja dodająca dwie liczby.

  Args:
    x (int): pierwsza liczba do dodania
    y (int): druga liczba do dodania

  Returns:
    int: suma dwóch liczb
  """
  return x + y
```

W powyższym przykładzie docstring opisuje, co robi funkcja `add`, jakie argumenty przyjmuje i jakie wartości zwraca.

Uwaga: pamiętaj, że docstringi muszą być umieszczone bezpośrednio po deklaracji funkcji i muszą być zamknięte w potrójne ". Docstringi nie mogą zawierać żadnych instrukcji kodu ani nie mogą być używane jako zwykłe komentarze.

Docstringi i type hinty to diabelsko użyteczne rzeczy, które naprawde gorąco polecam stosować. Nawet jak pracujesz samodzielnie nad projektem amatorskim. Dlaczego? Bo sam też zapominasz co robił dany kod. O wiele łatwiej wrócić do własnego projektu nawet, wtedy, kiedy jest on dobrze udokumentowany za pomocą type hintów czy docstringów i treściwych komentarzy.

Bzdury typu "Dobry kod nie wymaga komentarzy" zostawmy między bajkami.

12.6 Operator is

Operator `is` w Pythonie służy do sprawdzania, czy dwie zmienne wskazują na to samo miejsce w pamięci. Natomiast operator `==` służy do sprawdzania, czy dwie zmienne zawierają tę samą wartość.

Oto przykład użycia operatorów `is` i `==`:

```
x = [1, 2, 3]
y = x
z = [1, 2, 3]
```

```
print(x is y)  # wypisze True
print(x is z)  # wypisze False
print(x == y)  # wypisze True
print(x == z)  # wypisze True
```

W powyższym przykładzie zmienna `x` i `y` wskazują na to samo miejsce w pamięci (obie zmienne wskazują na tę samą listę), dlatego operator `is` zwraca wartość True. Zmienna `z` zawiera tę samą wartość co zmienna `x`, ale wskazuje na inną lokalizację w pamięci, dlatego operator `is` zwraca False. Natomiast operator `==` sprawdza tylko wartości obu zmiennych, więc zwraca True zarówno dla porównania `x == y`, jak i `x == z`.

Uwaga: pamiętaj, że operator `is` jest szybszy niż operator `==`, ponieważ nie musi porównywać wartości zmiennych, ale sprawdza tylko, czy wskazują one na to samo miejsce w pamięci. Dlatego operator `is` jest często używany w miejscach, gdzie szybkość jest ważna, a dokładność porównania nie jest konieczna.

Dodatkowo `is` to element języka, 'niezmienialny' powiedzmy. Zaś użycie operatora `==` zależy od tego, jak zaimplementowano magiczną (dunder) metodę `__eq__`. Co to znaczy? Otóż możemy sami definiować to, jak Python będzie porównywał obiekty przy `==`. Poczytaj.

12.7

13 Ekosystem narzędzi Pythonowych

Trochę o fajnych narzędziach, które wypada znać. Zakładam, że każde z tych narzędzi sobie zainstalujesz, pobawisz się nim.

13.1 Plik README - co powinien zawierać i jak wyglądać

Jest jedna rzecz, którą każdy porządny projekt powinien zawierać. Plik README. Dobrze napisany, czytelny i obszerny plik README jest podstawową formą dokumentacji dla każdego projektu, zawierającą elementarne informacje o samym projekcie, ale moim zdaniem nie tylko. Dobrze napisany plik README powinien zawierać nieco więcej. Co dokładnie? Już opisuję.

13.1.1 Technologia

Zazwyczaj gdy pracujemy z plikami README, to mają one rozszerzenie `.md`, czyli według konwencji piszemy je w Markdownie. Markdown to coś, co pozwala nam niejako formatować tekst. Używanie go w README nie jest wymogiem, ale dość powszechnie przyjętym standardem, który ułatwia nam nieco życie i daje większe możliwości aniżeli zwykły plik tekstowy.

13.1.2 Jakie sekcje powinien zawierać dobry plik README?

Poniżej przedyskutujemy to, jakie sekcje, co opisujące, powinien zawierać dobry plik README.

13.1.3 Tytuł

README zaczynamy oczywiście od tytułu. By to zrobić, należy po prostu wpisać w linii tytuł a przed nim dodać `#`, co oznaczy daną linię jako nagłówek Markdown.

13.1.4 Opis projektu

Tutaj umieszczamy elementarne informacje o projekcie.

1. Jaki to komponent systemu? Np. API, Worker, frontend, biblioteka.
2. Za jakie funkcjonalności odpowiada? Np. Jest to API systemu generującego faktury.
3. Jaki jest kontekst? Trochę dodatkowego kontekstu, tego nigdy za wiele. Np. Faktury te są potem wysyłane do klientów, (...). Używamy tutaj standardowego szablonu z książki wzornictwa firmy.
4. Kim są stakeholderzy - kto jest odbiorcą tego projektu?
5. Jaki problem biznesowy rozwiązuje ten projekt?
6. Kim są jego użytkownicy końcowi?

Podsumowanie powinno być napisane w języku domeny danego problemu. Co to znaczy? Otóż jego opis, opis funkcjonalności projektu powinien zawierać słownictwo z danej dziedziny w której działamy. Jeśli robimy projekt o traktorach, to posługujmy się terminami traktorzystów. Będąc między wronami, kracz jak one. Jest to preferowane podejście w porównaniu do posługiwania się czystym, suchym technicznym żargonem.

13.1.5 Stos technologiczny

W tej sekcji opisujemy najważniejsze technologie użyte, przy tworzeniu projektu. Zależności, używane aplikacje zewnętrzne, serwisy etc.

To pozwoli czytającemu na szybkie zapoznanie się z wyborami technologicznymi dokonanymi w projekcie, użytymi do rozwiązania danego problemu.

Technologie powinny być króciutko opisane, odpowiednie materiały zalinkowane, dla wygody czytającego.

13.1.6 Instrukcja tworzenia środowiska lokalnego

Tutaj umieszczamy kroki, które należy wykonać, aby lokalnie postawić środowisko. Dodatkowo to w tej sekcji umiejscawiamy instrukcje jak wykonać często używane operacje, jakich komend najczęściej się używa jak np. czyszczenie bazy danych, albo jej tworzenie, migracje etc. To tutaj ląduje też wiedza, która jest mocno specyficzna dla danego projektu np. jak rozwiązano problem lokalizacji, internacjonalizacji używając np. PhraseApp czy OneSky.

Zalecam, by akurat tę sekcję opisać szczególnie dobrze, mając na wzgląd użytkowników mniej technicznych, który być może będą potrzebowali postawić środowisko lokalnie w celach testowych. Czasami są to nietechniczne osoby, testerzy, stakeholderzy, produkt ownerzy etc. Im też należy się możliwość posiadania środowiska lokalnego. Dodatkowo cały proces stawiania środowiska powinien być jak najbardziej zautomatyzowany.

13.1.7 Deployment

W kilku zdaniach, zwięźle i krótko, należy opisać jak aplikacja jest zdeployowana, w jakim środowisko żyje, gdzie należy szukać bardziej szczegółowego opisu architektury i tak dalej. Do tego kilka słów o CI/CD

13.1.8 Autorzy

Lista głównych osób w projekcie. Przydatne, gdy przeskakujemy na nowy projekt i szybko chcielibyśmy ustalić, kto trzyma kontekst i kogo najlepiej pytać o rzeczy, bo ma największą wiedzę co do projektu.

13.1.9 Podsumowanie

Plik README to ważna i integralna część systemu. Teraz wiesz, jak powinien wyglądać.

13.2 pdoc3

Automatyczna generacja najważniejszej rzeczy na świecie, czyli dokumentacji. To narzędzie plus obszerne i sensowne docstringi == szczęśliwy developer.

Do poczytania: https://pdoc3.github.io/pdoc/

13.3 Pycharm/Visual Studio Code

Najlepsze IDE/edytor do Pythona. Ja akurat jestem #teamPyCharm albo Visual Studio Code też daje rade.

13.4 Robienie notatek

Jest milion narzędzi do tego. Ja polecam od siebie np. Notion. Inne opcje?

Do poczytania: https://bootcamp.uxdesign.cc/i-tried-40-project-note-taking-apps-what-i-chose-and-my-top-10-list-1d39d41852e4

13.5 Pyenv, poetry i inne nicponie

Pyenv, poezja i inne łobuzy - nowoczesne zarządzanie zależnościami i wersjami Pythona, czyli o współczesnych wersjach Pythona, środowiskach i zarządzaniu zależnościami.

13.5.0.1 PIP Pip to narzędzie, które większość z Was powinna już znać. Służy do instalowania pakietów używanych w rozwoju Pythona i od kilku wersji jest domyślnie dostarczany z Pythonem. Ale co to dokładnie znaczy instalować pakiety?

W skrócie pip dostarcza narzędzi do pobierania pakietów z Python Package Index - PYPI. Jest to domyślny indeks pakietów Pythona, gdzie prawie każdy może dodać pakiety. Domyślny to dobre słowo, ponieważ pip pozwala nam używać różnych indeksów. Tak więc na przykład twoja firma może mieć swoją własną, hostowaną wersję pakietów, a następnie używać jej jako prywatnej wersji pypi. Pozwala to na przykład na lepszą weryfikację pakietów, tylko prywatne połączenia sieciowe podczas procesów CI/CD/development. Jest to dość ciekawa opcja, szczególnie biorąc pod uwagę ostatnie złośliwe ataki na popularne pakiety Pythona open source.

13.5.0.2 Indeks pakietów Czym dokładnie jest indeks pakietów/package indeks?

Właściwie to nic skomplikowanego. Jest to po prostu serwer http, powiedzmy, który dostarcza listę pakietów/paczek pythonowych - pakietów i pewnych metadanych o nich. Nic więcej.

Ciekawe zadanie do domu, aby poeksperymentować z czymś nowym: spróbuj zaimplementować własną wersję pypi i dodać do niej pewne ficzery, takie jak chroniony przez tokeny dostęp do paczek lub nawet więcej - tokeny z funkcją granularnych uprawnień/permissionów.

13.5.0.3 Domyślna instalacja pakietów Zazwyczaj jest tak, że mamy jedną, maksymalnie dwie wstecznie niekompatybilne wersje Pythona zainstalowane na naszym komputerze. W przeszłości był to Python2 & Python3, obecnie większość czasu tylko Python3 jest zainstalowany jako że python2 umar, zginoł, przepadł w odmętach przeszłości.

Tak czy inaczej. Oznacza to, że w Ciemnych Wiekach lub domyślnie instalowano pakiety globalnie, dla całego systemu. To jest złe z wielu powodów. Jeśli chodzi o to, co oznacza instalacja pakietu, w bardzo dużym skrócie, jest to nic innego jak pobranie pakietu kodu pythona o określonej strukturze, który zostaje pobrany i umieszczony w danym katalogu instalacji pythona, z dodatkowymi krokami możliwymi pomiędzy.

Co jeśli projekt A wymaga pakietu Z w wersji 1.0.0, ale projekt B wymaga pakietu Z w wersji 2.0.0? Czy przeinstalowałbyś ten pakiet za każdym razem, gdy przechodzisz do różnych projektów?

13.5.0.4 virtualenv Aby zwalczyć problem opisany w poprzednim akapicie -> pakiety instalujące się globalnie, pojawił się virtualenv. W skrócie jest to coś, co pozwala nam "stworzyć" inną, "instancję" instalacji Pythona. Np. tylko dla danego projektu zamiast ogólnosystemowej.

W ten sposób możemy mieć różne wersje pakietu Pythona dla różnych projektów.

Podzbiór funkcjonalności virtualenv jest zintegrowany z domyślną instalacją CPythona od wersji 3.3 jako moduł venv.

13.5.0.5 Poetry Co by było, gdyby pip i virtualenv miały dziecko, które w dodatku wali sterydy? Cóż, otrzymalibyśmy Poetry.

Problemem z pipem jest zazwyczaj zarządzanie wersjami zależności. Nawet jeśli wiemy, że nasz projekt A, wymaga pakietu Z w wersji 1.0.0, zazwyczaj na pierwszy rzut oka pip nie mówi nam o zależnościach tego pakietu Z. Wprowadza to możliwość wystąpienia problemów, gdy twój projekt osiągnie punkt, w którym ma zainstalowane trochę więcej pakietów. Ponieważ te pakiety również mają zależności, a ich zależności również mają zależności. Zależnościocepcja.

Zazwyczaj nie jest to piekło zależności jak w światach JS, ale w pewnym momencie może również stać się nieco podstępne, jeśli zablokujesz zależności tylko na najwyższym poziomie. I w pewnym momencie, jest to prawie gwarantowane, że będą z tym problemy. A poza tym - jeśli wersje tych zależności nie są domyślnie gwarantowane, co z debugowaniem? Mam na myśli, że jeden build może mieć wersje 1.2.3 jakiejś zależności, ale inny build, wykonany 10 minut wcześniej może mieć 1.2.2, jeśli wersje nie są rozwiązywane w deterministyczny, gwarantowany sposób, umożliwia to pojawienie się paskudnych błędów.

Jest to również zagrożenie bezpieczeństwa, ponieważ jeśli nie wiesz, jaką wersję zależności dokładnie masz, złośliwa wersja może znaleźć drogę bez naszej wyraźnej wiedzy, co jest całkiem łatwym sposobem na strzelenie sobie w stopę. Jak temu zaradzić?

Mamy coś, co nazywa się dependency resolving i dependency locking. Zasadniczo jest to po prostu proces upewniania się, że znamy zależności naszych zależności i ich zależności, a także mamy jasną listę ich wersji, zwykle podpisanych hashem. Pozwala to na coś, co nazywa się deterministycznymi buildami, co jest jednym z kluczowych elementów nowoczesnych CI/CD i aplikacji, które trzymają się wzorca Twelve-factor app.

To jest dokładnie to, co robi Poetry. A robi to prze-elegancko.

Poza tym, skoro już przy tym jesteśmy, Poetry ułatwia również zarządzanie projektami, zajmuje się tworzeniem i zarządzaniem virtualens za Ciebie i umożliwia łatwiejszą, bardziej scentralizowaną konfigurację projektu poprzez wprowadzenie pyproject.toml. pyproject.toml jest teraz zazwyczaj nowym standardowym plikiem konfiguracyjnym projektów pythonowych. A i jeszcze ułatwia budowanie pakietów, ponieważ może spakować/zbundlować twój kod pythona i opublikować go w indeksie pakietów twojego wyboru np. w pypi.

Ogólnie rzecz biorąc, Poetry jest w pytę.

13.5.0.6 Pyenv Python to osobliwe małe zwierzę, które zrzuca skórę od czasu do czasu. Oznacza to, że sam Python, tak jak nasze zależności, również ma swoje wersje. Każda wersja zawiera nowe funkcje, różne ulepszenia. Niektóre z nich są czasem nawet wstecznie niekompatybilne. Standardowo nie jest trywialne zainstalowanie różnych wersji Pythona i ogarnięcie tego, by jedna z drugą nie kolidowały.

Ale po co to w ogóle robić? Cóż, tak samo jak w przypadku zależności. Jeden projekt może zależeć od Pythona 3.10, inny od 2.7, a jeszcze inny od 3.12. Potrzebujemy czegoś takiego jak virtualenv, który zapewniłby izolację, ale zamiast na poziomie projektu, to na poziomie systemu i nie dla paczek ale dla wersji pythona..

Jak to zrobić? Za pomocą pyenv. Pyenv został wzbogacony o zgrabną wtyczkę, która pozwala nam tworzyć takie jakby virtualeny, ale z różnych wersji/interpretacji pythona. Pyenv + pyenv-virtualenv pięknie się integruje z Poetry.

Tak czy inaczej. Mamy pyenv-virtualenv, który jest wrapperem dla pyenv, który z kolei jest opakowaniem wokół zarządzania wersjami pythona, pracujący z Poetry, która jest wrapperem dla pip i pip-tools, zintegrowanym z virtualenv, który jest również rodzajem wrappera.

Mamy więc wrapper wrappera pracującego nad wrapperem wrappera. Wrapperocepcja, zależnościocepcja, dziki, kuny, jenoty.

13.5.1 Piptools

Jeśli twój projekt jest wystarczająco prosty lub nie chcesz zawracać sobie myśli tym wszystkim, możesz użyć pip-tools do przypięcia swoich zależności i mieć z głowy całe to rozważanie. pip-tools jest wystarczająco dobry dla niektórych projektów, natomiast ja w ramach wygody i innych rzeczy, jakie oferuje poetry, korzystam tak naprawdę praktycznie wszędzie, gdzie mogę. Convention over configuration.

13.6 Walimy pythona sprzętem

W Pythona czasami trzeba walnąć sprzętem, oprzyrządowaniem czy też mówiąc po angielsku, toolingiem. Od kiedy zacząłem brzmieć jak prawilniaki na fejm mma? Coś poszło nie tak z pisaniem tego akapitu. Zacznijmy od nowa.

Formatowanie i analiza statyczna kodu Pythona, czyli jego oprzyrządowanie/tooling, to ważny element podejścia leniwego człowieka do zapewnienia jakości kodu. To rzeczy, które robią i dbają o jakość za nas, żeby łatwiej się pracowało. Pomówimy trochę o nich, ale najpierw o Pipelinach.

13.6.1 Pipeliny

Czym są i dlaczego ich potrzebujemy Automatyzacja rzeczy? Rurociągi na ratunek. O jak to dziwnie brzmi spolszczone. Kiedy chcemy dbać o jakość naszego kodu w Pythonie, zwykle chcemy dbać o takie rzeczy jak formatowanie, spójne wzorce importu, bezpieczeństwo i utrzymywanie naszych standardów na bieżąco. Jeśli chcemy to zrobić w naszym repo/w chmurze automatycznie, możemy użyć pipelineów.

Pipeliny to po prostu zestaw kroków, które składają się na nasz proces CI/CD. Jest to mniej więcej tylko kawałek kodu, który wykonuje za nas pewne kroki. Zazwyczaj pajpy

są definiowane jako plik yaml, który określa jakie kroki/akcje chcemy podjąć w ramach naszego procesu CI/CD, czyli analizowanie, sprawdzanie jakości, formatowanie naszego kodu i budowanie/deployowanie go.

Do najczęściej znanych narzędzi służących do tego w chmurze należą: GitHub Actions, GitLab CI/CD, Bitbucket Pipelines, CircleCI, Azure DevOps. Zazwyczaj są to rzeczy, które odpalają się, gdy np. tworzymy merge/pull request, wpychamy jakiś kod do repo, mergujemy jedną gałąź w drugą. Uruchamiają one różne kontrole, budowanie, testy i co tam jeszcze.

Flow wygląda tak:

Trigger jest odbierany (np. Branch jest pchany do repo) -> pipeline jest odpalany -> różne kontrole/akcje są wykonywane -> na podstawie tego pipeline może się udać lub nie.

Poza tym, że pajpy są tam w chmurze, uważam, że niektóre ich części są również integralną częścią lokalnego rozwoju. Głównie części związane z rzeczami o kontroli jakości/formatowaniu/etc.

13.6.2 Co sprawia, że kod jest dobry?

Obecnie trendem w Pythonie jest dbanie o pewne rzeczy, które choć nie są kluczowe, z czasem przyczyniają się do jakości, czytelności i łatwości utrzymania projektu. Na wysokim poziomie, w mojej opinii, każdy kawałek kodu Pythona zyskałby coś mając:

1. Spójne formatowanie
2. Uporządkowane importy, które są podzielone na sekcje
3. Bezwzględne importy zamiast relatywnych
4. Używanie nowoczesnych standardów, które są zgodne z nowymi konwencjami
5. Brak nieużywanych importów i nieużywanych zmiennych
6. Skanowanie bezpieczeństwa/podatności

13.6.3 Ciemna strona mocy - black

Pora przejść na ciemną stronę mocy.

13.6.3.1 Kilka słów o formatowaniu i blacku

Częściej niż w projektach, które nie są tak zautomatyzowane jakby być mogły i powinny, można znaleźć ludzi w pull requestach kłócących się o to, które formatowanie jest lepsze. Jak zmienić formatowanie? Które jest lepsze? Który jest bardziej zgodny z pep8?

To może być koszmar, który jest tak kontrproduktywny, że to się nawet w korpo nie mieści.

Aby pozbyć się takich problemów i mieć to załatwione za nas, w Pythonie mamy coś takiego jak Black. Black jest formatyzatorem kodu, który, cóż, po prostu formatuje kod za Ciebie. Możesz sprawić, by black automatycznie formatował Twój kod przed jego zacommitowaniem. W ten sposób można uniknąć wszelkiego rodzaju sporów o pep8 i preferencje formatowania kodu przez recenzentów/autorów, co sprawia, że cały projekt ma spójny wzór formatowania, co sprawia, że jest łatwiejszy do odczytania i tak dalej. Im łatwiejszy do odczytania jest kod, tym lepiej. Podejście leniwego człowieka. Jeśli wiesz, czego się spodziewać, nie będziesz zaskoczony. Im mniej na głowie, tym lepiej.

13.6.3.2 ' vs " Jedną z rzeczy wartych uwagi jest fakt, że Python jako język pozwala na użycie zarówno ' jak i " do oznaczania ciągów znaków. Black domyślnie preferuje podwójne cytaty nad pojedynczymi. Dlaczego. Czytelność, użycie pojedynczego cudzysłowu w języku angielskim i konieczność escapeowania go za każdym razem, gdy używamy go wewnątrz naszych łańcuchów, nie jest zbyt fajna. To plus trudniej się pomylić.

I tak dalej. Można się tutaj spierać, ja stoję po stronie podwójnego ciapka, ponieważ IMO jest to lepsze podejście. Czytelność jest królem.

13.6.4 Isort

Czy słyszałeś o sortowaniu importów? To ma sens wbrew pozorom i wcale nie jest fanaberią. Dlaczego powinieneś odpowiednio posortować swoje importy? Im większy projekt, nad którym pracujemy, zwykle tym więcej rzeczy importujemy z innych części kodu.

Z biegiem czasu te importy mogą stać się nieprzyzwoicie duże/liczne. Często tak jest. Isort to coś, co pomaga nam w tym, optymalizując nasze importy, sortując je odpowiednio, alfabetycznie, grupując w sekcjach i tak dalej. Wiem, że to może wyglądać jak drobnostka, ale to właśnie te drobne rzeczy dodają do ogólnej jakości kodu. Googlnij sobie przykład zmian w kodzie po isorcie i zobacz jak to wygląda.

13.6.5 Importy jak wódka - Absolutne

Nowym standardem w Pythonie jest używanie importów absolutnych. Dlaczego tak jest możesz przeczytać na własną rękę. Było wiele debat na ten temat, wynikiem których jest: preferujemy import absolutny kiedy tylko można. Sprawiają one, że jest mniej dwuznaczności i zapewniają wyraźniejsze rozróżnienie, czego naprawdę używamy, z którego pakietu. Mamy również narzędzie do tego celu, którym jest absolufy-imports. To narzędzie jest szczególnie przydatne w przypadku starszych projektów, gdzie możesz potrzebować poprawić importy w wielu plikach, aby dopasować je do nowej konwencji. To narzędzie robi to za ciebie.

To:

```
from .notifications.some_important_file import SomeClass
from .another_important_file import AnotherClass
```

Zostaje zamieniony na to:

```
from em.jobs.notifications.some_important_file import SomeClass
from em.jobs.notifications.another_important_file import AnotherClass
```

13.6.6 Bandit

Statyczna analiza naszego kodu pod kątem potencjalnych uchybień bezpieczeństwa.

13.6.6.1 Dlaczego czasami potrzebujesz bandyty w swoim życiu Kiedy piszemy nasz kod powinniśmy mieć na uwadze bezpieczeństwo. Chyba, że czasem chcesz narazić swoją firmę na potencjalną utratę milionów. Przesadzam z tym przykładem, ale jednak. Bezpieczeństwo jest ważne. W jakiś sposób możemy popełnić błędy proste, z powodu

zapomnienia i zaniedbania, którym można było zapobiec w inny sposób. Aby nam o tym przypomnieć i przestrzec istnieją różne narzędzia, które można wykorzystać.

Wśród nich jest bandit. Bandit jest narzędziem do analizy statycznej, które skanuje Twój kod w poszukiwaniu potencjalnie niebezpiecznych fragmentów kodu i ostrzega Cię o nich. Kiedy uruchomisz bandit na swoim kodzie, otrzymasz raport oraz listę miejsc w kodzie, w których znajdują się potencjalne problemy.

13.6.7 autoflake

Im mniej masz...

13.6.7.1 Odchudzamy kod jak Putin obywateli Czasami tak się zdarza, że w naszym kodzie możemy mieć niewykorzystane deklaracje importu lub niewykorzystane zmienne. Zdarza się to najlepszym. Aby automatycznie się tym zająć, możemy chcieć włączyć do naszych projektów autoflake. Jest to narzędzie, które po prostu się tym zajmuje - usuwa nieużywane importy i zmienne. Nie ma tu żadnej magii.

13.6.8 pyupgrade

Ten kawałek oprogramowania automatycznie aktualizuje niektóre stare wzorce składniowe na nowe. To wszystko.

13.6.9 bumpversion

Jest taka rzecz, którą nazywamy semantycznym wersjonowaniem lub semver. Jest to konwencja, która mówi nam, aby wersjonować nasz kod według następującego wzorca: MAJOR.MINOR.PATCH

Na przykład: v0.2.12

Część major jest zwiększana, gdy przechodzimy do głównych rolloutów, które zmieniają wiele.

Minor jest zwiększany, gdy robimy normalne wydania, np. z większymi funkcjami.

Patch jest czymś, czego używamy dla mniejszych funkcji, łatek, poprawek itp. Ten element rośnie najszybciej.

Abyśmy nie musieli zarządzać tym ręcznie, mamy narzędzie zwane bumpversion. Aktualizuje ono wersję, tworzy commit ze zmianami, tworzy tag git i tak dalej, wszystko automatycznie. Jest to zgrabne narzędzie, które można mieć w swoim CI/CD. Ułatwia to zarządzanie wersjami, tworzenie changelogów, filtrowanie commitów i zauważanie zmian, błędów, wersjonowanie pakietów/api itp. Możesz zobaczyć przykładową historię commitów i użycie bumpversion w historii commitów mojego projektu - braindead Zajrzyj.

Czy uruchamiamy to wszystko ręcznie? Nie. Chcemy być leniwi.

13.6.10 Git hooks & pre-commit

Zautomatyzuj nudne zadania, żyj długo i szczęśliwie.

13.6.10.1 Git hooks i pre-commit Jeśli chcesz, aby wszystko to działo się automatycznie, możesz stworzyć git-hooks, które są uruchamiane np. podczas commitu lub przed commitem. Jednym ze sposobów jest po prostu stworzenie pliku .pre-commit i umieszczenie go w folderze .git i wykorzystanie np. Makefile lub użycie czegoś takiego jak narzędzie pre-commit. Jest to miłe, poręczne narzędzie, które zajmuje się tym za ciebie. Musisz je zainstalować i stworzyć dla niego config, aby powiedzieć mu, które rzeczy ma robić przed commitem. Nie ma tu żadnej magii.

Pozwolę ci wygooglować szczegóły samemu.

13.6.11 Podsumowanie

Black, isort, absolufy-imports, pyupgrade, autoflake, bandit, bumpversion to narzędzia, które nieco ułatwią Ci życie.

Może to dobry pomysł, aby włączyć je do swojego lokalnego przepływu rozwoju i rurociągów?

13.7 Pytania i zadania

1. Napisz krótkie podsumowanie z wnioskami które narzędzie jest do czego i z przykładami użycia, a następnie podeślij wyniki swojej pracy.

14 Bazy danych

Trochę o tym, o czym warto poczytać i o co chodzi z bazami danych.

Co to w ogóle są bazy danych? To po prostu miejsce, gdzie w sposób w miarę trwały chcemy przechowywać jakieś dane.

Bazy danych są różnorakie i rozmaite, dzielą się na różne kategorie. My pomówimy sobie o tych Relacyjnych oraz na języku, do robienia zapytań, który króluje od dziesięcioleci w systemach bazodanowych - SQLu. Są bazy, które z niego nie korzystają, ale to nie w scopie tej książki.

Od razu zaznaczę, że przez to, iż istnieją różne rodzaje baz danych, to czasami powstają gównoburze. Która lepsza etc. Coś jak Linux vs Windows, które distro Linuxa najlepsze, Jabłka czy Śliwki, etc.

Prawda jest taka, że kretynizmem jest wychwalać w każdej sytuacji tylko jedno rozwiązanie. Każda baza ma własne zastosowanie i pewne nisze, do których nadaje się najlepiej. No, może poza OracleDB, która powinna już dawno umrzeć i przestać ludzi katować swoim istnieniem. Anyway.

14.1 SQL

SQL to po prostu język baz danych pozwalający robić zapytania na danych. Tyle musisz wiedzieć. Do tego poczytaj o prostych zapytaniach: SELECT, UPDATE, INSERT, klauzula WHERE i inne.

Przykłady do analizy jeśli nie chce ci się samodzielnie googlować:

```
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, ...);
```

14.2 Relacyjne bazy danych

To generalnie bazy danych, gdzie posiadamy jakąś strukturę formalną, powiedzmy, poprzez zdefiniowane tabele (najczęściej) oraz gdzie między tabelami możemy definiować relacje. Dla ułatwienia roboty dodam małe generalne podsumowanie:

SQL (Structured Query Language) to język zaprojektowany do komunikacji z bazami danych relacyjnych. Pozwala on na wykonywanie operacji takich jak tworzenie, modyfikowanie i zapytania dotyczące danych zapisanych w bazie danych.

Zaletami języka SQL są:

1. Prostota i łatwość użycia - język SQL jest prosty i intuicyjny, a jego składnia jest łatwa do zrozumienia.

2. Wszechstronność - język SQL może być używany do wielu różnych rodzajów baz danych, takich jak MySQL, Oracle, Microsoft SQL Server i wiele innych.
3. Wydajność - SQL jest zoptymalizowany do wykonywania szybkich zapytań na dużych zbiorach danych.
4. Standaryzacja - język SQL jest dość standaryzowany, co oznacza, że różne bazy danych będą miały podobną składnię i zestaw funkcji.

SQL jest często używany w aplikacjach internetowych do zarządzania danymi zapisanymi w bazie danych, takich jak rejestracja użytkowników, przechowywanie produktów w sklepie internetowym i tworzenie raportów.

14.2.1 Tabele

Tabele to po prostu coś, gdzie przechowujemy dane. Tabele mają kolumny i rekordy/wiersze.

Tabela to struktura danych w bazie danych, która przechowuje dane w postaci wierszy i kolumn. Każdy wiersz tabeli reprezentuje pojedynczy rekord danych, a kolumny tabeli określają różne pola, które składają się na ten rekord.

Na przykład, jeśli tworzymy bazę danych pracowników, moglibyśmy utworzyć tabelę zawierającą informacje o pracownikach, takie jak ich imię, nazwisko, stanowisko, wynagrodzenie i datę zatrudnienia. Tabela mogłaby wyglądać mniej więcej tak:

Imię	Nazwisko	Stanowisko	Wynagrodzenie	Data zatrudnienia
Jan	Kowalski	Dyrektor	10000	2020-01-01
Anna	Nowak	Manager	8000	2020-03-01
Marek	Wiśniewski	Inżynier	6000	2020-05-01
Agata	Kwiatkowska	Asystentka	4000	2020-07-01

Tabela może zawierać dowolną ilość rekordów i pol, a dane w niej zapisane mogą być różnych typów, takich jak liczby, tekst, daty itp. Czyli nieco inaczej niż w Pythonie, w SQL definiujemy typy danych i jest to obowiązkowe.

Tabela jest podstawową strukturą danych w bazie danych i jest używana do przechowywania i udostępniania danych w aplikacjach.

Do poczytania:

1. https://www.plukasiewicz.net/SQL/Introduction (trochę formalnie wszystko opisane, ale może być)

14.2.2 Indeksy

Czym są indeksy? Indeksy to takie specjalne tabele wyszukujące, które pozwalają nam przyśpieszyć zwracanie danych. Takie jakby wskaźniki, co jest gdzie, dzięki którym sql wie gdzie przeskoczyć i skąd odczytać daną wartość. Taki jakby spis treści w książce.

Dzięki nim SELECTy, klauzule WHERE są znacznie szybsze. Wolniejsze za to będą operacje dodawania i aktualizowania danych - przy dodawaniu danych oprócz samej tabeli trzeba jeszcze zaktualizować indeks co powoduje dodatkowy narzut.

Indeksy można tworzyć na jednej lub więcej kolumn, można je łączyć.
Kiedy warto zastanowić się dwa razy zanim stworzymy indeks?

- w przypadku małych tabel
- w bardzo często aktualizowanych tabelach, gdzie dane się zmieniają nierzadko
- kiedy mamy bardzo dużo NULLi w jakiejś kolumnie

14.2.3 Relacje

Relacje to opis powiązań pomiędzy parą tabel. Istnieje wtedy, kiedy dwie tabelki są połączone przez klucz podstawowy lub obcy.

Każda relacja jest opisana określonym typem relacji, więzi między tymi dwoma tabelami. Pierwszym przykładem będzie **jeden do wielu** (one to many/foreign key). Przykładem takiego czegoś może być relacja kupujący <-> faktura. Faktura ma tylko jednego kupującego z definicji natomiast jeden kupujący może mieć wiele faktur.

Inny przykład to **wiele do wielu**. Tutaj jako przykład niech posłuży fan artysty. Jeden artysta może mieć wielu fanów. Jeden fan może lubić wielu artystów.

Do poczytania:

1. https://developeronthego.pl/sql-schemat-bazy-danych/
2. https://analityk.edu.pl/relacyjna-baza-danych-o-co-chodzi-z-tymi-relacjami-sql/

14.2.4 Normalizacja

Normalizacja to proces w którym pozbywamy się zbędnych danych oraz ustanawiamy relacje między tabelami.

Pomaga to uniknąć duplikacji danych i bałaganu w nich.

Normalizacja danych to proces optymalizacji struktury bazy danych poprzez rozdzielenie danych na mniejsze, bardziej specjalizowane tabele, z których każda zawiera jedynie niezbędne dane. Celem normalizacji jest zapewnienie, że dane są przechowywane w możliwie najlepszy sposób, tzn. tak, aby uniknąć redundancji i zapewnić integralność danych.

Normalizacja danych jest przeprowadzana w kilku krokach, zwanych stopniami normalizacji. 1NF, 2NF, 3NF.

Normalizacja danych ma kilka korzyści, takich jak:

1. Ułatwienie zarządzania danymi - dzięki mniejszej liczbie tabel i brakowi redundancji dane są łatwiejsze do zarządzania, mniejszy bałagan.
2. Poprawa wydajności - mniejsze tabele są łatwiejsze do przeszukiwania i zapytań, co prowadzi do lepszej wydajności.

Ale o szczegółach później.

14.2.5 Transakcje i współbieżność

Z transakcjami chodzi o to, że czasami chcemy pewne operacje grupować i wykonywać je razem. Np. wyobraź sobie przykład z linku niżej - przelew bankowy.

Co jeśli podczas wykonywania operacji, którą można podzielić na następujące kroki:

1. Pobierz środki z konta A i zaktualizuj stan konta A.
2. Dodaj pobrane środki do konta B i zaktualizuj stan konta B.

... nastąpi odcięcie prądu, błąd dysku, whatever? A no właśnie, klient B nie dostałby hajsu a klient A by go stracił. Niefajnie. Tu wchodzą transakcje całe na biało, pozwalając nam na to, by w sposób trwały, przed zakończeniem danej grupy operacji, zapisać je. Wszystko albo nic.

Jeśli po drodze coś się nie uda to transakcja jest cofana jakby jej nigdy nie było i wszyscy szczęśliwi.

Tutaj warto też poczytać o SELECT FOR UPDATE oraz o Race Condition. Ważne przy współbieżności.

Do poczytania: https://mst.mimuw.edu.pl/lecture.php?lecture=bad&part=Ch7

14.2.6 Subskrypcje/Powiadomienia

Czasami chcemy, żeby o naszych zmianach np. dodanie rekordu do bazy danych, informować zainteresowane strony. Wyobraźmy sobie, że np. mamy notyfikacje w naszej aplikacji i informujemy użytkowników o promocji, o updejcie aplikacji czy coś podobnego.

Istnieją zaawansowane rozwiązania, można po prostu odpytywać API, natomiast jest to średnie rozwiązanie.

Są od tego odpowiednie protokoły i serwisy 3rd party, które ładnie rozwiązują ten problem.

Natomiast najprostszym jest chyba... LISTEN i NOTIFY z Postgresa. Dużo osób nie zdaje sobie sprawę, że postgres posiada wbudowaną obsługę notyfikacji. A tu jednak. Fajny kombajn z tego postgresa, wszystko ogarnia. Anyway.

Do przejrzenia: https://www.cybertec-postgresql.com/en/listen-notify-automatic-client-notification-in-postgresql/

14.2.7 Uprawnienia i bezpieczeństwo

Bazy danych czasem udostępniają coś takiego jak column/row-level security. Co to znaczy? Otóż dzięki temu możemy ustalać sobie, który użytkownik może widzieć jakie kolumny czy jakie rekordy nawet w której tabeli w której bazie/schemacie.

Niektóre DB wymagają od nas instalacji dodatkowych pakietów by to osiągnąć, inne mają to domyślnie, jeszcze inne w ogóle nie udostępniają tak granularnej kontroli. Postgres dla przykładu jednak tak. Fajna sprawa.

Aby skonfigurować np. row-level security, należy dodać reguły dostępu do tabeli, które określają, które wiersze są widoczne dla poszczególnych użytkowników lub grup użytkowników. Na przykład, można utworzyć regułę, która pozwala użytkownikowi o nazwie “john” widzieć tylko te wiersze, w których pole “department” ma wartość “marketing”.

Row-level security jest użytecznym narzędziem do ograniczania dostępu do danych w bazie danych i może być używane w połączeniu z innymi mechanizmami bezpieczeństwa, takimi jak uprawnienia użytkowników i role.

14.2.8 Profilowanie

Zapytania możemy profilować podobnie jak kod Pythona. Są do tego specjalne narzędzia. Jest coś takiego jak `EXPLAIN` i `ANALYZE`, są statystyki, są profilery.

EXPLAIN pozwala nam z wyprzedzeniem zbadać to, co dana baza danych planuje zrobić by wykonać query. Jest to tak zwany 'plan zapytania'. Warto zapamiętać te dwa terminy. W przypadku postgresa `EXPLAIN` generuje plan. `EXPLAIN ANALYZE` zaś dodatkowo go wykonuje, podając rzeczywiste dane i statystyki.

Więcej do poczytania: https://www.cybertec-postgresql.com/en/3-ways-to-detect-slow-queries-in-postgresql/

14.2.9 Kolejność kolumn

Nawet kolejność kolumn np. w postgresie ma znaczenie i wpływ na to, jak szybko wykona się nasze zapytanie. Ogromny wpływ nawet.

Podsumuję tutaj wnioski, jakich oczekiwałbym od juniora. Dokładnego mechanizmu bym nie wymagał, natomiast fajnie by było wiedzieć, że ilość kolumn w tabeli ma wpływ na wydajność, to, którą kolumnę przetwarzamy również ma wpływ, ogromny. Warto zatem zapamiętać, że nie jest pożądane tworzenie ogromnych tabel z setkami kolumn. Czasem trzeba je dzielić i znaleźć kompromis między wygodą a wydajnością.

Do poczytania: https://www.cybertec-postgresql.com/en/column-order-in-postgresql-does-matter/

14.2.10 Podsumowanie

Bazy danych to niezbędny element prawie każdego systemu. Warto kapkę coś tam wiedzieć. Językiem dominującym w świecie bazodanowym jest SQL. Znajomość podstaw tegoż raczej nam nie zaszkodzi a może pomoże.

14.3 Tenanty i co to takiego

Trochę o tenant pattern w Django realizowanego za pomocą django-tenants i postgresa.

(Notka: ten kawałek został napisany kilka lat temu przeze mnie i Dominika Szadego, który był wtedy moim juniorkiem <3)

14.3.1 Perspektywa młodego

Witam! Mam na imię Dominik, jestem junior developerem w thirty3.

Środowisko, które będzie wymagające, ale jest chyba najlepsze dla takiego początkującego jak ja. Miejsce, gdzie jest mentor, który pomoże mi w trudnych chwilach, odpowie na wszystkie moje pytania i wskaże drogę, powie o błędach, które popełniam.

Czy dzięki temu nauka programowania jest łatwiejsza niż wcześniej? Jak najbardziej! Czy czyni ją łatwą? Nie!

Dzisiaj chciałbym napisać kilka słów o moim dotychczasowym doświadczeniu, nowych zadaniach, mentoringu i tak dalej, we wspólnym artykule z moim mentorem - Olafem, i co najważniejsze - o tenancy pattern w architekturze oprogramowania.

Powiedziałbym, że proces uczenia się można podzielić na dwie części:

zrozumienie nowego zagadnienia (technologii, narzędzia itp.), które kończy się tym, że ma się ogólne pojęcie o tym, jak rzeczy są robione, co pozwala budować rzeczy na podstawie przykładu, robić niewielkie modyfikacje istniejących rzeczy i tak dalej;

niekończący się proces doskonalenia, który prowadzi do tego, że jest się w stanie tworzyć złożone rzeczy od podstaw;

Dla mnie bycie młodszym programistą oznacza, że często będę stawał przed problemami, które będą wymagały od mnie wykonania pierwszej części - nauczenia się czegoś nowego, aby je rozwiązać. To jest dokładnie to, jak mógłbym opisać mój pierwszy miesiąc w thirty3 - bycie poza moją strefą komfortu programowania i robienie rzeczy, których nigdy wcześniej nie robiłem. Co jest W PYTE.

14.3.1.1 Pierwsze dni Pierwsze dni w nowej pracy zawsze są trudne i przysięgam, że kiedy konfigurowałem swoje środowisko pracy w poprzednich firmach, zawsze coś szło nie tak - brakowało mi jakichś narzędzi, pakietów, dostawałem dziwne błędy. Na szczęście uruchomienie projektu w thirty3 po raz pierwszy było zupełnie odwrotne.

W tym przypadku difference-makerem było połączenie Dockera i Makefile. Wszystko, co musiałem zrobić, to pobrać dockera (i docker-compose) na mój komputer i postępować zgodnie z cholernym README.md, aby mieć wszystko gotowe i działające. Aplikacja działa. Dokumentacja, praktyki są zdefiniowane, kod jest jasny i łatwy do zrozumienia, testy są. To była bryza.

Przynajmniej do pewnego momentu. Nie minęło wiele czasu, gdy zostałem uderzony przez architekturę multi-tenant. Co to jest. Wyobraź sobie, że masz aplikację używaną przez wiele firm (tenantów). Chciałbyś mieć pewność, że nie dojdzie do przypadkowego wycieku danych między firmami, a jednocześnie, że architektura będzie skalowalna i wydajna.

14.3.1.2 Tenanty na ratunek W thirty3 używaliśmy django-tenants do rozwiązania tego problemu, przynajmniej w kilku przypadkach. Istnieje jedna instancja bazy danych do przechowywania danych dla aplikacji, ale wiele schem - jedna dla każdego tenanta (firmy). Tworzy to logiczną separację między danymi. Zanim przeskoczę do przykładów, jak próbowałem zrozumieć tę koncepcję i początki, więc, miejmy nadzieję, natychmiast zauważysz moje błędy i zrozumiesz, jak to działa. Załóżmy, że mamy bardzo prosty projekt Django, który pozwala firmom tworzyć projekty, nad którymi będą pracować.

Potrzebujemy dwóch django appek:

```
companies
projects
```

Oraz modele zadeklarowane w odpowiednich plikach models.py:

```
# `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()
```

Zachowanie, które chcielibyśmy mieć polega na tym, że każda Firma ma swoje Projekty, które nie są dostępne dla innych firm.

Jak wspomniałem wcześniej każdy najemca ma swój schemat w bazie danych - jakby baza w bazie. Powstaje on wraz z utworzeniem modelu, który dziedziczy po TenantMixin (Company). Rzeczą, która pozwala nam odróżnić lokatorów jest unikalny atrybut schema_name (atrybut TenantMixin), którą musimy podać przy tworzeniu każdego obiektu Company. (Uwaga Olafa: to nie musi być nazwa schematu - może to być ID lub prawie wszystko naprawdę, tak długo jak jest unikalne. Ogólnie schema_name jest tylko metadaną, która pozwala nam wiedzieć gdzie szukać w db).

Poza tym musimy stworzyć specyficzny schemat o nazwie "public", którego celem (O: Właściwie jest on tam w postgresie domyślnie, po prostu tworzymy model w naszej tabeli z tenantami i setupujemy publica jako wspolną schemę) jest przechowywanie globalnych danych nie specyficznych dla danego tenanta/firmy oraz wszystkich schema_name tenantów.

14.3.1.3 Tenanty w Django

Pytanie, które powinniśmy sobie teraz zadać brzmi: skąd Django wie, które dane powinny być przechowywane w schemacie "publicznym", a które w schematach dla konkretnych najemców? Ogarnia się to w pliku settings poprzez ustawienie zmiennych SHARED_APPS i TENANT_APPS. Są to listy aplikacji Django (podobnie jak INSTALLED_APPS). Umieszczenie aplikacji w np. TENANT_APPS będzie oznaczało, że tabele dla Modeli z tej aplikacji będą tworzone w każdym schemacie tenanta. Z drugiej strony, jeśli dodamy naszą aplikację do listy SHARED_APPS, to tak jak w przypadku innych aplikacji gdzie nie występują tenanty, tabele zostaną utworzone w schemacie "publicznym".

```
SHARED_APPS = ["companies", ...]
TENANT_APPS = ["projects", ...]
```

Innym pytaniem jest, w jaki sposób Django wie, na którym schemacie lokatorów należy wykonać akcje? Najemcy są identyfikowani przez URL, np. żądanie URL "tenant.something.com" spowoduje, że nazwa hosta zostanie wyszukana w odpowiedniej tabeli w schemacie "public". Jeśli zostanie znalezione dopasowanie, kontekst schematu jest aktualizowany, co oznacza, że zapytania będą wykonywane w dopasowanym schemacie tenanta. Django-tenants dostarcza kilka narzędzi do ustawiania schematów z perspektywy kodu.

```
with schema_context(schema_name):
    # queries will be performed against the schema "schema_name"
```

lub

```
with tenant_context(tenant_object):
    # queries will pe performed against the schema of tenant_object.
```

Wiedząc to wszystko przyjrzyjmy się poniższym fragmentom kodu, aby zidentyfikować pewne błędy, które popełniłem podczas procesu myślowego.

```
class TenantsTestCase(BaseTenantTestCase):
    def test_tenants_example(self):
        companies = Company.objects.all()
        ...
```

Oczekiwane zachowaniem dla mnie byłoby uzyskać wszystkie firmy, a jednak wynikiem był pusty QuerySet. Ok, może muszę stworzyć jakiegoś tenanta zanim wyświetle, logiczne, spróbujmy.

```
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()
```

Tym razem otrzymałem błąd

`Nie można otrzymać listy tenantów będąc w kontekście tenanta, wejdź w schemat publiczny.`

Zadałem sobie pytanie "Co się dzieje?", Ale udało mi się znaleźć gdzieś użycie schema_context. Więc spróbowałem:

```
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()
```

Świetnie, tym razem nie ma błędu. Tak czy inaczej, zmienna companies to wciąż pusty QuerySet. Ostatnia próba:

```
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()
```

Wreszcie, tym razem otrzymałem QuerySet z dwoma obiektami Company. Ale czekaj, przecież stworzyłem jeden. Czas połączyć to wszystko w całość.

Okazuje się, że kiedy uruchamiamy testy z Django-tenants, tworzony jest nowy tenant, z schema_name "test" i wszystkie zapytania wykonywane są przeciwko temu schematowi, chyba że go przełączymy. (O: Przynajmniej w naszym przypadku, ponieważ używamy przypadku FastTenant -> w przeciwnym razie tenanty są tworzone np. per TestCase, co trwa bo migracje są aplikowane per tenant etc.)

```
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"
```

Teraz pamiętajmy, że Company, który jest naszym obiektem tenanta jest przechowywany w schemacie "public" więc puste QuerySety, które otrzymaliśmy wcześniej były poprawne, ponieważ próbowaliśmy szukać obiektu Company w schematach, które ich nie zawierają. Idąc dalej, tworzenie obiektu Project musi odbywać się w kontekście konkretnego schematu tenanta, ponieważ to właśnie tam przechowywane są jego tabele.

```
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()
```

Dla mnie praca z tenantami rozwiązuje się wokół śledzenia, jak zmienia się kontekst, aby zawsze wiedzieć, jakie zapytania mogę wykonać i jakich efektów się spodziewać.

14.3.2 Perspektywa dziada

Ok. Wystarczy z perspektywy Dominika. Teraz czas na mnie. Dam Ci trochę oglądu z szerszej perspektywy. To, co przeczytałeś do tej pory, to zrozumienie przez Dominika koncepcji tenantów i tego, jak używaliśmy jej w thirty3. Jest mniej więcej poprawne, niektóre rzeczy są zbyt uproszczone, ale ogólna idea jest gdzieś tam taka jak trzeba. Jestem z niego trochę dumny, mi zajęło znacznie więcej czasu, aby zrozumieć pewne rzeczy. Starczy klepania po plecach. Teraz postaram się przekazać wam więcej informacji dotyczących procesu decyzyjnego, który mieliśmy, gdy zaczęliśmy używać tenantów, dlaczego ich używamy i dlaczego wy też możecie chcieć to zrobić.

Zacznijmy.

14.3.2.1 Czym są tenanty? Pierwsza rzecz - tenanty. Czo to jest? To taka koncepcja, używana najczęściej w np. produktach SaaS, że upraszczając nieco, to są to tak jakby Twoi klienci. Ja przynajmniej lubię myśleć w ten sposób. Coś jak wtedy, kiedy zamiast tworzyć

rozwiązanie na miarę dla danej Firmy, z którego korzysta tylko ta firma, to tworzysz generalne rozwiązanie, gdzie firma jest po prostu jakby userem.

Niektóre rzeczy są zdefiniowane globalnie w DB i współdzielone między userami/firmami, inne rzeczy są zdefiniowane i powinny być dostępne tylko dla tej konkretnej Firmy i tak dalej. W normalnym przypadku, gdy tylko ta firma będzie korzystać z aplikacji, nie musisz myśleć o tym dużo. Problem pojawia się, gdy chcesz zglobalizować tę aplikację i mieć wiele firm. Wszystkie z nich mają jakieś prywatne dane, jakieś publiczne. Te dane powinny być oddzielone i niedostępne dla innych firm korzystających z danego SaaSa. Potrzebujesz kolejnej warstwy abstrakcji, która logicznie wiąże lub enkapsuluje te dane firmowe. Aw shiet, here go tenants. Jakie są inne korzyści?

14.3.2.2 Skalowanie SaaSa W miarę upływu czasu i rozwoju naszych aplikacji, gdy zaczynasz zdobywać klientów, którzy nie są twoją najbliższą rodziną ani inwestorami, sprawy zaczynają się komplikować. Prywatność staje się nagle ważna. Naruszenie danych/wycieki są kosztowne. Następnie produkt zaczyna zdobywać trakcję, twoja baza użytkowników rośnie, optymalizacja staje się problemem. Zdarza się to w prawie każdym udanym produkcie.

Dobrze jest pomyśleć o tych problemach i przygotować się na nie, ale tylko tyle ile trzeba, żeby nie przesadzić z inżynierią. W naszym przypadku, w większości przypadków, decydowaliśmy się użyć wzorca Tenant w tym celu, używając schematów DB do realizacji planu. Dzięki temu trudniej jest by dane wszystkich klientów wyciekły czy żeby jeden klient uzyskał wjazdy do wrażliwych danych drugiego klienta, łatwiej skalować nasze aplikację, nie obciążając przy tym zbytnio naszego czasu pracy.

14.3.2.3 Metody skalowania bazy danych Co jest czynnikiem, który najczęściej nas ogranicza, przynajmniej w większości aplikacji? DB. Jakie są sposoby skalowania DB? Poziomy i pionowy. Pionowy oznacza, że masz jedną DB, na którą po prostu rzucasz więcej zasobów - lepszy sprzęt. Takie skalowanie ma swoje granice. Gdy je trafisz, nieważne ile masz pieniędzy - to już koniec. Czy możesz coś z tym zrobić?

Tu z pomocą przychodzi skalowanie poziome, czyli używanie większej ilości maszyn/DB zamiast jednej. Jest to jednak dość podstępne - samo postawienie drugiej bazy obok nic nie da. W grę wchodzą nagle takie rzeczy jak wzorce master-slave, spójność danych, sieć węzłów i tak dalej. Dość złożony temat moim zdaniem.

Oczywiście ten sposób ma również ograniczenia, ale często są one znacznie większe niż ograniczenia pionowo skalowanego systemu. A więc tak - zarządzanie tenantami w produkcie podobnym do SaaS można zrobić na wiele różnych sposobów. Pierwszym z nich jest DB per klient. Tutaj prawdopodobnie mielibyśmy jeden większy DB z rzeczami współdzielonymi globalnie w aplikacji, a następnie mniejsze (lub większe) DB z danymi unikalnymi dla klienta.

Drugim jest Schema per client, czyli jedna baza danych z takimi jakby ‘mini bazkami’ w środku.

Trzeci to niestandardowe uprawnienia i relacje w tabelach, na przykład z danymi wszystkich klientów umieszczonymi w jednym schemacie, jednym db.

Pierwszy jest kosztowny i kłopotliwy w zarządzaniu na mniejszą skalę.

Trzeci często kończy się niechlujnymi tabelami DB, obawami o prywatność i koszmarem permissionów.

Drugi natomiast... Cóż, prawie nie nakłada kosztów na sytuację, w której miałbyś zwykłą architekturę pojedynczego schematu/DB - nie jest to architektura SaaS, ale ułatwia skalowanie i rozdzielanie danych klienta poprzez abstrakcję DB - zamiast mieć wiele DB, które są kłopotliwe w zarządzaniu, używa jednego DB tak, jakby było ich "wiele", przynajmniej w pewnym sensie.

Dlatego zdecydowaliśmy się na to rozwiązanie. I szczerze mówiąc, jesteśmy całkiem zadowoleni z wyników.

Dużym atutem tenantów jest również fakt, że kwerendy dalej pozostają proste. Zapytanie o faktury tylko z danej firmy? Wystarczy ustawić odpowiedni kontekst tenanta i to wszystko.

14.3.2.4 Różne metody zarządzania tenantami Tak zwana ścieżka wyszukiwania, którą ustawiamy dla zapytań Postgresa za pomocą naszego db routera, może być ustawiona na wiele sposobów. Tradycyjny wzorzec tenanta wykorzystuje subdomeny jako sposób identyfikacji tenanta- np. x.myproduct.com będzie wyszukiwał tenanta x. To jeden ze sposobów. Należy pamiętać, aby zabronić użytkownikom rejestrowania tenantów z nazwami często używanych subdomen np.`ftp, mail, static` i tak dalej, w przeciwnym razie może czekać nas niemiła niespodzianka. Pamiętaj również, aby mieć certyfikaty, które mają symbol wieloznaczny dla subdomen - w przeciwnym razie zostaniesz bez SSL dla subdomen swoich najemców, co jest całkiem do bani.

Innym rozwiązaniem jest na przykład umieszczenie go jako części adresu url, ale nie subdomeny. Na przykład:`api.example.com/v1/tenant/someendpoint`. My używaliśmy tego drugiego.

14.3.2.5 Co tenanty nam dały Osiągnęliśmy:

1. zero dodatkowych kosztów zarządzania dodatkową infrą
2. zero dodatkowej infry
3. łatwość skalowania -> zmiana sposobu ustawiania ścieżki wyszukiwania dla db routera i gotowe, skalowanie poziome w trzy minuty
4. dane klientów są od siebie logicznie odseparowane
5. zapytania dalej pozostają proste, tak samo jak permissiony i tabele

W każdym razie. Plus minut tak wyglądają i działają tenanty.

PS: minusy posiadania mnie jako mentora - prawdopodobnie zaczniesz pisać.

14.4 ORM

Co to takiego ORM? A no parę słów o tym jakże przydatnym narzędziu.

14.4.1 Czym jest ORM?

ORM, czyli Object-Relational Mapping, a po naszemu **Mapowanie obiektowo-relacyjne**. Takie narzędzie, które wartości z SQLa, za pomocą odpowiednich 'klas' czy 'mapowań',

"konwertuje" na obiekty w Pythonie, które są łatwiejsze do obsługi tak naprawdę. To zbiór przydatnych metod, ułatwień i abstrakcji.

14.4.2 ORM vs czysty SQL

Różnica między ORM a tradycyjnym podejściem, gdzie samodzielnie piszemy czysty SQL, jest diametralna.

14.4.2.1 Wincyj abstrakcji Pierwszą różnicą będzie fakt, że ORM ukrywa wiele rzeczy przed nami, dostarczając warstwę abstrakcji ponad samym SQLem. Robi to kosztem wydajności i kosztem tego, iż zakłada się, że programista wie, w jaki sposób ORM działa, gdyż ich twórcy podejmują za nas pewne decyzje. Te decyzje wpływają na to, jak ORM działa i jak tworzy zapytania. Czasami ich nieznajomość sprawia, że możemy strzelić sobie w twarz, nieświadomie. Niewiedza momentami potrafi zaboleć. Do tego ta wydajność.

14.4.2.2 Mniejsza wydajność Nie jest to niewiadomo jaki narzut, natomiast przy skomplikowanych zapytaniach prawdopodobnie będzie trzeba się odnieść do napisania czystego SQLa. Może też się okazać, że coś, co w czystym SQLu zajmie minutkę, w ORMie, przez np. nietypowość zagadnienia, może zająć godziny. I vice versa. Tak też bywa.

14.4.2.3 Mimo wszystko warto Niemniej jednak dla większości, zdecydowanej większości, w którą wliczasz się Ty, bo inaczej tej książki byś nie czytał/czytała, ORM będzie znaczącym ułatwieniem. Jednakże, jak to zawsze mam w zwyczaju, muszę zaznaczyć, że warto wiedzieć i znać chociaż podstawy SQLa i wiedzieć, jak ORM działa pod spodem, chociaż tak mega mega pobieżnie.

14.4.2.4 Debugowanie/profilowanie ORM ORM jest dodatkowo trudniejszy w debugowaniu i profilowaniu - query są generowane przez silnik, często potrafią być niezbyt przyjazne jeśli idzie o czytelność, dla człowieka plus nieco nieoptymalne jeśli idzie o wydajność. Ponownie, dla 99% przypadków nie będzie miało to znaczenia, bo zapytanie zwracające listę userów będzie tak samo proste i tu i tu.

14.4.2.5 Serializacja z/do obiektów Pythonowych ORM dostarcza nam również pewną warstwę abstrakcji w postaci mapowania danych z bazy do obiektów Pythonowych. Serializacja danych i też ich walidacja. W przypadku pisania ręcznie zapytań SQL to często twoją odpowiedzialnością staje się sprawdzenie, czy dane są poprawne, ich interpretacja czy też konwersja. Jeśli o ORMa idzie to albo udostępnia on szereg narzędzi, które nam z tym pomagają albo wręcz zajmuje się tym za nas. Coś fajnego.

14.4.3 Podsumowanie

ORM to po prostu forma abstrakcji i uproszczenia interakcji z bazą danych z poziomu danego języka. Ma swoje wady i zalety, trzeba je znać, by podejmować świadomy wybór. W 95% wypadków ułatwi ci życie, natomiast pozostałe 5% chyba jest poza zakresem tej książki przeznaczonej dla junior wannabes.

15 Internet

Tutaj poruszymy sobie różnorakie kwestie związane z siecią, Internetem i całą resztą. Mimo tego, że jako junior-python wannabe najbardziej przejmujemy się samym Pythonem i programowaniem, to należy pamiętać, że kod, który piszemy a potem uruchamiamy, nie działa w jakiejś próżni.

Wszystkie nasze web appki, programy etc. uruchamiane są w jakimś konkretnym środowisku. To środowisko oddziaływuje na sposób egzekucji naszych programów, to jak one działają. Dodatkowo nierzadko musimy z nim w jakiś sposób wchodzić w interakcję, często obustronną. Co to znaczy? A no, że oprócz samego naszego Pythona wypada znać też całą otoczkę gdzie to bangla i tak dalej, gdyż inaczej można sobie czasem strzelić w stopę. Do tego wiedza o środowisku i rzeczach powiązanych, z których pośrednio lub bezpośrednio korzystamy, często nie mając świadomości, czyli po prostu inne komponenty `Systemu` jaki projektujemy, tworzymy czy utrzymujemy, są jego integralną częścią jak i czymś co wpływa na naszą prace i nasz kod.

Przez System rozumiem tu jakiś układ, zestaw elementów. W IT jest to zazwyczaj, dla przykładu, nasza webappka, serwery gdzie jest ona uruchamiana, klient, etc.

Pomówmy zatem o tym całym systemie i środowisku.

15.1 Droga żądania

Większość aplikacji webowych działa w modelu klient - serwer. Klient robi zapytania (requesty) do serwera, serwer zwraca odpowiedź (response).

Jak to się jednak dzieje, że jesteśmy w stanie wysłać to zapytanie. Co się dzieje w chwili, kiedy wpisujesz `grski.pl` w pasku przeglądarki a następnie widzisz mojego bloga?

Otóż sprawa wygląda następująca.

Zakładając, że jesteś połączony z Internetem, twoje zapytanie zostanie przetworzone przez twojego ISP (Internet Service Provider/Dostawca Usług Internetowych), który podbije do czegoś, co nazywa się DNS. DNS to Domain Name Server, czyli taka książka telefoniczna, która zawiera mapowanie domen/adresów jak grski.pl, do lokalizacji (IP) w sieci.

Co to znaczy? Każdy serwer ma swoje mniej lub bardziej unikalne IP. Tak zwany adres IP. To coś jak adres zamieszkania.

IP wygląda tak: **172.16.254.1**. To 32 bitowa liczba, przynajmniej w przypadku standardu IPv4. Coś takiego trudno zapamiętać, prawda? Mi tak. Natomiast `grski.pl` już tak. Stąd powstało takie oto mapowanie i domeny. Domeny ułatwiają zapamiętanie i pozwalają na to, by ułatwić życie użyszkodnikom.

Zarzucę też ciekawostkę. Jeśli IPv4 definiuje adres IP jako 32 bitową liczbę, to pytanie do Ciebie, jakie problem może się tu pojawić w dzisiejszych czasach? Otóż 32 bitowa liczba jest mała jak na dzisiejsze standardy. Pomyślcie ile urządzeń jest połączonych do Internetu, większość z nich ma unikalne adresy publiczne. Matko bosko. Generalnie zbliżamy się do punktu, gdzie nie będzie wolnych unikalnych adresów IP. Przykra sprawa.

Powstało zatem IPv6, które ten problem rozwiązuje. Przykład IPv6: **2001:0db8:85a3:0000:0000:8a**
Tutaj rozmiar to już 128 bitów. 2 do potęgi 128 daje od ciula dużo opcji, starczy nam na trochę tej przestrzeni adresowej.

Dobrze, wróćmy jednak do zapytania.

Mamy obecnie tak: twoja przeglądarka (klient) -> ISP -> DNS -> Serwer.

Następnie na serwerze często znajdują się Load Balancery/Proxy, czyli takie kawałki softwareu, które odpowiednio nakierowują żądania/zapytania do nich trafiające na pasujące zasoby lokalne/usługi/API etc.

Z LB/Proxy żądanie trafia do docelowego serwisu. Następnie otrzymujemy odpowiedź i gotowe.

Pomiędzy może być jeszcze wiele innych usług jak cache, CDN etc, ale przedstawiłem tutaj wersję skróconą. A jak już o CDN mowa, to...

15.2 CDN

Czym jest CDN i dlaczego jest ważny? To w zasadzie dzięki nim możemy bez problemów korzystać z mobilnej sieci, mniej płacić za Internet, to dzięki CDNom nie zacina ci się film ze śmiesznymi kotkami na wykopie, a twój operator może obsłużyć obecną liczbę klientów zamiast np. połowy. Wiem, trochę przesadzam, ale no tak troszkę tylko. Czym zatem jest?

15.2.1 Wprowadzenie

Najpierw jednak garść informacji, by zyskać nieco perspektywy. Żyjemy w czasach, gdzie egzystencja bez Internetu jest praktycznie niemożliwa, a na pewno bardzo niewygodna. Większość dzisiejszych luksusów w jakiś sposób bazuje/korzysta z tego wynalazku. Często jednak nie zdajemy sobie sprawy z tego, jak ogromny ten Internet jest i jak szybko się rozwija, pozwólcie, że wyjaśnię.

W 2018 roku przekroczyliśmy kolejną barierę - to wtedy na Ziemi pękł nowy rekord użyszkodników internetu - 4 miliardy ludzi korzystających z Internetu, czyli to aż 53% populacji. Jest to wzrost niesamowity, biorąc pod uwagę fakt, że jeszcze 4 lata wcześniej użytkowników Internetu było około 2,4 miliarda.

W 2016 dziennie przez Internet przelatywało 44 miliardów GB danych na dzień. Biorąc pod uwagę fakt, że wtedy userów było znacznie mniej, to uwzględniając użytkowników w 2018, daje to nam około 51 miliardów GB dziennie. Oczywiście jest to mocno niedoszacowany wynik, gdyż nie dość, że użytkowników przybywa, to i jeszcze pochłaniają oni coraz więcej danych, ale na potrzeby tego artykułu wystarczy, gdyż jakiś obraz to nam daje. Obecnie mamy 2022 i około 5 miliardów. 62,5% populacji.

Średni użytkownik smartfona zużywa w ciągu miesiąca 2.9 GB transferu mobilnego (dane ze stycznia 2018). To 50% więcej niż rok wcześniej.. Sieć rozwija się w zatrważającym tempie. Dlaczego w zatrważającym? Cóż, o ile chodzi o połączenie kablowe, to aż takiej tragedii nie ma, no bo można dołożyć kabel czy dwa, chociaż to też wszystko skomplikowane i kosztowane, to prawdziwy problem rodzi sieć danych komórkowych, gdyż jest ona mocno ograniczona przez fizykę, a biorąc pod uwagę ciągły rozrost... Cóż, mamy się troszkę o co martwić albo nasza sieć nieco się zapcha.

15.2.2 I wtedy wchodzi CDN, cały na biało

Tak. Sytuację całą łagodzi właśnie CDN. Cóż to takiego? To takie serwery, które cachują najpopularniejsze treści w Internecie, by ogólnie odciążyć sieć, skrócić czasy ładowania

i zapobiec pewnym problemom, poprawić bezpieczeństwo. Serwery te rozrzucone są po całej Ziemi w miejscach strategicznych geograficznie dla sieci.

W sieci mamy dostawców treści. Te treści są różne, tekst, obrazki, filmy, multimedia. Dostawcy treści umieszczają je na swoich stronach, serwerach i jest spoko. W momencie, kiedy chcesz sobie coś w Internecie przeczytać czy obejrzeć, to twoje urządzenie łączy się poprzez Internet z serwerem dostawcy treści i przesyła do Ciebie określoną treść. Wszystko spoko, prawda? No nie.

Problem pojawia się, kiedy tych danych i użytkowników przybywa na całym świecie. Problem wynika z architektury Internetu. Gdy oglądasz odcinek swojego ulubionego serialu, to twój komputer nie łączy się bezpośrednio z serwerem dostawcy, nie. Zanim się to stanie musi się on przejść przez dziesiątki innych urządzeń, które skierują go we właściwe miejsce, tak samo odpowiedź od tego serwera.

Wyobraź sobie, że jesteś w urzędzie i żeby załatwić określoną sprawę potrzebujesz podpisu dziesięciu różnych urzędników a na koniec jeszcze podpis przełożonego z Ameryki, do którego jest długa kolejka. Zabiera to dużo czasu, energii i tak dalej, prawda? Tak. Skomplikowana sprawa ogółem. Rolą CDN'a jest skrócenie tej listy potrzebnych podpisów do jednego urzędnika, który jest akurat w lokalnym urzędzie.

Czyli jak z serwerami - twoje zapytanie zamiast tłuc się do serwera w Azji czy Ameryce i męczyć jeden serwer, spyta najpierw lokalnego gościa, który jest miasto obok. W 99% przypadków on wystarczy. W pozostałym 1% trzeba będzie tarabanić do Ameryki, ale na miejscu sprawę załatwimy szybko, bo dzięki pomocy z obsługą petentów lokalnie, wujek Sam ma do obsługi mniej, przez co kolejka jest znacznie mniejsza.

15.2.3 Szczegóły

Z serwera Dostawcy Treści do CDNów przesyłane i cachowane są pewne dane - jakie? Te, które są najbardziej popularne - to bardzo ważne, by na CDNach utrzymywać głównie te dane, które są najbardziej popularne, gdyż dzięki temu CDNy przejmują większość ruchu, redukując obciążenie sieci osiągając wysoki hit rate.

Jest to skomplikowany proces, bo przecież w różnych regionach popularne są różne treści, a to jak się one zmienią, nie jest banalne do przewidzenia.

Algorytmy, które się tu wykorzystuje do tego, by przewidzieć co i gdzie będzie popularne, to naprawdę bardzo ciekawa sprawa i ważna - przestrzeń i zasoby CDNów są ograniczone, zatem wybór tych treści jest trudny. Niesamowity przykład takiej optymalizacji i przewidywanie tego, co będzie akurat popularne, można zaobserwować na przykładnie Netflixa i tego, jak oni to rozwiązują.

15.2.4 Czym jest hit rate, lifetime?

Użyłem wcześniej wyrażenia hit rate. To termin, który określa jaki procent requestów userów może być przetworzona przez CDN i tylko CDN, a jaka potrzebuje pomocy z serwera Dostawcy Treści. Obecnie niektórzy potrafią tak zoptymalizować swoje serwery, by hitrate do cache wynosił nawet w okolicach 99%. Niesamowite wyniki.

Do tego dochodzi jeszcze określenie czasu, przez który raz wgrane treści mają być dostępne - lifetime - po jego wygaśnięciu cache jest 'usuwany' z serwera i na jego miejsce

wskakują nowe (albo wciąż te same, jeśli dalej są popularne)/uaktualnione dane. Jest on zupełnie różny, zależnie od danych, regionu, samego dostawcy usługi.

15.2.5 Czy CDN to jeden ogromny serwer?

Nie, to często całe klastry rozproszonych geograficznie serwerów. Setki tysiące maszyn, które mają pewną hierarchię i według niej działają. Jak? Mniej więcej taką. Serwer dostawcy treści to CP.

Następnie mamy CD & LCF - to taka centrala można by rzec.

Potem jest CCF a pod nim CDPF. CCF to lokalny urząd, a CDPF to urzędnik.

Domyślnie, kiedy robisz jakiś request danej treści, to ląduje on w CCF'ie, CCF sprawdza sobie, czy to, czego potrzebujesz, jest gdzieś w jego zasobach, czyli na serwerach CDPF, gdzie trzymane są zcachowane treści. Czyli w skrócie sprawdza, czy to, o co prosisz, jest gdzieś 'skopiowane' na lokalnym serwerze.

Jeśli na jednym nie ma, to leci do kolejnego z CDPFów pod swoją kontrolą. Co, jeśli nie znajdzie na żadnym ze swoich CDPFów? Wtedy zgłasza fakt do CD & LCF, który pyta się kolejno pozostałych CCFów.

Jeśli każdy CCF stwierdzi, że tego contentu nie ma na CDPFach pod ich kontrolą? Wtedy CD & LCF robi request do serwera twórcy treści, stamtąd sobie dane pobiera i zachowuje lokalnie. Także oryginalny serwer jest męczony w bardzo niewielkiej ilości przypadków, dzięki czemu sam serwer jak i jego okoliczna sieć jest znacznie odciążona, ruch zostaje rozrzucony po lokalnych i rozproszonych CCFach zamiast być skupiony w jednej lokalizacji.

To między innymi dzięki takim rozwiązaniom (lub podobnym) GitHub z pomocą firmy Akamai, byli w stanie sprostać niedawnemu rekordowemu atakowi DDOS skierowanymi przeciwko tej popularnej platformie, który w szczytowej fazie przybrał rozmiar 1.35 Tbps - prawie półtora Tb na sekundę. Niesamowite. To dzięki temu Wykop jako-tako działa. Dzięki temu Netflix nie zapycha całego Internetu.

15.2.6 Podsumowanie

Wiele rzeczy jest, dzięki którym nasze dni są łatwiejsze, a nawet tego nie wiemy. CDNy były pewnie dla większości z was czymś właśnie takim. Oczywiście w tekście sporo jest uproszczeń, także bear with it.

15.3 Cache

Czym jest cache? Cache to taka jakby baza danych, ale o przeznaczeniu nieco innym. W domyśle cache zachowuje dane na określoną ilość czasu, zazwyczaj dość krótką, relatywnie do bazy danych, które czasami zachowują dane permanentnie.

Cache zatem to taka baza danych o krótkim terminie ważności, w której przechowujemy zapamiętane wyniki komputacji, tych, które są kosztowne zazwyczaj i tylko te, które dotyczą odczytu a nie zapisu do bazy danych na przykład.

Czyli jeśli mamy sobie jakiś widok/funkcję, cokolwiek, która przyjmuje argument, u nas to będzie akurat jakiś request, to dla podobnych albo takich samych, cache zwróci wynik 'z pamięci' zamiast liczyć/pobierać od nowa.

Do poczytania:

1. https://www.techtarget.com/searchstorage/definition/cache
2. https://realpython.com/lru-cache-python/
3. https://realpython.com/python-memcache-efficient-caching/

15.4 Chmura

AWS, Azure, GCP to dostawcy usług chmurowych. Co to znaczy? Czym jest chmura? Chmura to po prostu taka jakby serwerownia, ale u kogoś innego na chacie. Kto inny martwi się pewnymi rzeczami.

Za odpowiednią opłatą dostawcy chmurowi ogarniają za nas pewne rzeczy, udostępniają dodatkowe usługi, zajmują się większością rzeczy.

To często takie ułatwienia, gdzie w zamian za pewien koszt i fakt, iż dostawcy chmurowi czasem decydują za nas w pewnych kwestiach, ciężar opieki nad niektórymi sprawami przerzucamy na zewnętrzną firmę. W dużym skrócie.

AWS to chmura od Amazona, Azure od Microsoftu a GCP od Googla.

Która lepsza? Ta, której używa twój docelowy pracodawca. W gruncie rzeczy są one jednak do siebie podobne, zmieniają się tylko nazwy niektórych usług. Dodatkowo jednak chmura będzie posiadała lepszy toolset do określonych zadań, druga do innych.

IMO jeśli o korpo idzie to najwięcej chyba na Azure/AWS.

Startupy to głównie AWS.

GCP to mieszanka.

Źródło: instytut danych z dupy.

Ja osobiście najczęściej spotykałem się z AWSem, ale to tylko ja. Polecam zapoznać się i pobawić trochę w stawianie różnych usług w chmurze samodzielnie. Każda z nich oferuje programy, gdzie za zarejestrowanie otrzymamy pewien zasób hajsu do przepalenia na nasze zabawy. Wykorzystaj, bo warto. Doda ci to samodzielności i +10 do fejmu jak nauczysz się podstawowych rzeczy z DevOpsowania. A co to to całe devopsowanie? Link numer 4. W skrócie to taki gość od infrastruktury i ogarniania serwerów, czyli miejsca, gdzie nasze aplikacje są uruchamiane.

Anyway.

Do poczytania:

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/ <- mocno dodatkowo. Ansible to soft do automatyzacji pewnych zadań, żeby ręcznie nie klikać.
4. https://devopsiarz.pl/devops/kto-to-jest-devops-engineer/
5. https://en.wikipedia.org/wiki/Infrastructure_as_code
6. https://12factor.net/

15.5 Docker

Cóż takiego to ten Dokier cały? Wszędzie o nim piszą.

Otóż docker to narzędzie służące budowaniu i uruchamianiu kontenerów, konteneryzacji. Taka jakby maszyna wirtualna symulująca komputer w komputerze. Główna różnica leży jednak w tym, że kontenery są o wieeeele bardziej wydajne jeśli idzie o zasoby.

VMki stawiają cały system i emulują wszystko od podstaw. Docker korzysta z gotowych komponentów twojego systemu przez co jest o wiele mniej zasobożerny.

Dzięki kontenerom możemy pobierać gotowe 'obrazy', które zawierają wszystko, czego aplikacja potrzebuje do uruchomienia jak i samą aplikację.

Wyobraź sobie, że musisz ręcznie instalować wszystkie zależności, pakiety etc.

Teraz wyobraź sobie to samo na 50 serwerach, bo akurat musicie skalować aplikację. Docker nam to ułatwia. Raz zbudowany kompletny obraz wymaga jedynie uruchomienia.

Obecnie standardem jest to, że aplikacje się konteneryzuje.

Konteneryzacja aplikacji za pomocą Dockera ma wiele zalet:

1. Niezawodność: Kontenery Docker pozwalają na uruchamianie aplikacji w sposób niezależny od środowiska, co oznacza, że aplikacja działa tak samo na różnych systemach operacyjnych i infrastrukturach.
2. Elastyczność: Kontenery Docker są łatwe do przenoszenia między różnymi środowiskami, co oznacza, że można łatwo rozszerzyć aplikację na nowe platformy lub przenieść ją do chmury.
3. Oszczędność zasobów: Kontenery Docker są lżejsze niż tradycyjne maszyny wirtualne, co oznacza, że można uruchomić więcej aplikacji na tym samym sprzęcie.
4. Szybkość: Kontenery Docker są szybsze od maszyn wirtualnych, ponieważ nie wymagają instalowania systemu operacyjnego i wszystkich potrzebnych bibliotek.
5. Łatwość użytkowania: Docker udostępnia narzędzie do tworzenia, dzielenia się i uruchamiania kontenerów, co ułatwia zarządzanie aplikacjami i ich zależnościami.

Ogólnie rzecz biorąc, konteneryzacja aplikacji za pomocą Dockera pomaga zwiększyć niezawodność, elastyczność, wydajność i łatwość zarządzania aplikacjami, co jest szczególnie przydatne w środowisku chmurowym.

Do poczytania:

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/

15.6 Docker-compose

W telegraficznym skrócie Docker Compose to takie skrypty na sterydach, które pozwalają nam łatwiej zarządzać kilkoma kontenerami. Wyobraź sobie, że mamy w jednej aplikacji bazę danych, broker i workera w dockerze. Milion plików Dockerfile potencjalnie, dużo komend do klepania etc. Bałagan ogółem.

Tutaj przychodzi z pomocą plik docker-compose i narzędzie o tej samej nazwie. Pozwala ono na łatwe zarządzanie wszystkimi kontenerami aplikacji jako jednym całościowym systemem, zamiast zarządzać każdym kontenerem osobno. Docker Compose umożliwia również łatwe rozwiązywanie zależności między kontenerami, takich jak połączenia sieciowe i wymagane pliki danych.

15.7 Docker Hub

Docker Hub to publiczna platforma do przechowywania i udostępniania obrazów Docker. Pozwala ona na łatwe dzielenie się obrazami z innymi osobami i używanie gotowych obrazów gotowych do uruchomienia. Docker Hub jest również integrowany z narzędziem Docker Compose, co umożliwia łatwe uruchamianie wielu kontenerów jednocześnie.

16 Rekrutacja

Wszystko co z rekrutacją związane.

16.1 Jak wygląda proces

Na ogół rekrutacje wyglądają dość różnie, ale składają się zazwyczaj z podobnych klocuszków. Jedne są dość krótkie, inne długie, najpopularniejszym wzorcem, będzie jednak rekrutacja w podanym stylu...

16.1.1 Konkrety

Pierwsza będzie rozmowa telefoniczna/przez Skype. Tutaj mogą już paść pierwsze techniczne zadania/pytania, mające na celu odsiać szeroką grupę kandydatów, którzy kompletnie się do danej pracy nie nadają, ale zazwyczaj na tym etapie często są to jednak proste rzeczy, o które pyta Cię osoba nietechniczna, większość zagadnień, o które możesz zostać zapytany, przynajmniej jak wynika z mojego doświadczenia, będzie dotyczyć twojej osoby i twojego profilu zawodowego, co robiłeś, co chcesz robić, dokąd zmierzasz, czemu zmieniasz firmę.

Wbrew pozorom takie rzeczy też są ważne, bo istotne jest jakim kandydatem jesteś, jaki jest twój charakter i czy będzie się dobrze z tobą współpracować, naprawdę. Co prawda należy zaznaczyć, że twój typ osobowości bardziej liczy się w małych firmach. Chodzi mi o to, że w dużej firmie jest o wiele więcej różnych zespołów, różniących się charakterami, gdzie jest o wiele większa szansa na to, że gdzieś przypasujesz. W małej firmie o to trudniej – musisz pasować lepiej.

Czy to znaczy, że zostaniesz zatrudniony jedynie na podstawie charakteru? Nie. Nie jest to AŻ tak ważne. Zapamiętaj jedno – nie bądź dzbanem. Po prostu.

Czy to znaczy, że powinieneś udawać kogoś, kim nie jesteś? Zdecydowanie nie. Jeśli będziesz udawał na rozmowie, to podczas pracy i tak wyjdzie na jaw, jaki jesteś. Ty się będziesz męczył, pracodawca się będzie męczył. Po co?

Następnie może być już spotkanie na żywo lub zadanie domowe w postaci jakiegoś projektu czy też testu typu Codility.

W spotkaniu na żywo oczywiście należy spodziewać się pytań już bardziej konkretnych. Tutaj najczęściej do rozmowy dołączy już lub ją poprowadzi, osoba ściśle techniczna. Jakich pytań można się spodziewać? Cóż, to zależy.

Oczywiste jest, że nikt nie będzie pytał cię z zakresu rzeczy, których nie ma w twoim CV albo o rzeczy kompletnie niezwiązane z daną pozycją. To raczej jasne. Jeśli jest inaczej, to trochę nieprofesjonalnie ze strony prowadzącego rozmowę.

Co do tych rzeczy, które się w nim znajdują, możesz śmiało założyć, że z każdego słowa umieszczonego w CV ktoś może cię przepytać. Oprócz tego są też pewne pytania z zakresu ogólnej informatyki, algorytmiki czy podstaw baz danych, które często padają, niezależnie od pozycji i firmy.

Naturalnie warto brać pod uwagę daną ofertę, pod którą kandydujemy i profil samej firmy, jeśli jest to firma w dużej mierze zajmująca się tematyką bazodanową, to prostą sprawą jest, że możesz oczekiwać pytań z tej działki.

Mała uwaga, zanim przejdziemy dalej, niektóre zadania są zaprojektowane tak, żebyś nie był ich w stanie rozwiązać, czasami konieczne będzie dopytanie o coś, czasami w ogóle nie będzie to możliwe. Niektórzy rekruterzy używają takiej techniki, czy też jeszcze innej, wprowadzając jakiś element stresujący podczas wykonywania zadania, by sprawdzić, jak kandydat radzi sobie w sytuacjach pod napięciem.

Przykładem czegoś takiego może być próba wmówienia niepoprawności rozwiązania, które w gruncie rzeczy jest dobre, czy też jakieś przeszkodzenie podczas rozwiązywania, które ma cię rozproszyć. Raczej bardzo rzadko coś takiego się stosuje, ale słyszałem o takich praktykach, także bądź świadom, chociaż wątpię, by ktoś użył czegoś takiego w stosunku do Juniora.

Jak nie jesteś pewien czegoś, pytaj, nie wstydź się. Upewnij się, że dobrze zrozumiałeś zadanie, podaj jakiś przykład i zapytaj, czy o to chodzi. To nie jest tak, że za każde pytanie, które zadasz, dostajesz jakieś punktu ujemne czy coś w tym stylu. To pokazuje, że nie boisz się pytać i komunikować. Jak dla mnie to plus.

Listę pytań rekrutacyjnych, z którymi ja się spotkałem, podam ci nieco później.

Potem mogą nastąpić kolejne jeszcze rozmowy lub nie. W każdym razie, po jakimś czasie, jeśli firma ma normalne podejście do kandydata, powinieneś czy też powinnaś, otrzymać jakąś informację zwrotną, czy ci się udało, czy nie, jeśli nie, to nad czym warto by popracować.

W skrócie: typową polską rekrutację, można podzielić na: - pogadanka o tobie, twoich doświadczeniach, - rozmowa techniczna, - opcjonalnie zadanie domowe.

Przy czym zdradzę Ci, że ja obecnie raczej staję się wybredny – jeśli rekruter przysyła mi zadanie, które zajmie mi kilka(naście) godzin, to... Dziękuję za rekrutacje. I nie mówię tutaj tego z poziomu arogancji, nie uważam się, za Bóg wie kogo, absolutnie nie.

Po prostu cenię swój czas, bo te kilka godzin, które spędzę na rozwiązywaniu, minie mi bezpowrotnie. Poza tym w swoim wolnym czasie wolę programować to, na co mam ochotę, a nie jakiś tam nudny projekt rekrutacyjny.

Jednakże ty, na początku, kiedy idziesz do lub szukasz pierwszej pracy czy zwyczajnie masz niedużo doświadczenia – nie możesz wybrzydzać.

Wyjątkiem od tej reguły są zadania od gigantów IT. W takich mi się jednak nie zdarzyło pracować to po pierwsze, po drugie nie słyszałem, by stosowali oni takie praktyki.

16.1.2 Jakie pytania się pojawią?

Jak rekruterzy ustalają pytania? Czy jest jakaś lista, z której wybierają wszyscy? Z mojego doświadczenia oraz z rozmów z innymi ludźmi, to wychodzi na to, że... nie.

Takie listy raczej nie istnieją. Rekruterzy po prostu zadają pytania, które... Ktoś im kiedyś zadał. Mają kilka swoich ulubionych pytań, czasem wymieniają się nimi między sobą i tyle. Zdarzają się też oczywiście jakieś sztampowe przykłady pytań, które trafiają się wszędzie, ale takie znajdziesz w liście pytań, którą zamieściłem w tej książce gdzieś niżej.

Raczej nie ma jakiś wewnątrz firmowych tajemnych list, które zawierają pytania do potencjalnych pracowników. Także tutaj nie wysilaj się niepotrzebnie i ich nie szukaj. Zwyczajnie należy być przygotowanym i tyle. Warto jednak zaznaczyć, że jeśli firma często para się pisaniem REST API to o to pewnie będzie pytać. Ogółem dobrą wskazówką jest przejrzeć profil projektów firmy i zobaczyć z czym mają codziennie do czynienia

i przygotować się pod to. Jeśli firma pracuje w Pythonie to nie będą raczej pytać o technologie związane z Javą.

Co to znaczy być przygotowanym? To nie tylko znać jakieś konkretne zagadnienia, nie. Rozmowy rekrutacyjne są trudne, często bywają stresujące, zwłaszcza jeśli zależy nam na danej posadzie. To wszystko ogromnie wpływa na to, jak będzie ci się pracować nad danym Ci zadaniem. To raz. Dwa, częstym problemem jest też to, że ktoś zna dane zagadnienie, ale nie potrafi go wykorzystać w praktyce.

Samo przeczytanie, nie wiem, Cormena, nie jest wystarczające. To nie są jakieś naukowe badania czy dywagacje, a raczej konkretne problemy, do których będziesz musiał znaleźć rozwiązanie. Zresztą, dziwisz się? Najprawdopodobniej jest tak, że twój rekruter ostatnio miał styczność z taką głęboką teorią, kiedy sam kończył studia i dziś raczej tego nie pamięta, więc siłą rzeczy o to nie zapyta, zapyta za to o konkretne wykorzystanie, konkretny problem.

Także jeszcze raz powtórzę, by być przygotowanym, nie wystarczy wiedzieć, czy znać jakieś zagadnienie, trzeba umieć je naprawdę dobrze wykorzystać w praktyce, gdyż podczas rozmowy, gdzie na pewno będziesz się stresować, trudniej będzie ci wpaść na nowe pomysły, więc naprawdę dobrą radą jest to, by pewne rzeczy były dla ciebie już po prostu naturalne tak samo, jak i tabliczka mnożenia jest. Przynajmniej mam nadzieję, że jest.

16.1.3 Nie tylko wiedza

Pamiętaj też o tym, że programista to nie jest tylko jakaś encyklopedia, czy maszyna do klepania kodu. Programista to też, a w zasadzie przede wszystkim, człowiek, a w życiu ludzi ważna jest komunikacja i inne umiejętności miękkie. Bardzo. Zwłaszcza kiedy startujesz na Juniora. Uważaj na to podczas rozmowy i staraj się być komunikatywnym. Ponawiam apel, nie bądź dzbanem. W skrócie NBD.

Możesz być naprawdę super programistą, ale jeśli kiepsko się z tobą pracuje, nie jesteś jak Peszko i nie robisz atmosfery, to raczej pracodawca nie będzie zadowolony, a tym bardziej twój zespół.

To zaś przełoży się na mniejszą wydajność, gorszą atmosferę w zespole, jak i spadek zadowolenia współpracowników. Lepiej zatrudnić kogoś z kim super się współpracuje, kto ma dobre nastawienie, umie pracować w zespole i ma fajne umiejętności miękkie, bo tych technicznych po prostu się douczy i to szybko jak jest jakiś w miarę ogarnięty. Proste.

O wiele ciężej za to jest kogoś nauczyć tych umiejętności wymienionych wcześniej, technicznych z kolei to żadne problem. Zatem wiedz, że twoje umiejętności to nie wszystko, bardzo ważne jest też nastawienie do problemów i ich rozwiązywania, dopasowanie do zespołu, trzeba też być najzwyczajniej w świecie człowiekiem, unikać bycia aroganckim chamem oraz być miłym i życzliwym, jednak w taki sposób, który nie da innym wejść ci na głowę. Tak po prostu.

Niech nie wydaje ci się, że przesadzam. Sam byłem świadkiem, gdy osoba, będąca najbardziej kompetentną w danej dziedzinie w konkretnym zespole, została zwolniona z racji tego, że praca zespołowa układała się dość słabo.

Komunikatywność to przymus, zwłaszcza w pracy zdalnej, ale... Nie ma co przesadzać w drugą stronę. Nie gadaj za dużo. Każde kolejne zdanie to potencjalna szansa na to, byś wplótł w nie jakąś kompletną głupotę, która może cię skreślić. Wiem, bo mówię z

doświadczenia. Trzeba trochę zdrowego rozsądku mieć i wiedzieć, kiedy warto jednak powiedzieć nieco mniej.

Nie przesadzaj też i bądź profesjonalny. Na tym etapie powiedzmy, że nie zawsze jest miejsce na zbytnie spoufalanie się, więc ty z taką inicjatywą nie wychodź, obecnie, jako kandydat, jesteś w pozycji, powiedzmy, niższego statusu, także oddaj swojemu rozmówcy należyty szacunek i zachowaj się profesjonalnie.

Co innego, jeśli twój rekruter z góry zaproponuje, by atmosfera była luźna, żeby obyło się bez zwracania per pan i tak dalej. Zazwyczaj tak właśnie jest, ale dobrze to wygląda, jeśli to osoba starsza zaproponuje zejście na komunikację nieformalną, aniżeli kiedy wymusisz niejako to ty, kandydat. To taka rada.

Jak już przy tym jesteśmy, to wspomnę może też kilka słów o ubiorze. Otóż ja lubię się kierować jedną zasadą, czyli lepiej ubrać się zbyt elegancko, niż przyjść w klapkach i się ośmieszyć tak naprawdę. Ubiór też o tobie świadczy, podobnie ogółem cały twój wygląd. Nawet jeśli ci się to nie podoba, to niestety, albo stety, nasz wygląd wpływa na to, jak ludzie nas odbierają, niezależnie od tego, czy są owego faktu świadomi, czy nie.

Tutaj ponownie, znów pomoże ci po prostu zdrowy rozsądek, bo to, w co powinieneś się ubrać, zależy często od samego miejsca, gdzie aplikujesz, wiadomo, że nieco inaczej można przyjść ubranym na rozmowę do start-upu czy nowego softwarehousa, a inaczej do ogromnej korporacji.

Ja jednak zazwyczaj ubieram się następująco: czarne spodnie, prosta koszula w stonowanych barwach, jeśli to spotkanie bardziej formalne, to do tego dorzucam marynarkę. Taki raczej dość bezpieczny zestaw, który praktycznie wszędzie będzie odpowiedni. Także pamiętaj też o tym, że mimo iż całym mięchem i najważniejszą rzeczą, są umiejętności, wiedza i to, jaki jesteś, to prezentacja też się liczy. Jak cię widzą, tak cię piszą. Istnieje sporo badań, gdzie wykazano to, iż ludziom, którzy ładnie wyglądają, chętniej przypisujemy pozytywne cechy, natomiast tym, którzy wyglądają byle jak, cechy negatywne.

Zatem wyglądaj jakoś, najlepiej jak normalny człowiek, a nie stereotypowy programista, Mireczek, którego siłą wyciągnięto z piwnicy po dwóch tygodniach maniakalnego grania w Wiedźmina trójeczkę. Wyprostuj plecy, ściągnij łopatki, mów wyraźnie i powoli. Jak podajesz rękę, to podaj rękę jak człowiek, a nie zwiędłą witkę. Mów normalnie, a nie prawie szeptem, tak, że cię nie słychać. No! Bądźże kimś z godnością!

16.1.4 Rezultat

Co, jeśli się nie dostaniesz? Cóż, próbuj dalej, podszkol się. Nie ma co się poddawać. Uwaga, czasem na odpowiedź trzeba trochę poczekać. Jeśli rekruter nie wraca do ciebie z informacją na następny dzień, to nie przejmuj się, to wcale nie oznacza, że się nie dostałeś. Osobiście raz jak się rekrutowałem, to rozmowę odbyłem jakoś na początku czerwca, a samą pracę dopiero dogadaliśmy w połowie sierpnia, więc.

Czasami trzeba tydzień, dwa poczekać, a czasem tej informacji nie będzie w ogóle. Uważam to trochę za buractwo, ale taka niestety jest praktyka i to dość powszechna niestety. Dlaczego coś takiego jest brakiem szacunku?

Otóż jeśli kandydat poświęcił swój czas, przygotował się, wykonał jakieś zadania, przyszedł na rozmowę, to powinien, chociaż dostać jakąś informację o stanie swojej wiedzy, o tym, co zrobił źle.

Nie kosztuje to wiele pracy, ot, po prostu przesłanie informacji, prosty e-mail lub dwie minuty na telefonie, a, zwłaszcza osobie, która dopiero zaczyna, może to pomóc.

Jeśli komuś nie chce się nawet czegoś takiego zrobić to... Cóż, coś o firmie to świadczy i powinieneś się cieszyć, że takimi burakami pracować nie musisz. Z drugiej strony – jeśli to ty pojawiasz się na rozmowie bez jakiegokolwiek przygotowania, olewając rozmówcę, to gratuluję, zignorowałeś NBD.

Dostałeś się, jest oferta? Świetnie. Tylko...

Co z tym okropnym pytaniem, które pada zawsze gdzieś po drodze, czyli ile chciałbyś zarabiać? Jak sobie z tym radzić? O tym w innym rozdziale.

Tak samo, jak o przykładowych pytaniach, na jakie możesz się natknąć.

16.1.5 Poćwicz

Rekrutacje to też umiejętność. Na początku może ci wychodzić średnio. Mimo tego, że w innych warunkach rozwiążesz zadanie, na żywo podczas rekrutacji nagle zjada stres, albo stajesz w miejscu. Zdarza się. Dlatego warto ćwiczyć i przed rekrutacją poprosić kogoś (np. grskiego), żeby przeprowadził z Tobą testową rozmowę. Poważnie.

Rekrutacje to też test tego, jak dobrze umiesz się sprzedać, bez kłamania oczywiście, oraz test twoich umiejętności miękkich, negocjacyjnych. Nie raz zdarzało mi się zarabiać więcej od moich znacznie bardziej doświadczonych kolegów, tylko dlatego, że nie bałem się prosić o podwyżki czy spore kwoty plus wypadałem całkiem znośnie na rekrutacjach, gdyż często brałem w nich udział nawet dla samego ćwiczenia, sam prowadziłem, byłem w nich dobry. Dzięki temu mogłem negocjować fajne warunki nad wyraz często.

16.1.6 Negocjacje

Ludzie generalnie nie lubią negocjować, a szkoda. Szkoda dlatego, że zwyczajnie się to opłaca.

Moim rekordem jest sytuacja, kiedy bonus za zatrudnienie, jaki uzyskałem po prostu grzecznie o niego pytając i wskakując na calla na 5 minut, wyniósł 24 000 zł. 5 minut dodatkowej rozmowy a 24k w kieszeni. Dejmn, stawka godzinowa wychodzi tu niezła.

Podobnie jest z pensją. Jeśli pół godziny negocjacji może przynieść ci 5% większa pensje, załóżmy nawet 300 zł miesięcznie, to daje to 3600 zł rocznie. Już sporo. A przypominam, że to tylko kilka(naście) minut zazwyczaj.

16.2 Studia a rekrutacja

Kiedy ja szukałem pierwszej pracy byłem świeżo upieczonym osiemnastolatkiem, będącym w ostatniej klasie liceum. Zero wykształcenia, doświadczenia, nic. A mimo to się udało. Jak widać można. To tak na wstępie zaznaczam, jeśli powątpiewasz w siebie, by dodać Ci trochę otuchy. Jak grskiemu się udało to tobie też może o ile włożysz w to trochę pracy i będziesz miał/miała szczęście. Zanim przejdziemy do szczegółów dwa słowa o studiach/wykształceniu.

16.2.1 Czy tak zwany papier jest nic niewarty?

Generalnie można odnieść wrażenie, że mam negatywny stosunek i mówię o studiach w sposób pejoratywny. Nic bardziej mylnego. Wydaje mi się, że studia mogą być wspaniałym miejscem, gdzie wiele się nauczysz i poznasz cudownych ludzi. Niemniej jednak, jeśli idziesz tam tylko po papier bo mama kazała, nie masz pojęcia co tam robisz, nic się nie uczysz, no to kufa po co.

Wtedy tak, wtedy to całe przedsięwzięcie nie ma sensu. Lepiej już idź tyrać do jakiejkolwiek roboty bo i tak zero pożytku z tych studiów wyciągniesz męcząc się jedynie.

Jeśli zaś sensownie podchodzisz do tematu swoich studiów to będą one jedynie atutem a nie czymś, co może ci zaszkodzić. Wtedy to fajna okazja.

16.2.2 Czy żałuję nie pójścia na studia?

Życie niejako wybrało za mnie, zostałem trochę zmuszony do szybkiego podjęcia pracy, to fakt. Można zatem zadać sobie pytanie, czy tego żałuję? Nie. Mając 3 lata doświadczenia, kiedy moi rówieśnicy gdzieś tam dopiero zaczynali raczkować, byli stażystami, ja już zarządzałem zespołem, poznałem cudownych ludzi i rozwinąłem znajomości w branży.

Nie oddałbym tego za te dodatkowych kilka lat beztroski.

Zaznaczam jednak, że to tylko mój przypadek. Nie każdy będzie miał tyle szczęścia. Studia to zatem coś co naprawdę warto rozważyć, niemniej nie jest to element konieczny.

Studia to fajny czas by wyrobić sobie znajomości, bo łatwiej się gada z Mańkiem, z którym chodziło się na schodki na piwo, który obecnie jest gdzieś Team Leaderem, a inaczej z świeżo poznanym Tadeuszem, który jest obcą osobą. Nie ma jednej recepty.

16.2.3 Czy brak studiów nas przekreśla?

Często padające pytanie. Czy brak studiów skreśla kandydata? Otóż moi drodzy, jeśli ktoś ocenia cię tylko przez pryzmat twojego wykształcenia a nie Was, jako człowieka, to pytanie czy wy z taką firmą chcecie współpracować? Dla mnie to czerwona flaga od razu.

W całej mojej karierze i obecnie można powiedzieć, że dziesiątkach rekrutacji, tylko raz zdarzyło mi się, że wparował prezes na rekrutację i padło pytanie, cytuję:

> Czemu mielibyśmy pana zatrudnić skoro nie ma pan studiów?

Po rozmowie technicznej już wiedział dlaczego i ostatecznie zaoferowali mi pracę. Grzecznie odmówiłem. Z burakami nie pracuję, natomiast klasę trzeba zachować i szacunek do drugiego człowieka. Jakby tego było mało to rok później pisali znowu, czy może się coś zmieniło :-)

Przewrotny ten los, czasem chce grać ze mną w pokera.

Zatem nie, brak studiów o niczym nie przesądza.

16.3 Jak szukałem pierwszej pracy

Trochę o tym jak te poszukiwania u mnie wyglądały.

16.3.1 Spamer pierwszej wody

Czyli pierwszej klasy. Generalnie moje podejście, z perspektywy czasu niekoniecznie optymalne, było następujące: przewertuj wszelakie ogłoszenia. Cokolwiek związanego z programowaniem i jakkolwiek chociaż z backendem? Pisz do nich. Było to pięć lat temu, rok 2017, zatem nie było aż takiego wysypu ogłoszeń jak teraz. Banieczka nie była aż tak duża.

Pisałem do różnych firm, waliłem mejle, wiadomości. Nawet jak nie szukali albo gdy nie pasowałem jeśli idzie o profil kandydata. Spamowałem en masse. Trwało to tygodniami. Wysłałem przynajmniej z 80 aplikacji do najróżniejszych firm. Dostałem może z 8 czy 9 odpowiedzi. 90% nie rzuciło nawet zwykłym "Oddzwonimy do pana". Nie dziwię im się.

16.3.2 Jakieś rezultaty

Niemniej jednak z tych 8 czy 9 odpowiedzi udało się uzyskać 3 mejle, gdzie zaproszono mnie do rozmowy. W jednym przypadku chodziło o regulara(xD), w drugim o juniora, a w trzecim o staż.

16.3.3 Pierwsze rozmowy

Pierwszą oblałem na etapie testu algorytmicznego, Codilime. 10% mi zbrakło. Trudno, jedziemy dalej.

Druga, czyli ta, gdzie wakat był na staż, poszła jak po maśle. Już oczyma widziałem się w tej robocie.

Zanim nastała data umówionej trzeciej rozmowy, dostałem telefon. Prezes jednej z firm, do których spamowałem, znalazł moją wiadomość w swoim folderze SPAM, ale mimo tego przeczytał. Jakoś tak się złożyło, że miałem w CV coś tam o tym, że umiem te całe pajtony albo, że kiedyś używałem, a akurat im się coś tam zwolniło, potrzebują robochłopa.

To powiedziałem zatem, że będę czymkolwiek chce żebym był, dawaj człowiek mi ten interview. No to dał.

Nie poszła mi ta rozmowa za świetnie. Spodziewałem się pytań o Pythona, a tu nagle zonk bo gość przepytywał mnie głównie z podstaw programowania, struktur danych i takich tam, które wtedy moim forte nie były. Całe szczęście pamiętałem jakieś podstawy, bo robiłem kurs od harvardu - CS50, więc parę zdań udało mi się sklecić. W głowie jednak dalej była chillera utopia, gdyż w moim odbiorze, ta druga rozmowa poszła mi perfekcyjnie.

16.3.4 Zdziwionko

Cóż, okazało się, że ta rozmowa o staż niekoniecznie tak świetnie mi wypadła jak myślałem. Nie oddzwonili. Tutaj trochę posmutniałem, że nici z tego i trzeba będzie znowu spamować, aplikować znaczy.

16.3.5 Zdziwionko dwa

Otóż kto inny oddzwonił. Prezes od wiadomości w SPAMie. Zadzwonił z informacją, że zrobiłem spoko wrażenie, dużo jeszcze jest do poprawy oczywiście, ale całkiem ze mnie

fajny gość więc możemy się dogadać.

Zaoferowali mi pracę zdalną, co dla mnie, będącego jeszcze w liceum, było idealne. Hajsik, jak na tamte czasy, też spoko bo chyba jakoś 20 ziko za godzinę, gdzie na budowie zarabiało się 11 zł, a korepetycji udzielałem za 30 zł. Także tego no.

Biorąc wszystko to pod uwagę nie zastanawiałem się nawet chwili, od razu się zgodziłem i przy okazji poinformowałem tą ostatnią firmę, z którą miałem umówioną rozmowę, że podziękuję.

16.3.6 Okres próbny

Generalnie umowa była taka, że pierwsze dwa miesiące mam okres próbny, jak będą postępy to będzie i przedłużenie. No to co, zakasałem rękawy i cisnąłem z tematem.

Po dwóch miesiącach dostałem podwyżkę i przedłużenie. Udało się.

Skakałem z radości ogółem. Tutaj jeszcze raz, poza tymi podziękowaniami na wstępie, pragnę podziękować Jakubowi Gąsiorskiemu i Maciejowi Mondrzyckiemu. To te dwie osoby w dużej mierze odpowiadają za to, gdzie jestem i kim jestem. Pozwolili mi złapać pierwsze doświadczenie, zainwestowali i podjęli pewne ryzyko, zatrudniając nieznanego typa, dzieciaka z liceum co ledwo 18 skończył, do poważnej firmy. Kudos.

16.3.7 Podsumowanie

Tak wyglądały moje poszukiwania pierwszej pracy. Nie były idealne, teraz zrobiłbym to nieco inaczej, przede wszystkim skupiając się na mniejszej liczbie firm, pisząc bardziej spersonalizowane wiadomości, bardziej bym się przyłożył do portfolio projektów na githubie, do zrobienia ładnego CV oraz lepiej przygotował przed rozmowami rekrutacyjnymi. Tobie to polecam. Wybierz kilka ofert czy firm, które ci się spodobają. Napisz porządne wiadomości/list motywacyjny wręcz i powysyłaj. Ofert i firm szukaj na justjoinit, chyba najlepszy portal z ogłoszeniami.

A jak już o CV mowa....

16.4 CV

Czytelne, ładnie odpicowane CV to połowa sukcesu. Porozmawiajmy sobie zatem o sztuce tworzenia czytelnego CV. Ekspertem w tej dziedzinie nie jestem, ale wydaje mi się, że widziałem już wystarczającą liczbę cefałek by wiedzieć, co jest plus minus ok, a co nie.

16.4.1 Moje ostatnie CV

Żeby nie analizować na sucho, pokażę wam przykładowe CV, w tym wypadku moje, które uważam za całkiem znośne.

16.4.2 Analiza

Z czego składa się CV wyżej?

1. Dane kontaktowe Rzeczy takie jak numer telefonu, adres email czy miasto zamieszkania. Dokładnego adresu nie ma potrzeby podawać, ale czasem warto, by rekruter wiedział plus minus skąd pracujemy.

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 3: Pierwsza strona 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 4: Druga strona cv

2. GitHub Chyba oczywiste. Fajne portfolio na githubie dużo pomaga.
3. Blog Tutaj oczywiście nie każdy takowego posiada, ale ja akurat tak. Uważam, że mam kilka fajnych artykułów, stąd go zalinkowałem. Daje to też jakieś tam poczucie tego, że wykazuje inicjatywę i robię coś 'od siebie'. Ja jak rekrutowałem to zawsze jakiś tam mały plusik za coś takiego dawałem. Dodatkowo posiadanie bloga pozwala rekruterowi przeczytać kawałek Twojego tekstu, poznać cię w jakiś sposób. Zdarzały mi się też rozmowy, gdzie rekruterzy otwarcie przyznali, że niekoniecznie planowali zaprosić mnie na rozmowę, ale po zajrzeniu na bloga zmienili zdanie. Tylko pamiętaj, żaden content jest lepszy niż gówno content wstawiony tylko po to, żeby był. Póki siedzisz cicho to ludzie nie wiedzą, żeś głupi - to powiedzenie ma tu zastosowanie. Zatem przemyśl dwa razy co wstawiasz na bloga i czy ci on potrzebny. Nic na siłę.
4. Krótkie podsumowanie Tutaj w dużym skrócie przedstawiłem siebie, swoje wartości i dotychczasowe doświadczenia. Krótko i zwięźle.
5. Doświadczenie Lista kolejnych prac jakie podejmowałem. Tutaj mogą znaleźć się też staże, udział w projektach różnorakich etc. Najświeższe na górze, najstarsze na dole. Kolejność sortowania. Opisz przykładowe technologie, jakie były wyzwania, jakie akcje podjęto, jaki był rezultat i co wyróżniało tę pracę na tle innych. Takie porady.
6. Umiejętności Czyli słowa kluczowe dla rekruterów. Rekruterzy często są nietechniczni. Z technicznego żargonu mogą nie zrozumieć, a jedyne co robią, to skanują cefałki w poszukiwaniu określonych słów kluczowych. To tutaj jest miejsce, by je umieścić. Ważne jednak jest to, by nie wrzucać tu wszystkiego jak leci. Jeśli napisałeś hello worlda w Javie to nie dodawaj jej do swoich umiejętności. Załóż, że z każdego skilla tu wpisanego będziesz przepytywany. Jak przyłapią Cię na kłamstwie to Game Over.
7. Przykłady projektów Też chyba jasne.
8. Rekomendacje artykułów Jeśli nie masz bloga to do pominięcia. Ja dodałem taką sekcję bo z niektórych tekstów jestem dumny i chciałem je wyeksponować.
9. Referencje Też opcjonalne.
10. Hobby(?) U mnie akurat brak tego, bo miejsca miałem mało z racji dużej liczby wpisów w Doświadczenie, natomiast, jak to często u Juniorów bywa, jak cefałka nieco pusta, to dorzuć coś od siebie o swoich hobby, zainteresowaniach, generalnie tym, jakim człowiekiem jesteś. Na juniora/stażystę i tak bardziej zatrudnia się osobę a nie konkretne umiejętności, bo trzeba zainwestować, więc daj się poznać z dobrej strony.
11. Zgoda na przetwarzanie danych Na samym dole malutkim tekstem jest klauzula, którą niestety trzeba dorzucić do każdego CV, bo inaczej firma nie może przeprocesować takiego dokumentu.

16.4.3 Zdjęcie

For the love of god, don't. To nie rekrutacja na asystentkę prezesa żeby zdjęcia wrzucać. Nie. Nie ma potrzeby. To jakiś dziwny polski wymysł z tymi zdjęciami w CV. Zagramanico tego nie znają.

16.4.4 Podsumowanie

Moje CV wygląda jak wygląda. Jego elementy opisałem. Nie jest to jakaś absolutna lista, ale chyba wskazówki całkiem spoko. CV koniecznie po angielsku. To IT jest, tu po angielskiemu nadajemy. Do tego format schludny i czysty. Co do czcionki jest jedna zasada, byle nie Comic Sans. Chyba tyle.

16.5 Co umiałem idąc do pierwszej pracy - studium przypadku

Dziś o tym, co konkretnie trzeba umieć, żeby zdobyć pierwszą pracę, albo raczej o tym, co ja umiałem, gdy pierwszej pracy szukałem. Tytuł chwytliwy, bo w różnych firmach są różne wymagania co do juniorów i w jednej taki gość jak ja jest stażystą, a w drugiej już juniorem, różnie to bywa. Niemniej jednak jest to tylko moja perspektywa, nie jakiś absolutny wyznacznik, ale te rzeczy niżej pozwoliły mi zostać junior python devem, także ten.

Oprócz tego opiszę też jakieś swoje projekty, które umieściłem w CV.

Czy to, o czym tutaj napiszę, jest konieczne albo faktycznie wystarczające? Nie wiem. Mi na rozmowach wystarczało, ale za dużo ich przecież nie miałem, więc moje doświadczenie jest ograniczone, ale powiedzmy, że na taki jakiś staż się złapać to umiejąc to, co niżej, raczej bez problemu dacie radę.

Wybrany przeze mnie porządek jest raczej losowy, jeśli coś jest moim zdaniem ważne, to wyraźnie to zaznaczam.

16.5.1 Googlowanie

To pierwsza i chyba najważniejsza umiejętność. W końcu szeroko pojęta informatyka to nauka nie o programowaniu, tylko o... przetwarzaniu informacji. Częścią tego procesu jest wyszukiwanie, analizowanie i wykorzystywanie danej informacji potrzebnej, by rozwiązać jakiś problem, czy wdrożyć coś nowego. Mówiąc krótko – googlowanie. Co przez to konkretnie rozumiem?

Umiejętność zwięzłego przepisywania błędów, szukania informacji na odpowiednich stronach. Najlepiej po angielsku – w tym języku znajdziemy oczywiście najwięcej materiałów.

Chodzi tu też przede wszystkim o to, by w razie przestoju, problemu, czy błędu nie rozłożyć rąk i stanąć, tylko coś robić, szukać pomocy, informacji. O to chodzi. 99% problemów jest już gdzieś rozwiązanych, trzeba tylko poszukać. Jeśli trafiasz na ten pozostały 1%, to jesteś raczej na tak zaawansowanym poziomie, że nie mamy o czym rozmawiać.

16.5.2 Angielski

To konieczność. Mówiliśmy o tym w jednym z pierwszych rozdziałów. Mnogość materiałów dostępnych po angielsku, fakt, że to język międzynarodowy, spopularyzowany i wszechobecny, jasno przemawia za tym, że jest on konieczny każdemu programiście. Zaraz pewnie znajdą się osoby, które stwierdzą, że oni żadnych języków obcych nie znają, a pracują. Spoko, tylko gdzie? Co to za firma? Raczej nie znam szanujących się firm zatrudniających osoby bez chociaż podstawowego angielskiego.

Jak bez niego współpracować z zagranicznymi klientami, współpracownikami, którzy nierzadko również nie są z Polski i po polsku nie mówią? Jak czytać dokumentacje, kod?

Plus zagranica najczęściej lepiej płaci.

> Mówta co chceta, ale na koniec dnia hajs się zgadzać musi. 2 List Proroka Stonogiasza do Polaków

Jaki poziom jest potrzebny? Minimalnie taki, by być w stanie przeczytać dokumentację techniczną jakiegoś języka/frameworka. Na początek starczy.

16.5.3 Algorytmy i struktury danych

Tak, wbrew pozorom je też trzeba znać. Niby to 'nudna teoria' i tak dalej, ale... To konieczność. Nie po to, by samemu je implementować od nowa, ale w dwóch celach. Pierwszym jest przejście rekrutacji, gdyż często będziesz z nich odpytywany, drugim jest świadomość, jak pewne rzeczy działają pod spodem. Czyli to, o czym zawsze wspominam, do porzygu aż.

16.5.3.1 Podstawowe algorytmy

1. Insertion Sort – chyba nie muszę tłumaczyć, prosty algorytm sortowania. Podstawa podstaw.
2. Selection Sort – podobnie jak na górze. Bardzo podstawowe, nieefektywne.
3. Bubble sort
4. Merge sort

b) szukania

5. Linear Search – podstawa kompletna
6. Binary Search

To chyba tyle z takich absolutnych podstaw. I teraz uwaga – czy każdy z tych algorytmów znam dokładnie na pamięć, implementowałem po kilka razy i wykorzystywałem własne, zmodyfikowane wersje? Nie. Po prostu wiem jak działają – znam schemat ich pracy, ich wady i zalety, kiedy użyć jakiego. Kiedy nie używać danego algorytmu, złe przypadki i tak dalej. Bo to ważne jest. Czasami diametralnie zmienia prędkość pracy naszego programu.

Jak już przy algorytmach jesteśmy, to warto wspomnieć o czymś takim jak **rekursja**, która jest z kolei czymś, co trzeba znać by poznać Merge sorta czy Binary search.

Następną rzeczą ważną jest **złożoność obliczeniowa**, czy też tak zwane **big O notation**. O co chodzi? Tak po chłopsku to jest to ilość jakiś tam 'ruchów', których nasz program będzie potrzebował do wykonania danej instrukcji dla zestawu danych n-elementowego. Wyróżniamy różne przypadki dla różnych sytuacji – w najlepszym przypadku, w najgorszym i w średnim. Zazwyczaj podaje się najgorszy, bo to pozwala nam stwierdzić, że 'no cóż, gorzej już nie będzie'.

Czyli mówiąc jeszcze prościej, to liczba kroków potrzebnych do zrobienia czegoś i jak ona rośnie, kiedy tego czegoś jest więcej do zrobienia. Są różne złożoności - 1, log n, n, n*log n,* *n*n i tak dalej. O szczegółach rozpisywał się nie będę, szkoda czasu. Generalnie im mniejsza ta złożoność, tym program szybszy.

Oprócz notacji o `krokach` jest też notacja względem np. pamięci/przestrzeni dyskowej. Analogicznie.

16.5.3.2 Struktury danych Tutaj sprawa jest prosta. Struktury danych to rzeczy, które niejednokrotnie mogą nam ułatwić znacząco życie, są to po prostu jakieś dane ustawione w określony sposób. To taki jakby konkretny sposób myślenia o danych. Tu warto trochę zagłębić się w szczegóły.

Jakie konkretnie te struktury danych kojarzyłem? Już mówię.

1. Array – chyba każdy zna, co?
2. Dictionary – też?
3. LinkedList – większość?
4. Stack
5. Heap
6. Set
7. Queue
8. Trees
9. HashTable
10. Trie

Przy czym te trzy ostatnie to raczej tak ekstra, ponad poziom bym rzekł. Co tutaj warto wiedzieć? Jakie są wady/zalety danej struktury, kiedy jakiej używać. Złożoności obliczeniowe czy też pamięciowe dla nich i podstawowych operacji na nich jak szukania, wstawianie, sortowanie. Podobnie jak przy algorytmach.

Znowu zaznaczam, że sam to praktycznie żadnej z tych struktur danych nie implementowałem – jedynie wiem jak w przybliżeniu działa i wygląda większość z nich. Jedyne co implementowałem sam, to Linked List.

W każdym razie. To serio się przydaje. Uwierzcie.

16.5.4 Narzędzia

Trzeba umieć operować narzędziami, które są potrzebne do pracy, koniec i kropka. Co to było w moim przypadku?

IDE – PyCharm jak i Vim. Obecnie korzystam praktycznie jedynie z PyCharma no i teraz jeszcze sublime. Wcześniej był to vim z pluginami, ale... przy większym projekcie dla mnie wygodniejszy po prostu jest PyCharm.

Python – co to virtualenv, pip i te sprawy? Po co to, na co? A pipenv? Pypi?

Virtualbox/WSL – ponownie, warto się troszkę przyuczyć, o co chodzi, bo przydatne narzędzia, zwłaszcza gdy pracujemy przy kilku większych projektach.

16.5.5 Słowniczek

A teraz dorzucę też trochę takich różnych ogólnych pojęć z szeroko rozumianej branży. Czy musicie każde z nich znać i kiedyś korzystać z danego narzędzia i tak dalej? No nie, ale fajnie jest kojarzyć chociaż, daje to takie rozeznanie i kiedy rozmawiamy z innymi ludźmi, może bardziej doświadczonymi, to pewne terminy będą dla nas znajome, chociaż część. Wtedy, zamiast jakiegoś niezrozumiałego żargonu słychać już minimalnie znajomy bełkot :D

1. VPS – w skrócie virtual private server, taki twój własny kawałek jakiegoś serwera można by powiedzieć
2. VPN – virtual private network, taka sieć prywatna można rzec
3. Docker, Vagrant – to pierwsze to tylko nazwę kojarzę, i że to coś z serwerami i deployem appki, a to drugie to już poczytajcie, bo fajne narzędzie
4. Protokół HTTP, HTTPS, SSL, SSH , IMAP, POP3 (!) - tak tylko pobieżnie który idzie do czego i czym się różnią
5. IDE – czyli zintegrowane środowisko programistyczne
6. MVC – model, view, controller – model przetwarzania danych używany w sporej ilości nowoczesnych frameworków
7. API – application programming interface (!)
8. Cookie, Session, Cache – ciasteczka, sesja, cache storage
9. MySql, Postgres, mongodb, oracle, sqlite, sql, nosql, sql injection - (!) bazy danych, konieczność w większości aplikacji
10. Kody http: 2xx, 3xx, 4xx, 5xx, ale w praktyce w sumie starczy: 200, 201, 301, 302, 400, 404, 500
11. AJAX, JS, ecmascript, typescript, react, vue – javascript, te sprawy, wiadomo
12. Klasa, Metoda, Obiektowość, programowanie obiektowe, funkcyjne, OOP (!!!) Generalnie obiektówka ważna sprawa
13. CMS – content managment system
14. DDOS, DOS – takie ataki złych hakerów
15. VCS – version control system, czy też system kontroli wersji, np. GIT
16. Devops – czym się tu zajmujemy
17. DOM – document object model – dla JS'owców konieczność
18. DRY, KISS, YAGNI (!!) - takie 3 zasady, których warto się trzymać podczas tworzenia kodu
19. linux, bash – linux to system operacyjny, bash to język skryptowy tak w uproszczeniu
20. Framework, library – różnice między nimi, czy to to samo?
21. Unit test, functional test, (!!) - testy jednostkowe i funkcjonalne, o nich niżej
22. Hash, crypto – co to takiego ten hash, kryptografia – czym to się w ogóle zajmuje.
23. Machine code, binary, hex – kod maszynowy, binarny, heksowy. Takie różne systemy liczbowe przydatne, bo nimi posługuje się komputer, a nie dziesiętnym.
24. Production, dev, qa, localhost – produkcja, czyli serwer, na którym stoi rzeczywista strona/appka klienta, dev – środowisko developerskie, lokalne, localhostowe, qa/testing – takie do testowania można rzec, zanim poleci na produkcje, często wędruje tam, chociaż jak to mówią najlepsze testy to tylko na produkcji. Tak serio to nie, nie próbujcie tego w domu.

25. Segfault – segmentation fault, czyli co się dzieje, jak trochę nie ogarniesz w niskopoziomowym języku, lub rzadziej, w wysokopoziomowym
26. Bluescreen - chyba wiesz :-)
27. Cieknąca pamięć - googlaj
28. Integer overflow - też
29. Serwer – tam sobie wrzucamy i odpalamy różne appki/strony (w ogromnym uproszczeniu) i one sobie tam żyją
30. Spaghetti code – po prostu kod, który jest tragiczny
31. TDD – test driven development, DDD – domain driven development, ot ciekawostka
32. Firewall - no chyba tłumaczyć nie trzeba
33. Compiler, linker, interpreter, assembly
34. Design pattern, the big4/gang of 4
35. PNG, JPG, GIF, image sprite - formaty plików w dużym uproszczeniu
36. IP, TCP, PORT, Apache, Nginx, gunicorn
37. Bootstrap, jQuery, Material Design, gulp
38. Asynchroniczność

Dużo? W sumie to nie, bo większość ci się powinna obić kiedyś tam o uszy. Ponownie – chodzi o to, żebyś po prostu kojarzył te terminy i wiedział co jest pięć.

Większość z nich wyjaśniona jest tutaj: https://www.hackterms.com/ a jak tam nie? Google. Albo DuckDuckGo, kto co woli.

16.5.6 Języki programowania

Dobra, teraz pora o samych językach pogadać.

W moim przypadku jest to Python. Co konkretnie umiałem?

Po pierwsze zacznijmy od znajomości biblioteki standardowej Pythona. Byłem z nią chociaż pobieżnie znajomy. Co to znaczy? A no, że jak chciałem coś posortować, to nie pisałem sam funkcji, tylko cyk sort(), albo sorted(), zależnie od przypadku i już. Chodzi tu po prostu o takie generalne rozeznanie – tu macie w sumie spis: https://docs.python.org/3/library/index.html Czy wszystko? Absolutnie nie, po prostu coś tam trzeba kojarzyć, nikt nie oczekuje, że znasz to na pamięć. Przewertuj dokumentacje, oficjalna dokumentacja Pythona czy Django jest super.

Na czym się skupić? Podstawowe typy danych, struktury w Pythonie, najczęściej używane funkcje, jak w Pythonie wygląda OOP, co to te generatory, iteratory, list/dict comprehensions, f-strings, odczyt, zapis do plików, otwieranie jakiegoś linka za pomocą pythona, co to GIL i wielowątkowość w pythonie, datetime, collections.

Jak? Po prostu wykorzystując w praktyce – zrób kilka różnorodnych projektów, czytaj kod innych osób, zobacz, jak ktoś rozwiązuje dany problem, porównaj ze swoim pomysłem.

Poza tym miałem styczność z takimi bibliotekami w pythonie jak BeatifulSoup, Selenium.

Do tego jest jeszcze C – ale tylko podstawy, które ogarnąłem na CS50. To tylko w ramach nauki o tym, jak pewne rzeczy wyglądają poziom niżej.

Oprócz tego znam html5/css3 na poziomie pozwalającym mi samodzielnie tworzyć proste szablony/modyfikować te bardziej złożone.

JavaScript – tutaj w sumie to wszystko sobie wygoogluje, ale fajnie jak kiedyś dla testu zrobisz sobie, nie wiem, skrypt zmieniający zawartość diva na stronie, albo coś, żeby zobaczyć jak to działa. Pamiętaj też o tym, że często wystarczy sam JS, nie trzeba pierdyliarda bibliotek.

Przy jQuery to samo co JavaScript – niedużo ogarniam, wszystko sobie googluje w miarę potrzeb. Grunt to umieć zaincludować do htmla samego jquery, reszta google. W 2022 raczej bym nie zawracał sobie nim głowy jednak. API JSa rozrosło się i rozwinęło w porównaniu do 2017, kiedy to ja zaczynałem.

16.5.7 Frameworki

To trochę o tym, z jakimi miałem styczność.

Django, flask, dawno dawno temu web2py, wordpress. Czy w każdym z nich jestem mistrzem? Absolutnie – w żadnym, ale jak wezmę dokumentacje, to coś już tam samodzielnie stworze, przynajmniej w tych dwóch pierwszych.

Na twoim miejscu celowałbym w dwa - Django i FastAPI. Ale to personalna nota, nie jestem fanem Flaska.

16.5.8 Wzorce projektowe

W sumie to pewnie ich używasz, ale nawet o tym nie wiesz i to normalne. Chodzi o to, żebyś kojarzył, iż coś takiego istnieje. Moja wiedza mniej więcej się do tego ograniczała wtedy. Może z nazwy kilka bym wymienił, ty możesz dodatkowo poczytać. Singleton. Factory. Takie tam.

16.5.9 Testowanie

No jest coś takiego. Jak to wygląda? Chodzi o przeklikanie appki? No nie zawsze. Fajnie jest wiedzieć, co to są te testy jednostkowe, testy funkcjonalne, integracyjne i takie tam. Czy musisz być w nich biegły i umieć je pisać? Nie, w zasadzie to większość programistów niezbyt je ogarnia tak w sumie, ale zawsze plus za to i to całkiem duży.

16.5.10 Systemy operacyjne

Jak słyszałeś, że jest coś takiego jak Linux, to spoko. Jak nie, no to też nie najgorzej. Ty już na pewno słyszałeś bo pisałem o nim wcześniej. Szczerze to mocno zależne od twojej pracy, czy będzie ci potrzebne, ale moim zdaniem dobrze jest wiedzieć na początku chociaż jakieś podstawy. Nie każę ci tu instalować od razu Gentoo czy coś, ale jak będziesz umiał poruszać się po systemie za pomocą konsoli, utworzyć, przenieść czy zmienić nazwę pliku, usunąć go, za pomocą linuxowej (albo windowsowej) konsoli, to będzie świetnie. Czyli my jesteśmy bezpieczni po przeczytaniu tej książki.

Jeśli o mnie chodzi, to po prostu korzystałem na co dzień z Linuxa, ale nie jest to konieczne. Z czego konkretnie? Manjaro (głównie), Solus, Ubuntu. Do tego drugiego stworzyłem nawet paczkę z pycharmem-community edition bo w repo nie było z racji, że to nowy projekt. Czy ty też musisz? No nie. Tak jak wyżej napisałem, wystarczy umieć się poruszać, bo nuż widelec przyjdzie ci coś zrobić za pomocą SSH, chociażby na VPSie i co? Lipa. A tak to cyk i umiesz.

Chociaż teraz na przykład w ogóle nie korzystam z linuxa, bo pracuję na macu, poprzednio był to windows z WSL – tak akurat się w pracy złożyło. Tak tylko mówię, żeby raz jeszcze zaznaczyć, że znajomość linuxa nie jest konieczna, ale baaardzo przydatna.

Ale załóżmy, że jesteś ambitny. Co w takim razie? Polecenia: **cd, ls, touch, mv, cp, ps, kill, rm, man, echo, grep*, cat, chown, cat, sed** Do tego jak instalować pakiety w danym distro, jak aktualizować.

To wszystko.

16.5.11 Jak już o systemach mowa

To porozmawiajmy o rzeczach nieco niżej poziomowych. Tak, te okropne, archaiczne rzeczy. Yup. Otóż wyobraźcie sobie, że moje zrozumienie informatyki, programowania i ogólne pojęcie, wzrosło zasadniczo po tym, jak trochę ogarnąłem te właśnie podstawy.

To, jak komputer reprezentuje wszystko w pamięci, jak ta pamięć wygląda, stos i sterta, różnice między nimi, kiedy które używamy. Struktury danych jak wyglądają, czemu arraye indeksujemy od 0, a nie 1, jak wygląda ta linked list od środka, liczenie w binarnym systemie, hexowym.

Co to ten assembler, jak wygląda proces kompilacji, jaka różnica jest między językami kompilowanymi a interpretowanymi, typowanie statycznie a dynamicznie.

Brzmi nudno, ale te podstawy pozwolą wam stać się po prostu lepszymi programistami. Można tego nie umieć i też znaleźć pracę, ale pytanie. Chcemy być szeroko pojętymi specjalistami czy też może po prostu klepaczami kodu od templatek?

Mi ta wiedza, o tym jak to działa od strony niskopoziomowej, przydała się na rekrutacji do obecnej pracy akurat.

Warto też wiedzieć, jak pewne rzeczy zrobić w konsoli – poza IDE. Lepiej wtedy poznajesz mechanizmy działania niektórych narzędzi.

16.5.12 Debugowanie

Jak coś idzie nie tak, to trzeba to naprawić.

Zaprzyjaźnij się chociaż minimalnie z debuggerem, zapoznaj z metodami debugowania.

Bo poza debuggerem można jeszcze robić kapkę inaczej – metody gumowej kaczki, czy też metodą 'XD'. Fajna sprawa. Na czym one polegają?

Ta pierwsza na tym, że tłumaczysz komuś, czemuś, w domyśle gumowej kaczuszce, jak twój program ma działać krok po kroku. W pewnym momencie może zaskoczyć ci, że AHA! Przecież inaczej to napisałem albo porównujesz to też z kodem i patrzysz czy faktycznie tak jest.

A metoda 'XD'? To nic innego jak zwykła metoda wypisywania – dodajesz do kodu linie prinutjące pewne dane, jakąś wiadomość charakterystyczną, żeby łatwo odnaleźć w logach i potem patrzysz, gdzie się sypie. U mnie to zazwyczaj ta charakterystyczna wiadomość to 'XD' bo jestem zwolennikiem tej filozofii, ale nieważne.

Ostatnią opcją jest przechodzenie przez twój kod krok po kroku i wdawanie się w rolę komputera - wykonujesz instrukcje dokładnie tak jak zapisano, nie myśląc za dużo i np. na kartce papieru podstawiając wartości.

Czyli tak. Ogarniamy debugger. Tak zwany odpluskwiacz. Nie, nikt nie używa tego drugiego terminu.

16.5.13 Dobre praktyki

Chodzi tu głównie o organizacje kodu, nazywanie zmiennych i tak dalej. Zmienne powinny być nazywane zwięźle, ale dokładnie, opisowo, konkretnie. Co robi zmienna x? No tego nie wie nikt, ty też zapomnisz, ale jeśli zmienna nazywa się number_of_clients to już coś nam to mówi.

Co do nazw mówimy, to mamy różne metodologie nazywania zmienny, metod, klas. Camel case, underscore i tak dalej. Jaka? Bez znaczenia, byleś trzymał się jednej konsekwentnie. Poczytaj PEP8.

Funkcje, które piszesz nie powinny być też zbyt długie – nie upychaj wszystkiego do jednego wora. Dziel problemy na mniejsze elementy, rozbijaj je i programuj, twórz w sposób pozwalający na ich wielokrotne wykorzystanie bez powtarzania kodu.

Myśl o przypadkach, kiedy user niekoniecznie zachowa się jak oczekujesz. W ogóle zanim zabierzesz się za programowanie to pomyśl. W sumie programowanie to głównie myślenie a dopiero potem pisanie. Przeanalizuj problem, rozważ różne przypadki i dopiero bierz się za implementacje. Pójdzie ci szybciej i stworzysz lepszy kod niż gdybyś bez rozmysłu wziął się po prostu za pisanie.

Kod twórz w miarę prosty i płaski, ładnie podzielony. Skomplikowane dziedziczenie i miliony abstrakcji nie będą twoim przyjacielem.

Dbaj o czytelność kodu, nazw plików, konsystencje.

16.5.14 VCS - o gicie

Co to VCS i po co ci to nie będę się zagłębiał, ale jedno jest oczywiste.

Potrzebujesz znać Git'a. A jak nie znasz? Możesz się gdzieś tam dostać, ale to raczej do takiej firmy, z której trzeba uciekać. To nie jest trudne, naucz się. Git commit, merge, branch, checkout, push, pull, rebase. Chyba więcej na początek nie trzeba, mi to starcza.

16.5.15 Projekty

We flasku zrobiłem dwa, powiedzmy, normalne projekty – jeden nieco bardziej rozbudowany blog (dodawanie wpisów, edycja, komentarze, upvoty, tagi, wyszukiwanie i tak dalej) i prostą emulację najprostszego blockchaina w historii i to w sumie niepełny.

W Django też ze dwa, ale kapkę większe – czytnik mang, dziennik szkolny. Czyli generalnie po prostu CRUD'y.

Do tego skrypt w js+python – lubię wiersze Asnyka, więc pythonem sobie ze-scrapowałem jego zbiór dzieł z neta za pomocą beautifulsoupa, wrzuciłem do .txt w odpowiednim formacie (CSV), a następnie JS'em generowałem z tego losowe wiersze, wystawiając na świat htmlem.

Oprócz tego zrobiłem kiedyś jeszcze prosty układ współrzędnych – wklepujesz jakiś wzór, a skrypt ci rysuje w przeglądarce, funkcjami typu translacja wykresu, odbicie, symetria itd. Ależ pani matematyczka w liceum była zadowolona. Pozdrawiam serdecznie panią Kasię.

Co jeszcze...

Trzy strony na zlecenia – banał. Pobierałem jakiś szablon mniejsza lub większa edycja i cyk, gotowe, czyli klasyczny landing page.

Jedna to typowy lp.

Druga podobnie, ale z integracją galerii z flickera.

Trzecia zaś już kapkę inna – landing page, galeria + jeszcze statycznie generowany jekyllem blog dedykowany dla tej strony.

No i w sumie to jeszcze mój prosty blog, ale to domyślny szablon jekylla lekko zmodyfikowany, więc nie ma o czym pisać, ważny jest jednak fakt, że coś tam miałem napisane – jakieś proste arty, ale zaraz opiszę.

Całość osadziłem na gh-pages w ramach oszczędności na hostingu :D Swoją drogą to dość wygodny sposób do hostowania landing pagów.

Oprócz tego to kiedyś robiłem tam jakieś dwie czy trzy stronki na Wordpressie, ale kiedy to było, nawet nic nie pamiętam, więc do cv nie wpisywałem. Serwerem też się zajmowałem, bo sam hostowałem swoją appkę na vps'ie, czy ustawiałem serwery do CS'a za lat podstawówkowych.

16.5.16 Blog

Mój zbyt bogaty nie był, ale był. Parę artykułów tam jest. To chyba taki plusik, bo pokazuje, że w sumie to jednak coś tam się tym zajmuje, interesuje i tak dalej. Jak ja rekrutowałem i kandydat miał ciekawego bloga to zawsze był plusik.

Co konkretnie na nim było? 6 krótkich artów na tematy techniczne/powiązane i do tego kilka mniej związanych z programowaniem.

A i taki protip: czytaj jakieś blogi/fora branżowe, nowinki, co się w świecie dzieje. Trzeba być cały czas na świeżo z wiadomościami, bo jak nie to wypadniesz z obiegu. Tu w ciągu roku zmienia się tyle, że głowa mała.

16.5.17 GitHub

Koniecznie załóż, i powrzucaj jakieś projekty. Bardzo duży plus jak nie masz doświadczenia komercyjnego i nie wiesz co wpisać w CV - po prostu wypełnij je jakimiś projektami i tyle ;)

Ino jak masz wrzucać crap tylko po to, by go mieć, to może przemyśl.

16.5.18 Podsumowanie

Wyżej zawarłem informacje podsumowujące to, co umiałem, gdy szedłem na rozmowę. Większość z tego możecie nauczyć się dzięki CS50 (C, Python, Linux, algo/struktury i tak dalej) na edexie – szczerze polecam każdemu początkującemu. A co potem? Projekty, projekty i projekty.

Oczywiście notka: powyższy tekst jest pełen uproszczeń doprawdy znacznych, ale co zrobić. O części rzeczy pewnie też zapomniałem, ale trudno. Nie uważam się za żaden autorytet, dlatego też we wpisie wielokrotnie powtarzam, że to tylko moja opinia i opis stanu mojej wiedzy jak szedłem na pierwszą rekrutacje, a nie żaden absolut czy wyznacznik, zatem traktujcie wszystko to, co tu napisane, z pewną dozą dystansu. Sam pracuje od grudnia dopiero, a to, co wyżej to jedynie relacja moich umiejętności, jakie posiadałem idąc na rozmowę.

17 Przykładowe pytania i zadania

To, z jakimi pytaniami się spotkasz, zależy ściśle od tego, co umieścisz w swoim CV, dla przykładu podam ci tutaj listę pytań, z którymi ja się spotkałem na rozmowach rekrutacyjnych. Dodatkowo dorzucę kilka od siebie. Wszystkie, przynajmniej w kilku słowach, omówię.

17.1 Django

1. Distinct w Django model
2. Django middleware - czym jest, gdzie się używa Odp: To kawałek kodu, który na wejściu i przy wyjściu przetwarza naszego requesta w jakiś sposób. Na tym poziomie handlujemy np. autentykacje.
3. Django droga requesta Odp: Middleware -> router -> views -> modele/kod/cokolwiek/ -> znowu middleware(opcjonalnie) -> klient
4. Django select related
5. Jak działają migracje w Django
6. Czy makemigrations potrzebuje połączenia z bazą danych?
   Odp: Nie, nie potrzebuje. Django sobie porównuje modele z ostatnio zapisanym stanem (chyba haszami) i jeśli wykryje zmiany, to zabiera się do działania.
7. Czy migrate potrzebuje połączenia z bazą danych?
   Odp: Tak – wprowadza zmiany do bazy, więc jak najbardziej.
8. Czy możliwym jest zaimplementowanie logowania/autentykacji bez sesji?
   Odp: Jak najbardziej. JWT lub Cookies chociażby.
9. Różnica pomiędzy Flaskiem a Django
   Odp: Wygoogluj sobie, a najlepiej to sprawdź sam i porównaj, bo to banał.
10. Różnica między Django a FastAPI

    Odp: Jak wyżej.

1. Ustawianie unique_together na atrybucie modelu, który posiada `null=True`. Czy django na to pozwala? Jakie jest zagrożenie. Odp: w bazie danych null == null jest False, zatem (null, 1) i np. (null, 1) będą dla bazy danych unikalne. Dla pythona też.

17.2 Python

1. 3 ulubione featury Pythona 3 względem Pythona 2 Odp: Tutaj odpowiedzi oczywiście mocno indywidualne, więc nie podam jakiegoś wzorca, ale ja, o ile pamiętam, wymieniłem: f-stringi, unicode jako domyślne kodowanie, operator przypisania w wyrażeniach (walrus).

2. Co zmieniłbyś w Pythonie – podaj przykład nowego PEP' Odp: Kwestia mocno indywidualna. Ja rzuciłem coś o tym, by wywalić GIL'a. A jak już mówimy o GIL'u, to pora na klasyczne pytanie każdej rekrutacji na Python deva.

3. Co to GIL? Odp: Global Interpreter Lock, a to, czym jest, to już opisywałem, zatem powinieneś wiedzieć. W skrócie powoduje swego rodzaju blokadę, która sprawia, że tylko jeden wątek może się wykonywać w danej instancji interpretera naraz.

4. Wymień rodzaje comprehensions w Pythonie. Odp: Chodzi o np. List Comprehensions, Set Comprehensions, Dict Comprehensions

5. Podaj przykłady powyższych, po jednym na rodzaj. Odp: Przykładami mogą być...

```
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. Jakie są różnice między tuplą(krotką) a listą? Odp: Główna różnica to fakt tego, że jedną z nich można mutować, drugą nie – listę można zmieniać po utworzeniu, tuplę, czy jak kto woli po polsku, krotkę, już nie – po zainicjowaniu już jej nie zmienisz. Poza tym jest duża różnica w wydajności i implementacji 'pod spodem'. Z racji tego, że tupla jest niemutowalna, to interpreter w chwili inicjalizacji ma dokładną świadomość tego, ile będzie ona zajmowała, zatem alokuje dla niej tylko tyle pamięci, ile potrzebuje i ani grama więcej. Lista natomiast podczas inicjalizacji, jest tworzona w taki sposób, że zawsze alokuje się więcej pamięci niż potrzeba na faktyczne dane z listy użytkownika, by potem operacje dodania czegoś do listy działały szybciej. Jeśli tworzysz listę np. 50 elementową, to pod spodem okazuje się, że twoje lista ma zaalokowane miejsce nie na 50 elementów, a na np. 90. Dokładnych liczb możesz dowiedzieć się sam.

7. Czym są generatory i czym się różnią od list? Odp: Generatory to coś, co ułatwia nam operowanie na dużych ilościach danych, gdyż, są to takie swego rodzaju 'funkcje', które zamiast ładować jakiś cały określony data set do pamięci, przetwarza wartości po kolei i yield'uje kolejne wartości. Od listy różnią się tym, że lista ładuje wszystko do pamięci, generator już nie, upraszczając w dużym stopniu.

8. Czy w Pythonie można programować funkcyjnie? Odp: Tak. Jest to możliwe – Python jest językiem umożliwiającym programowanie w wielu paradygmatach takich jak funkcyjny, zorientowany obiektowo, imperatywny, proceduralny. Można by się sprzeczać, czy to programowanie byłoby 'czystym' (raczej nie) funkcyjnie i tak dalej, ale samo w sobie jest to możliwe. Niezbyt popularna opcja i niezbyt promowana. Ogółem niektórzy krzywo patrzą na to połączenie.

9. Enkapsulacja w Pythonie - czy da się? Odp: I tak i nie. Z jednej strony w Pythonie brak takich modyfikatorów dostępu jak private czy public, ale mamy umowne określenia, które powodują, że np. zmienne czy funkcje zaczynające się od

podkreślenia '_', są raczej prywatne i nie powinno się ich używać poza daną klasą. Niemniej jest to tylko konwencja, a nie element języka, nic nie będzie ci broniło używać metody z podkreśleniem wszędzie.

10. Jak rozwiązać problem GIL'a?
Odp: Stackless Python czy też inne implementacje lub wykorzystanie w naszym modelu współbieżności procesów zamiast wątków.

11. Co jest pod spodem dicta, jaka struktura danych teoretyczna? Odp: Hash map.

12. Jak przebiega access do elementów dicta? Odp: Na podstawie klucza liczony jest hash za pomocą określonego algorytmu. Z tego hasha uzyskuje się z kolei offset wskaźnika do referencji.

13. Zachowanie kolejności dodania itemów do słownika w Pythonie, czym? Odp: W Pythonie < 3.7: OrderedDict() z collections. W $>= 3.7$ sam dict() domyślnie zachowuje kolejność.

14. Różnice między listą, a setem. Odp: W liście elementy mogą się powtarzać. W secie nie. Na secie możemy wykonywać pewne operacje, których na listach już nie możemy – podobnie jak na normalnych zbiorach w matematyce – suma, część wspólna, różnica. W najprostszym przypadku można myśleć o secie jak o liście bez powtórzeń. Warto też pamiętać, że dane w secie są nieuporządkowane z założenia, więc set kompletnie nie nadaje się do porządkowania elementów.

15. Jakiego typu obiekty zwracają funkcje dict.keys(), dict.values(), dict.items()? Odp: Dict.items() - oczywiste. Lista tupli. Ale tak nie do końca. Bo jak się bliżej przyjrzeć to jest to klasa dict_items, która nie do końca jest listą – trochę to taka rozszerzona klasa, bo umożliwia nam wykonywanie na niej operacji takich, jak na setach. Podobnie z keys() i values(). Czyli na obiektach zwracanych przez te funkcje można wykonywać operacje sumy, różnicy czy części wspólnej ze zbiorów. Tldr – te funkcje zwracają iterable set-like object.

16. Operatory bitowe w Pythonie. Jakie znasz, do czego służą? Odp: « i » , przesunięcie bitowe w lewo i w prawo & - bitowa koniunkcja, czyli 'i' | - bitowa alternatywa, czyli lub ~ - bitowe negacja, czyli ~x = -x – 1 ^ - bitowa alternatywa wykluczająca

17. Kiedy używać list comprehension, a kiedy normalną pętlą krokową? Odp: Tam, gdzie możemy, powinniśmy używać list comprehensions, gdyż są one czytelniejsze i wydajniejsze, jeśli chodzi o czas wykonania, ponieważ są optymalizowane przez interpreter i lecą z prędkością gdzieś tam w jakimś stopniu zbliżoną do prędkości C, natomiast zwykłe for loopy są wykonywane normalnie przez interpreter z prędkością normalnego kodu Pythona. Jednak jeśli chcemy wykonać coś, co ma efekty uboczne lub skomplikowaną logikę, czasem warto jednak zostać przy zwykłych pętlach.

18. Jakie są różnice między pętlą for i while w Pythonie? Odp: Głównie składniowe oraz wydajnościowe. Pętla while będzie zawsze wykonywana przez interpreter Pythona, natomiast pętla for użyta w list comprehension, chociażby, może zostać zoptymalizowana i śmigać o wiele szybciej.

19. Czym jest iterator a iterable? Różnica iterator a iterable.
20. `float("NaN") == float("NaN")` jaki wynik?
21. `True is True` a tu?
22. `1 is 1` a tu?
23. Napisz program, który sprawdzi, czy podany przez użytkownika ciąg znaków jest palindromem (ciągiem, który czytany od początku i od końca brzmi tak samo).
24. Napisz program, który zliczy, ile razy dana cyfra występuje w podanym przez użytkownika ciągu cyfr.
25. Napisz funkcję, która przyjmie listę liczb i zwróci ich sumę oraz średnią arytmetyczną.
26. Napisz program, który utworzy słownik zawierający statystyki dotyczące liter (ile razy dana litera występuje w tekście).
27. Napisz program, który wczyta plik CSV i wyświetli jego zawartość w formie tabeli.
28. Napisz program, który zaimplementuje algorytm sortowania przez scalanie.
29. Napisz program, który zaimplementuje algorytm sortowania szybkiego (QuickSort).
30. Napisz program, który zaimplementuje algorytm Dijkstry do znajdowania najkrótszych ścieżek w grafie.

17.3 Git

1. Czym różni się merge od rebase? Odp: Merge tworzy nam 'merge commit', natomiast 'rebase' niejako wkleja nam commity z mergowanego brancha. Poza tym, używając rebase, można nieźle zepsuć git loga. Co jest lepsze? Zależy kogo zapytać, są różne szkoły.
2. Różnice między Gitem, a GitHubem. Odp: Git to narzędzie – system kontroli wersji, natomiast GitHub to serwis, na którym możemy hostować swoje repozytoria Git'a i ogółem współpracować z innymi.
3. Co robi git stash? A git stash pop?

17.4 Http/Rest

1. Czym jest REST? Odp: googlaj
2. Co odpowiada za to, jakiego typu treści można przesyłać na/z serwera? Odp: Nagłówki, konkretniej chodzi o content-type.
3. Gdzie lecą parametry w requeście? Odp: W urlu. ? i &.
4. 403 vs 401 Odp: Forbidden vs unauthorized. Różnica kiedy user jest niezalogowany czyli nie ma dostępu vs jest zalogowany ale nie ma pozwolenia.
5. Kody responsow http 1xx 2xx 3xx 4xx 5xx Odp: Zajrzyjcie na wiki.
6. Czym jest jwt? Odp: JWT to po prostu sposób autentykacji, gdzie mamy określony trzy częściowy token wygenerowany za pomocą jakiegoś sekretnego klucza/certyfikatu, który zawiera określony payload.

7. Czy możliwa jest już autentykacja na poziomie load balancera tak, żeby później aplikacja nie musiała tego robić dodatkowo? Odp: Jeszcze jak. JWT ftw.
8. Sposoby autentykacji requesta: basic auth i tokena - opisz i scharakteryzuj czym się różnią. Odp: Googlaj
9. Skąd serwer bierze username i password przy basic auth? Odp: Nagłówek. Jaki?
10. Skąd przeglądarka wie, że daną odpowiedź trzeba wyświetlić jako json a nie np. plaintext? Odp: Znowu - CONTENT-TYPE
11. Różnica miedzy PUT a PATCH. Odp: Jedno wymaga podanie wszystkich pól serializera, drugie nie. Dlaczego? Która metoda wymaga wszystkich pól, która metoda zaś nie?
12. Dostępne metody/verby HTTP. Odp: Googlaj.

17.5 Bazy danych

1. Czym jest join w sql, jak działa, przykład
2. Normalizacja bazy danych - co to, z czym się to je.
3. Rodzaje baz danych. Relacyjne i nierelacyjne, trochę o nich.

17.6 Ogólne koncepty programowania

1. Czym jest OOP.
2. Czym jest SOLID - wymień chociaż jedną
3. Załóżmy, że wpisuję coś w okno przeglądarki i klikam enter by przejść na daną stronę. Co się wtedy dzieje pod spodem?

17.7 Struktury danych

1. Czym jest drzewo/tree?
2. Zbalansowane drzewo binarne?
3. Jak działa hashmapa/słownik?
4. A zbiór?

17.8 Kamień papier nożyce

1. Napisz funkcję rock_paper_scissors(), która zasymuluje losowy przebieg gry w kamień papier nożyce i następnie wyświetli, który gracz, p1 czy p2, wygrał daną rundę. Czy zawsze ktoś wygrywa? Może być remis? Spróbuj uczynić swoje rozwiązanie jak najkrótszym. Spróbuj nie korzystać w ogóle z ifów. Tak, da się to zrobić bez tego. Mnie udało się napisać samą funkcję symulującą pojedynczy losowy przebieg gry w 5 linijkach kodu łącznie, bez ifa.

2. Teraz dopisz kod, która wywoła taką symulacje 10 razy i wyprintuje jej wynik na wyjście.

Przykład mojego rozwiązania:

```
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())
```

17.9 Operacja na liczbach

Tutaj dwa ćwiczonka do zrobienia.

1. Jak można zaimplementować w Pythonie mnożenie przez 2, bez użycia operatora mnożenia, potęgowania, funkcji pow, sqrt?
2. Co z dzieleniem przez dwa biorąc pod uwagę podobne obostrzenia, jak wyżej?
3. Napisz funkcję is_prime(number), sprawdzającą czy podana liczba jest pierwsza.
4. To co wyżej, ale w jednej linijce kodu.

17.10 Statystyki z logów

Tutaj polecimy po angielsku. Treść zadania jak i potencjalne rozwiązanie niżej. Całość można jeszcze uprościć i zrefaktorować, natomiast podaję tutaj rozwiązanie, które naskrobałem na szybko podczas rekrutacji na żywo. 30 minut.

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})
```

Zadanie bezczelnie podwędzone z rekrutacji.

17.11 Statystyki zapytań

Stwórz klasę, która będzie przechowywać całkowity czas egzekucji danego zapytania. Jako dane wejściowe dostaniesz rekordy, które będą zawierały id zapytania oraz czas trwania. Rekordy mogą być częściowe, czyli jedno id może mieć wiele rekordów. Powinno się takie wartości sumować razem.

Dodatkowo zaimplementuj w tejże klasie metodę `get_top_k_records`, która zwróci zadaną liczbę zapytań o najdłuższym czasie egzekucji.

```
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))
```

Zadanie bezczelnie podwędzone z rekrutacji.

17.12 Książka zleceń

Zaimplementuj Książkę Zleceń (ang. Order Book), która obsługuje zlecenia zwykłe, z Limitem Ceny (ang. Limit Order).

Następnie dodaj obsługę zleceń typu Góra Lodowa (ang. Iceberg Order). Co to jest i z czym to się je poczytać możecie w dokumentacjach technicznych giełd. Dla przykładu: SETSmm and Iceberg Orders, SERVICE &TECHNICAL DESCRIPTION z London Stock Exchange. Albo pogooglajcie.

Dane wczytujemy ze standardowe wejścia kolejne zlecenia i na bieżąco aktualizuje książkę. Jeśli zachodzi transakcja to wypisujemy ją na standardowe wyjście. Po dodaniu każdego zlecenia wypisujemy zaktualizowaną książkę na standardowe wyjście.

17.12.1 Format danych wejściowych

JSON. Każde zlecenie ma pole "type" oraz "order". Type zawiera typ zlecenia - Icebergo lub Limit. Order to dane o zleceniu.

- "direction" typ zlecenia ("Buy" albo "Sell"),
- "id" unikalny identyfikator zlecenia (dodatnia liczba całkowita),
- "price" cena (dodatnia liczba całkowita)
- "quantity" wielkość zlecenia (dodatnia liczba całkowita).

Zlecenia typu Góra Lodowa mają dodatkowe pole - `peak`, oznacza on wierzchołek tegoż zlecenia.

Zakładamy, ze dane wejściowe są poprawne.

```
{"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}}
```

17.12.2 Format danych wyjściowych

Jak już wczytamy informacje o nowym zleceniu i będziemy chcieli wypisać zaktualizowany stan książki zleceń, to obiekt JSON, jaki powinniśmy wyprintować powinien mieć następujące właściwości/atrybuty: `buyOrders` oraz `sellOrders`. Każde zlecenie powinno zawierać pola "id", "price" oraz "quantity". Posortowane według ceny. Zlecenia kupna nierosnąco. Sprzedaży niemalejąco. Jeśli cena taka sama to priorytet bierze czas dodania zlecenia. Przykład:

```
{"buyOrders": [{"id": 2, "price": 15, "quantity": 20},
               {"id": 1, "price": 14, "quantity": 20}],
"sellOrders": [{"id": 3, "price": 16, "quantity": 15}]}
```

Przykładowa sesja (sory za brzydkie formatowanie, skopiuj sobie i popraw samodzielnie):

```
{"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}
```

Inny przykład:

```
{"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}}
daje nam
{"sellOrders": [{"id": 3, "price": 100, quantity: 100},
                {"id": 2, "price": 100, quantity:
100}], "buyOrders": []}
```

Przykładowe rozwiązanie na moim githubie - github.com/grski
Zadanie bezczelnie podwędzone z rekrutacji.
Nie umieszczam tutaj gdyż jest ono za długie.

17.13 Skracacz adresów

W tym zadaniu stwórz proszę serwis skracający URLe. Użyj Pythona i dowolnego frameworka. Np. django czy fastapi.

Czyli serwis, który skraca urle np. z https://codesubmit.io/library/react zrobi http://short.est/GeAi9

Analogicznie odwrotne działanie też zachodzi.

- Język: **Python**
- Framework: **dowolny**
- Endpointy:
  - /encode - pełny URL koduje na skrócony
  - /decode - skrócony odkodowuje na pełny.
- Oba zwracają JSON
- To jak bedzie odbywało się kodowanie to detal implementacyjny. Wybierz jak uważasz. Byle dało się zakodować, a potem odkodować. Nie musisz doklejać bazy danych - możemy trzymać w pamięci.
- Stwórz dokumentację jak uruchomić i używać.
- Napisz testy.
- Korzystaj z GITa i za pomocą historii commitów pokaż swoje rozumowanie
- Kod pisz tak, jakby miał iść na produkcję. Czyściutki, elegancki, piękniutki.

Przykładowe rozwiązanie na moim githubie - github.com/grski
Zadanie bezczelnie podwędzone z rekrutacji.
Nie umieszczam tutaj gdyż jest ono za długie.

17.14 Generator statycznych stron

Co to znaczy? Zerknij w googla pod hasłem `SSG` albo `static site generator`. Generalnie to kawałek kodu, program, który generuje na jakiejś podstawie statyczne strony, coś przeciwnego do obecnego podejścia, czyli tworzenia SPA. W moim wypadku postanowiłem stworzyć bardzo prosty system na potrzeby własnego bloga. Tutaj detale implementacyjne zostawiam Tobie. Niech to będzie część zadania - spisz wymagania, rozpisz funkcjonalności etc. Niech poniesie cię wyobraźnia. Minimum jakiego oczekuje, to dostarczenie funkcjonalności wystarczających do prowadzenia bloga - index z listą wpisów i pojedyncze wpisy.

```
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)
```

W kodzie są komentarze, zatem dodatkowo opisywał raczej nie będę. Kolejny kawałek kodu:

```
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()}
```

Kolejny 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
        )
```

Składając wszystko do kupy:

```
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
    )
```

Szerzej rozpisane jest to we wpisie na moim blogu, a całość kodu źródłowego na githubie - braindead. Tam też jest historia commitów, opis etc. Zerknij.

17.15 Jednolinijkowiec

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.

Odpowiedź:

```
islice(accumulate(cycle("Close"), func=lambda x, y: x + ord(y), initial=0), 1, None)
```

Taka łamigłówka.
Główne

18 Studium przypadku różnych aplikacji

Tutaj porozmawiamy i przeanalizujemy różne przypadki konkretnych implementacji, pogadamy o decyzjach i o System Designie, dlaczego coś jest tak a nie inaczej, etc.

18.1 Skracacz linków

Zadanie polegało na zaprojektowaniu systemu skracającego linki, przy założeniu, że będziemy obsługiwać dość spory ruch. Nie rozpiszę krok po kroku, zamiast tego wrzucam tutaj notatki z rozmowy, krótki opis oraz pytania, jakie zadawałem.

1. Czy uwzględniamy autentykacje, czy też może będzie to załatwione poza serwisem? Jaka jest potrzeba biznesowa? `A jak uważasz?` Dla getów odkodowujących skrócony URL auth nie jest potrzebny. Dla tworzenia może się to przydać by móc przypisać usera do linka, jaki skrócił. Pozwoli to na edycję w przyszłości czy zarządzanie skróconymi URLami. `Zatem uwzględniamy` W tym wypadku zatem użyłbym JWT auth, po prostu z racji preferencji plus możliwości późniejszego przepchnięcia autha już np. na poziom Load Balancera, kiedy będziemy skalować.
2. Jaki będzie stosunek writeów do readów? Czyli ile getów przypada na każdy utworzony URL. `Załóżmy, że na wiele wiele readów przypadnie jeden write.` Zatem zwykła baza danych typu postgres. W razie potrzeby możemy dorzucić batchowe tworzenie readów z pośrednim zapisem do cache oraz skalować bazę danych najpierw wertykalnie później utworzyć klaster z write-only masterem i kilkoma read-only replikami/slave'ami.

```
/ - 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 filds 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 shorten 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 its not, then query the database for that qiven 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
Microsver cache <- redis
Persistent storage <- RDS with Postgres
```

18.2 Jak jedna cyferka może zepsuć aplikację - studium

Ostatnio w pracy trafił mi się, moim zdaniem, dość ciekawy przypadek do zbadania. Otóż otrzymaliśmy od klienta zgłoszenie, że przestały działać pliki dla wybranych zasobów - klienci nie mogli uzyskać dostępu do miniatur, informacji, czasami były problemy z samym pobieraniem i tak dalej, ogółem mejdżor fakap, trzeba to naprawić asap.

Cóż, siadłem zatem do tego.

Moje pierwsze podejrzenie padło na mnie samego. Dlaczego? Otóż kilka miesięcy temu, wyraźnie pamiętam, że majstrowałem coś przy tym, jak pliki są zwracane i tak dalej, w każdym razie, byłem praktycznie pewny, że to moja sprawka.

Wyglądałem troszkę tak, jak Nawałka po mundialu.

18.2.1 Uff

Niemniej jednak, po przeanalizowaniu zgłoszeń, okazało się, że problem zaczął pojawiać się nawet, zanim ja cokolwiek w tym konkretnym module zmodyfikowałem, zatem musiałem wykluczyć możliwość, że była to moja sprawka.

Hmm no dobra, co teraz. No nic, ruszmy dalej, może coś się ciekawego okaże.

Przeglądając kod, nie znalazłem niczego ciekawego, żadnych zmian, nic. Cóż zatem mogło spowodować występowanie problemu i to nagle?

Cóż, by znaleźć odpowiedź na to pytanie, musiałem przyjrzeć się temu, jak serwowane są pliki w aplikacji klienta.

Otóż hostowane są one na zewnętrznych serwerach, zależnie od regionu, które sobie wszystko przetwarzają, dbają o uwierzytelnianie i tak dalej, a następnie zwracają żądany content.

Okej. No to pora się przyjrzeć temu procesowi bliżej.

18.2.2 Ścieżka pliku

Pierwszą cechą wspólną, jaka się wyłoniła u wszystkich zgłoszonych problematycznych plików, było to, że pochodziły one z jednego obszaru - znaczy to tyle, że wszystkie były serwowane przez jeden serwer. Dobrze, to już jakaś wskazówka.

Spróbowałem zatem, odtworzyć ścieżkę, jaką typowy user i jego request obiera, i sprawdzić zachowanie aplikacji podczas takowej ścieżki. Pogrzebałem sobie w kodzie źródłowym, znalazłem kod, który odpowiada za obsługę wyświetlania i dostarczania linków do contentu dla użytkowników i wygenerowałem sobie zwykły url do danego zasobu, i faktycznie - kompletnie się on nie wyświetla, otrzymuje jakiś błąd. Ale moment. Spójrzmy na samego requesta, cóż się tam w nim dzieje.

W responsie nie pojawia się żaden błąd, zatem serwer nie widzi żadnego problemu i zwraca normalnego responsa. Hmmm, ciekawe...

18.2.3 Co się okazuje?

Okazuje się, że mimo tego, iż request, jakiego walimy, ewidentnie powinien zwracać nam responsa z plikami video, to z jakiegoś powodu, zamiast zwracać tego, co powinien, response ma typ... obrazka.

Sprawdźmy, co jest w body responsa. Najpierw rozmiar - w porównaniu z oryginalnymi plikami mniej więcej się zgadza. A co mówią nagłówki?

Po małej analizie okazuje się, że mimo tego, iż w typie responsa znajduje się obrazek, to serwer w body responsa serwuje nic innego jak dany plik video, który jeśli go zapisać, jest poprawnym filmem, hm ciekawe. I wiadomo teraz, dlaczego content nie jest w ogóle wyświetlany - jak przeglądarka/aplikacja ma wyświetlić film, skoro response każe wyświetlać jej ten content jako obrazek?

18.2.4 Zajrzyjmy do bazy

Postanowiłem zatem przejrzeć bazę danych w poszukiwaniu innych plików wideo, by zobaczyć, jaki mimetyp dla nich serwer wygeneruje.

Cóż, moje poszukiwania szybko pokazały mi, że serwer bez problemu identyfikuje pliki typu .avi czy .mpa jako video i zwraca je z poprawnym mimetypem, umożliwiając przeglądarce właściwe zachowanie. Skąd to zachowanie?

18.2.5 Format?

Czyżby to wina formatu? No bo .avi działa, .mpa też, ale okazuje się, że pewne pliki .mp4 już nie.

Bynajmniej, gdyż udało mi się znaleźć pliki .mp4 działające - to nie to.

A jak jest na innych serwerach? Na innych wszystko gra - brak żadnego pliku .mp4, który by nie działał z nieznanych powodów. Jeszcze ciekawsze.

Jak plik wygląda na dysku Przypomniałem sobie troszkę jednak o tym, jak pliki w ogóle są zapisywane na dysku, jak można rozpoznać, że dany plik jest akurat danego typu - bynajmniej nie po rozszerzeniu i pomyślałem, że przecież to może być to - coś być może w nagłówkach plików jest nie tak i stąd serwer nie rozpoznaje poprawnie typu. Może gdzieś przy zapisie coś idzie nie tak. Porównałem zatem dwa pliki w tym samym formacie za pomocą HxD - jeden działający, drugi nie.

Niestety, mimo drobnych oczywistych różnic w kodzie heksowym, tam, gdzie powinno być to samo, to wszystko się zgadzało, mimo tego, że na początku wydawało mi się, iż znalazłem w strukturze pliku błąd.

Kurczę.

Wpadłem zatem na inny pomysł - sprawdzę to innym programem, mediainfo, zobaczymy czy on poprawnie rozkoduje informacje o formacie pliku i tak dalej.

No i tutaj znowu zdziwienie - mediainfo bez problemu rozczytało plik poprawnie, wszystkie informacje o nim. Dlaczego zatem, skoro inne programy poprawnie rozpoznają pliki jako video, to serwer tego nie robi? Nie wiem, no ale nic, popróbuję na innych, może tu mi się poszczęści.

I zacząłem tak przeglądać i przeglądać. Nic. Kompletnie nic. Nagle...

18.2.6 Eureka

...dostrzegłem jednak pewien fakt. Otóż każdy działający, poprawnie rozpoznawany plik był kodowany, przy użyciu pewnej biblioteki w wersji załóżmy 2.11.2 lub starsze, natomiast wszystkie te pliki, które zostały zakodowane tą biblioteką w jakiejkolwiek późniejszej wersji, były błędnie rozpoznawane przez serwer. I tu się objaśniło wszystko.

Spojrzałem na stronę tej biblioteki i co się okazało? Mniej więcej w tym samym czasie, kiedy zaczął pojawiać się problem, miała miejsce premiera nowej wersji biblioteki. To utwierdziło mnie w przekonaniu, że ktoś gdzieś zrobił update jednej biblioteki na jednym tylko serwerze, nie biorąc pod uwagi faktu, że kod aplikacji działającej na serwerze wykorzystuje zależności akurat w pewnej specyficznej wersji i wersji starszych, oraz to do nich jest przystosowany, przez co plików kodowanych nowszymi wersjami nie wykrywa poprawnie a każdy plik, który ląduje na serwerze, zanim zostanie zaserwowany użytkownikom, jest przetwarzany właśnie za pomocą akurat zainstalowanej biblioteki.

18.2.7 Podsumowanie

Mimo tego, że de facto samego programowania tu nie było, to przyznam, że i tak świetnie się bawiłem podczas wykonywania tego zadania. Trochę jak detektyw odkrywający tajemnice wielkiej zbrodni.

Pokazało mi ono również, że wiedza ogólna, o której to tak często się mówi ‘a po co mi to’, czasem się przydaje.

Podsumowując, czasami wystarczy zmienić jedną cyferkę z 2.11.2 na 2.11.3, by nagle coś gdzieś przestało działać, a praca developera to nie zawsze tylko klepanie nowych landing pageów.

19 Kultura

O kulturze pracy programisty słów kilka. O nastawieniu i takich tam. Mimo, że pozornie rzeczy drugorzędne, tak naprawdę niektóre z nich odgrywają bardzo ważną rolę w procesie wytwarzania oprogramowania.

19.1 Trochę o menadżerach/liderach

Dobry menago/lider zespołu jest osobą, która może albo zespół doprowadzić do ruiny albo dodać mu skrzydeł. Zespoły to podstawowy blok budulcowy firm. Jeśli menago może zniszczyć lub uskrzydlić zespół, to podobnie może z całą firmą. To podkreśla jak ważny jest ktoś normalny w roli lidera. W jaki sposób uskrzydla się lub niszczy? Operując na tym, z czego zespół się składa. Na ludziach. Człowiek to podstawowa i najważniejsza jednostka w firmie, której chęć, a głównie wygenerowana niechęć do pracy, jest mocno zależna od wpływów liderów. Opiszę trochę przemyśleń moich z tym związanych a raczej obserwacji na temat drogi, jaką co niektórzy przechodzą i jak ich to zmienia w efekcie.

Na mej programistycznej drodze spotkałem już wielu menadżerów w wielu różnych firmach. Niektórzy byli wspaniali, wpłynęli pozytywnie na moje życie, inni byli kompletnymi kretynami, jeszcze inni zaś są dla mnie teraz prawie jak rodzina. Po pewnym czasie zacząłem przyuważać pewne wzorce, które występowały prawie zawsze, i które pozwoliły mi wyizolować trzy graniczne poziomy bycia menadżerem.

19.1.1 Poziom pierwszy: mistrz Excela

Mistrzowie Excela to często świeżo upieczeni menadżerowie, liderzy, którzy nie mają jeszcze za dużo doświadczenia, ale za to pełno ego, dumy i buńczuczności. Świat stoi przed nimi otworem i jest do zdobycia! Uwielbiają swoje arkusze, wszystko w nich zapisują, optymalizują, czasami uprawiają mikromanagement. To ludzie będący w początkowym okresie kariery menadżerskiej, gdzie człowiek może się nieco upoić początkowo swoją władzą.

Ten typ człowieka widzi często tylko jedną rzecz: piniondz. Oj często takich spotykałem do pewnego czasu, za często. Pazeroty. I to oni są właśnie najbardziej niebezpiecznym typem menadżera, jaki się firmie może przytrafić, spośród trzech wymienionych przeze mnie.

Pozornie rzecz ujmując mistrzowie Excela wyglądają całkiem spoko. Cyferki się zgadzają, plany, prezentacje, no bajeczka. ALe nie do końca. W rzeczywistości wszystko się rozjeżdża. Dlaczego są najbardziej niebezpiecznym rodzajem? Z powodu fasady, którą narzucają - nie są tymi, których zwykle podejrzewa się o bycie źródłem problemu, ale po poznaniu gatunku tego zwierza, w przyszłości ciężko nie dostrzec i potem bardzo często ich rozpoznajesz z łatwością. Jak? Po prostu znajdź najbardziej nieszczęśliwy zespół w firmie. Milczący zespół, który trwa w cichej rezygnacji, pozornie z jakiegoś nieuchwytnego powodu. Nie szukaj zespołu, który skarży się. Oni nadal mają nadzieję. Szukaj czegoś innego.

Zespoły, które skarżą się - nadal mają chęć aby robić przynajmniej jedną rzecz - skarżyć się, narzekać. Prawdziwe porażka i rezygnacja następują, gdy nic nie ma, gdy ludzie są cisi, gdy są tak zmęczeni wszystkim, że wolą pozostać cisi i udawać, że wszystko jest dobrze, ale

niestety wtedy jest tak źle, że nawet na zewnątrz, po bliższym spojrzeniu, widać jaka jest prawda. Gdy wrogowie kłócą się o coś lub się przekomarzają, to to nie jest jeszcze walka, a rozmowa, rozmawiają w pewien sposób, jest to jakaś forma komunikacji. Prawdziwa walka i dramat zaczyna się, gdy ustaje jakakolwiek forma komunikacji i pojawiają się działania, pojawiają się czyny i akcja. Terminy nie są dotrzymywane, projekty nie powodują. Często rotacja w zespole jest również bardzo wysoka. Ludzie albo zostają zaplątani w taki zespół z rezygnacji i po prostu pozostają tam przez bardzo długi czas, gnijąc na nizinach rozpaczy, albo uciekają natychmiast. Zwykle u góry takiego zespołu znajdziesz to, co nazywam Menedżerem Pierwszego Poziomu - Mistrz Excela.

Mistrzowie Excela widzą tylko liczby, tylko piniondz. Nigdy nie przyznają się do błędu, nigdy nie myślą, że popełnili błąd - zawsze to ktoś inny lub zespół jest winny. Nie widzą ludzi, nie widzą kolegów z zespołu, widzą zasoby. Zasoby, które mogą być spożytkowane, albo rzekłbym raczej wykorzystane, które mogą być "zoptymalizowane". Nie ma nic więcej. Jak prosta matematyka. Jeśli masz zadanie na 80 godzin, zajmie to około dwóch tygodni pracy jednego programisty, jeśli zatrudnisz dwóch programistów, zajmie to tydzień, prawda? Każdy zawsze wykonuje pracę w bardzo stałym tempie, niezależnie od kontekstu i sytuacji, prawda? Prywatne życie nie ma w ogóle wpływu na życie zawodowe, prawda? Marża i zyski to wszystko, co się liczy, prawda? Czy te liczby wyglądają dobrze? Jeśli tak, to wszystko idzie tak, jak powinno, prawda? PRAWDA?! Technicznie tak. Praktycznie nie.

Menedżer taki jak ten niszczy ducha. Niszczą ludzi, którzy tworzą zespół.

Bez ludzi nie masz zespołu. Bez ludzi twój produkt nie zostanie uruchomiony. Bez ludzi terminy nie zostaną dotrzymane, ale oni tego nie widzą. Zasoby są tylko tym. Zasoby. Przynajmniej w ich głowach.

To jest zwyczajnie szalone i niszczące. Nic gorszego niż poczucie braku docenienia, niż poczucie, że jesteś tylko zębem w dużej cholernie maszynie, którą można wymienić jednym ruchem ręki menedżera. Nie oszukujmy się, w rzeczywistości tak jest czasami, chociaż każdy z nas nie jest AŻ TAK specjalny, albo przynajmniej częściowo, ale kurwa, liderzy i menedżerowie są tam, aby to uczucie zniknęło. Przynajmniej częściowo.

Spotkałem wielu takich ludzi. Mają jedną rzecz wspólną. Zawieruchę, jaką produkują.

Najlepsze, co mogą osiągnąć menedżerowie pierwszego poziomu, to bardzo krótkotrwałe projekty. Projekty, w których ludzie nie mają szansy poznania się, spotkania z trudnościami i prawdziwymi wyzwaniami. Powiedziałbym maksymalnie parę tygodni. To jest możliwe z poziomu jeden, ponieważ w tak krótkim czasie nie są w stanie wystarczająco zniszczyć swoich członków zespołu. Nie mogą ich zaatakować do stanu rozpaczy, chyba że są bardzo utalentowani w tym.

Zwykle ci ludzie albo pozostają zamknięci w tym nastawieniu przez dłuuuuuuugi czas albo spróbują różnych rad, aby może poprawić lub naprawić zespół, ponieważ często nie widzą błędu w swoim zachowaniu. W każdym razie.

Zazwyczaj prowadzi to do nauki o empatii. Co nas prowadzi do...

19.1.2 Poziom drugi: empatyczny akolita excela

Tu rzeczy stają się nieco interesujące, powiedziałbym. Osoba, która osiągnęła ten poziom, jest w ciekawym miejscu. Zazwyczaj napędzana ciągłymi porażkami i dążeniem do poprawy ich arkusza, wydajności zespołu i innych BARDZO WAŻNYCH WSKAŹNIKÓW

KORPORACYJNYCH, uczą się o empatii. O tak! EMPATIA! To co powinniśmy promować, aby ludzie czuli się lepiej, bardziej związani z zespołem i firmą, więc wydają się lepiej? Brzmi dobrze, prawda?

Cholera, nie. To jest problem z menedżerami drugiego poziomu. Widzą empatię tylko jako narzędzie, narzędzie do zmuszenia zespołu do pracy. Albo do lepszej wydajności. W każdym razie, to nie jest prawdziwa empatia w żadnym razie. To po prostu obliczona manipulacja w rzeczywistości.

Nie zrozum mnie źle - prawdziwy lider POWINIEN i MUSI znać się trochę na manipulacji, powinno to być dla niego naturalne, ale nie powinien tego nadużywać. Dobrzy menedżerowie powinni umieć odczytywać ludzi i ich emocje, ale nigdy nie wykorzystywać tego przeciwko nim, nigdy nie manipulować swoim zespołem w celu wykonania pewnych rzeczy. To nie działa na dłuższą metę, ale może przynieść czasem krótkotrwałe korzyści. Co jest bez sensu.

Ta umiejętność powinna być wykorzystywana do współczucia, wiedzieć kiedy się wycofać, dać członkowi zespołu trochę luzu czasami, z różnych powodów, które zaobserwowałeś.

Więc chociaż menedżer drugiego poziomu jest trochę lepszy od pierwszego, to nadal jest zły, przynajmniej próbuje i udaje. To już coś, prawda? O jakim udawaniu i jakiej udawanej empatii tu mówię?

Oh, hi Susan. Dobra robota na warsztacie empatii z naszym trenerem Agile w zeszłym tygodniu. To bardzo ważne, aby promować empatię w naszej firmie! Twoja mama zmarła wczoraj, więc chciałabyś dziś wyjść wcześniej? Och, nie ten rodzaj empatii. Może kolejne warsztaty z trenerem Agile/Empatii pomogą Ci to poprawić? Firma za to zapłaci! Zastanów się nad tym Susan i trzymaj tak dalej, mistrzu!

Coś w tym stylu bym powiedział. To trochę trudne do opisania, ale można to poczuć i zauważyć przez większość czasu. To jest kiedy czujesz się jak bydło, które jest brane pod opiekę przez nie tak wielki właściciel, który nie kocha jego zwierząt, on tylko robi gołe minimum tak oni mają mięso akceptowalnej jakości. Tak, coś w tym stylu.

To jest poziom drugi. Choć nadal lepszy od poziomu pierwszego, to również stanowi zagrożenie, tylko mniejsze. Ten jest trochę trudniejszy do zauważenia na powierzchni, ale po rozmowie z nimi trochę, można również dostać.

Poziom drugi faktycznie może dostarczyć jakieś projekty. co najwyżej te krótko- i średniodystansowe. Może kilka miesięcy, pewnie najwyżej pół roku. W tym okresie da się jeszcze utrzymać fasadę, udawać, więc zespół może być jeszcze w porządku, będzie nawet osiągał fajne wyniki. Ale to jest pułapka, gdy trafi się na dłuższy projekt, gdzie ludzie naprawdę się poznają lub pracowali ze sobą wcześniej. To właśnie tam ostatecznie ponoszą porażkę. Czasami miażdżąco. Nie ma tak wielu rzeczy gorszych niż uczucie, gdy po uświadomieniu sobie, że twój kierownik/lider był liderem dwa. Że dbano o ciebie tylko po to, byś osiągał lepsze wyniki, manipulowano tobą. Resentyment podąża za goryczą, poczuciem zdrady.

Wracając do tematu - projekty długoterminowe. Teraz to jest miejsce, w którym pojawia się prawdziwe wyzwanie i zabawa, a wraz z nią wchodzi....

19.1.3 Poziom trzeci: człowiek

Zazwyczaj poziom trzeci to poziom pierwszy lub drugi, który po prostu zawiódł wystarczająco dużo razy, który został złamany wystarczająco dużo razy i połączony w całość

przez upływ czasu i autorefleksję. Rzadko jest to typ geniusza, który po prostu urodził się, aby być liderem/menedżerem.

W większości przypadków trójki doświadczyły w swoim życiu wiele razy porażki i totalnej, miażdżącej klęski. Są świadome, że jesteśmy tylko ludźmi i niczym więcej. A ludzie, na niektórych poziomach, są niezwykle twardzi i wytrzymali, ale czasami mogą być tak krusi i delikatni, że nawet najdrobniejsza rzecz może sprawić, że roztrzaskają się na kawałki. Skąd wiedzą? Bo już tam byli.

Wiedzą również, że nawet jeśli jesteś złamany, możesz być znowu cały. Nawet jeśli się roztrzaskasz, możesz zostać odbudowany. To samo dotyczy zespołu.

To są menedżerowie, którzy naprawdę się o Ciebie troszczą, z którymi się zaprzyjaźnisz i których autentycznie lubisz.

Dlaczego uważam ich za tak ważnych? Ponieważ jeśli masz z nimi szczerą więź, a oni mają szczerą więź z tobą, wasze partnerstwo, wasza praca zespołowa będzie po prostu lepsza. Kiedy w całym zespole panuje wspólne zaufanie i zrozumienie, dzieją się magiczne rzeczy. Nie ma potrzeby mikrozarządzania. Praca po prostu zaczyna się dziać, ponieważ ludzie są chętni do jej wykonywania, chętnie przychodzą do pracy.

Nawet jeśli to, nad czym pracujecie, jest całkowitym śmieciem, to przynajmniej macie siebie, prawda? Wiesz, że twój kierownik ma plecy twoje i twojego zespołu. Wiesz, że nie chodzi tylko o wskaźniki KPI i twoje wyniki. Co jak na ironię sprawia, że osiągasz lepsze wyniki.

To tylko prosty, ale bardzo głęboki i znaczący akt postrzegania cię takim, jakim jesteś - człowiekiem, a nie trybikiem w maszynie. Brzmi to tak prosto, a jednak czasami osiągnięcie tego poziomu zajmuje lata i jest trudne. Tak jest. Ludzie są skomplikowanymi istotami.

Jak rozpoznać poziom trzeci? Nie boją się swoich porażek, potrafią być naprawdę wrażliwe i pokazać swoją słabą stronę, podzielić się, sprawić, byś się trochę otworzył i odniósł. Mają bardzo dobre relacje ze swoim zespołem i znają ludzi, którzy pod nimi pracują. To jest złe słowo. Nie mają ludzi, którzy pracują pod nimi. Mają ludzi, którzy z nimi pracują. To jest jedna z większych różnic tutaj. Bycie menedżerem i posiadanie partnerstwa z zespołem w celu osiągnięcia wspólnego celu versus posiadanie zespołu niewolników, którym się zarządza.

Nie wiem, co jeszcze można tu powiedzieć. To są po prostu ludzie prawdziwie empatyczni, którzy nie skupiając się na poprawieniu spektaklu, poprawili go tylko tym aktem. Trochę ironicznie.

Jak dobrze znasz swoich kolegów z zespołu? Kiedy ostatni raz zrobiłeś coś miłego dla ludzi, z którymi pracujesz? Czy wszyscy czujecie się bezpiecznie w miejscu pracy/zespole? Te poziomy odnoszą się nie tylko do menedżerów czy liderów. Mogą odnosić się niemal do każdego i każdy w końcu wpływa na to, jaki poziom menedżera dostaniesz.

Menedżer zarządza zespołem/projektem, ale zespół również uczy menedżera wielu rzeczy. To jest relacja dwukierunkowa. Jesteś falą - wpływającą na inne fale wokół Ciebie, więc... Pomyśl o tym przez jakiś czas.

Teraz zatrzymaj się na chwilę i zadaj sobie to pytanie: którą z nich jesteś?

Podsumowując. Przestań tak bardzo skupiać się na KPI, arkuszach Excela, danych z Performance. Skup się na ludziach, z którymi pracujesz, zobacz ich takimi, jakimi są - ludźmi, a nie zasobami czy liczbami w arkuszach Excela. Poznaj ich, ich mocne i słabe strony.

Dobry generał wie, gdzie przydzielić konkretnego żołnierza na podstawie jego charakteru, dobry stolarz wie, gdzie użyć danego kawałka drewna na podstawie jego cechy. Dobry menedżer jest taki sam. Jest to z korzyścią dla wszystkich. Bądź dobrym menedżerem. Bądź empatyczny. Skup się na ludziach. To jest to.

Wszyscy już tam byliśmy. Jeśli obecnie znajdujesz się na poziomie pierwszym, lub drugim, nie przejmuj się. To część procesu. Ja też nie jestem trójką. Heck, nie jestem nawet menedżerem sam, więc co ja tam wiem!

19.2 Kultura projektowania

Ważnym i istotnym elementem każdego kawałka kodu czy systemu, który tworzysz, jest jego design. Co to konkretnie znaczy?

Otóż chodzi o to, by kod, jaki napiszesz był **przemyślany**. Tutaj wchodzi coś, co nazywamy `incremental design`. Znaczy to tyle, że generalnie nie da się od razu, dla projektów mniej trywialnych aniżeli Hello World, stworzyć rozwiązania idealnego. Zamiast tego trzeba myśleć o dostarczaniu rozwiązania jak o procesie, przez który przechodzimy krok po kroku.

Najpierw powinniśmy zrozumieć dany problem, dobrze go pojąć i wrzucić w nasz mózg, by przemielić go na analogie i koncepcje dla nas zrozumiałe.

Błędy i bycie w błędzie to część tego procesu. Oswój się z tym. Prawie nigdy twój pierwszy pomysł na rozwiązanie problemu, nie będzie kompletny czy poprawny. Zamiast tego problem warto rozbijać na malutkie cząstki, które swą trywialnością będą zbliżały się do wspomnianego wcześniej Hello Worlda. Oczywiście trzeba mieć tu smak i wyczucie, gdyż za bardzo rozdrabniać się też nie można.

Gdzie leży granica? Wyczucie, moim zdaniem, przyjdzie z czasem i praktyką.

Coś, co od siebie polecam, to następujące kroki:

1. Zrozumienie problemu i zadawanie pytań. To tu jest miejsce na przemyślenia, analogie i próbę oswojenia się z danym problemem. Pytaj, nawet o pozornie głupie rzeczy, zrozum o co tutaj chodzi i jaki jest problem. Zrozum **domenę** problem i **zamodeluj** go sobie w jakiś sposób w swojej głowie. Co to znaczy? Abstrakcyjny koncept przedstaw i wyobraź sobie w nieco mniej abstrakcyjny, zrozumiały dla siebie sposób.
2. Rozrysuj koncepcję z punktu wyżej lub podsumuj swoje przemyślenia w formie pisemnej. To tutaj warto by skonfrontować twoje wyobrażenia z rzeczywistością, zapytać odpowiednich osób, czy to właśnie o to chodziło. Diagramy są tutaj nieziemsko przydatne. Zaznaczam jednak, że nie chodzi o szczegółowy plan implementacji czy coś w tym stylu. Nie. Generalny rozrys tego, o co się rozchodzi w problemie.
3. Dopiero teraz pora na pierwsze implementacje najmniejszych kawałków. Zadanie podzielone na małe części, to zadanie łatwe do ogarnięcia. Mały kawałek łatwiej ogarnąć jednorazowo, aniżeli ogromny problem. Może zastosuj teraz TDD? Napisz jakiś kawałek kodu, który odwzoruje model i jakiś kawałek rozwiązania problemu, przetestuj i skonfronuj z rzeczywistością. Warto tu zaznaczyć, że kod należy pisać tak, by jak najłatwiej było go potem zmienić, zastąpić, zrefactorować. Im bardziej skomplikowany i powiązany z innymi fragmentami kod, tym trudniejszy w edycji i refactoringu. Nie jest to jednak zachęta do tego, by tworzyć ogromnego molosa,

który będzie zawierał tony boilerplate kodu i robił wszystko, jednocześnie nic. Nic bardziej mylnego.
4. Mając w ręku coś, co sensownie faktycznie rozwiązuje problem i odwzorowuje dobrze model problemu, teraz można nieco zrefaktorować, poprawić, podciągnąć, uczynić kod łatwiejszym i lepiej napisanym.
5. Krok 3 i 4 powtarzamy ile trzeba. Być może też krok 2.

Nie zachęcam tutaj do tego, by od razu z góry tworzyć **waterfallowy** model, gdzie z góry planujemy wszystko w szczególe.

Iteracja po iteracji poprawiamy swój design i kod, idąc do przodu. To o to się rozchodzi. Dzięki temu możemy testować malutkie kawałki, walidować swoje pomysły na bieżąco (np. z klientem poprzez dostarczanie malutkich fragmentów, ale często) i żyć ze znośnym kodem.

Programowanie paradoksalnie, jeśli dobrze zrobione, często nie opiera się na programowaniu.

Otóż najwięcej czasu powinno zająć analizowanie problemu i myślenie, planowanie. Zmiana koncepcji jest łatwa, to tylko myśl lub diagram. Jest to relatywnie tania operacja. Zmiana w kodzie zaś często bywa kosztowna. Twoim zadaniem jest, by zawsze **myśleć** i mieć na uwadze różne względy.

Sytuacja jak na memie wyżej bywa przykra. Pamiętaj, że w przyszłości to ty pewnie będziesz utrzymywał swój własny kod, przynajmniej przez chwilę. Zrób sobie z przyszłości zatem przysługę i przemyśl dwa razy kod, który piszesz, przyłóż się. Nie warto być niechlujem i leniem czasami. Znaczy paradoksalnie to właśnie lenie najwięcej pomyślą, bo łatwiej jest myśleć aniżeli stukać kod. Zatem podążajmy drogą prawdziwego, a nie pozornego, lenia i dbajmy o design. Dobrze zaprojektowana aplikacja potrafi być wdzięczna przy pracy zarówno dla programisty jak i dla użyszkodnika końcowego.

Czasem starczy poświęcić godzinę myślenia, by zaoszczędzić sobie 4-8h klepania kodu.

Przy okazji nie bój się tego, że ktoś rozliczy cię z koncepcyjnej pracy na zasadzie liczenia linijek kodu jakie wyprodukowałeś/wyprodukowałaś. To nie o to chodzi w pracy kreatywnej. Żaden sensowny pracodawca nie będzie rozliczał cię z tego, że napisałeś 20 linijek zamiast 40. Liczba linii wyplutego kodu nie jest żadną miarą. Oczywiście nie mówię o skrajnościach, kiedy jesteś juniorem i przez cały miesiąc dodałeś jeden commit zmieniając jedną linijkę, co prawdopodobnie oznacza, iż robisz kogoś w jajo.

Do obejrzenia: https://www.youtube.com/watch?v=flxmpq7_tdM

19.3 O wartościach

Podobno o wartościach i zasadach najwięcej mówią ci, co ich nie mają, bo reszta po prostu nimi żyje. Mimo tej mądrości ludowej, trochę się o nich rozpiszę, ale tylko delikatnie. Chciałbym wspomnieć mimo wszystko o pewnych wartościach, jakie mi w pracy przyświecają i jakie ja osobiście lubię widzieć u siebie oraz innych. Te cechy, to coś, co uważam, że stanowi podstawę dobrego programisty, kodera, pracownika.

1. Szczerość i otwarty umysł To dla mnie podstawa u każdego. Ludzie mają problemy z rozmawianiem ze sobą szczerze i z komunikacją. To na szczerości winna się opierać każda relacja, zwłaszcza zawodowa. To zaś wymaga otwartego umysłu i schowania ego do kieszeni.

Figure 5: Tom Scott - meme. Pozwoliłem sobie użyć wizerunku, mam nadzieję, że nie kole to w oczy.

2. Szacunek (ludzi ulicy, znaczy ten no, klawiatury) Miej szacunek dla drugiego człowieka, klienta, współpracowników. Szanuj ich czas, wiedzę i pieniądze. W ten sposób można budować relacje na lata. Szanuj też przede wszystkim siebie, bo jeśli sam się nie szanujesz, to trudno by inni to robili.
3. Empatia i bycie ludzkim Na koniec dnia jesteśmy tylko ludźmi. Postaw się czasami w butach drugiej osoby zanim coś powiesz. Dobra atmosfera i zaufanie budowane na empatii i sympatii do drugiego człowieka pozwala osiągnąć w zespole synergię, która niesamowicie usprawni, ale i uprzyjemni pracę. Bądź człowiekiem. Nie bądź dzbanem.
4. Poczuwanie się do odpowiedzialności i bycie dumnym ze swojej pracy Jak coś tworzysz, bierz za to odpowiedzialność, bądź z tego dumny. Ta zasada ma daleko idące implikacje jak dbanie o swój kod, dostarczanie holistycznych rozwiązań, które zostały przetestowane, czasem `going the extra mile`, jeśli trzeba, posiadanie i wykazywanie inicjatywy zamiast ślepego podążania za taskami.
5. Merytokracja Moim zdaniem ego chowamy do kieszeni. Krytykę, merytokratyczną znaczy się, twojego rozwiązania, kodu czy nawet swojej osoby, nie traktuj osobiście i jako atak. Nie. To raczej cenne lekcje, z których można się uczyć, ino trzeba odsunąć ego na dalszy plan, a skupić się na tym, co ważne, czyli dobro projektu, klienta, dostarczanie wartości. Nie znaczy to, że masz się ciągle poświęcać, nie. Szanuj siebie przede wszystkim, ale powtarzam. Ego chowamy do kieszeni a rzeczy w pracy staramy się bazować na logice, argumentom i dyskusjom prowadzonym w dorosły sposób. Dodatkowo chciałbym też nadmienić, że nie masz się też co bać proponować swoich rozwiązań, nawet jeśli idą niejako w sprzeczności z rozwiązaniami np. lidera, starszego programisty czy klienta. Jeśli pracujesz z odpowiednimi ludźmi, to docenią to a rozwiązanie wybierzecie na podstawie faktów, a nie innych czynników.
6. Współpraca Bądź niezależny w swojej pracy lokalnie, natomiast zorientowany na współpracę globalnie. Co to znaczy? Myśl i miej gdzieś z tyłu głowy szerszą perspektywę i staraj się koordynować swoje indywidualne starania tak, by wpasowywały się w jakiś szerszy plan. Przykład tutaj to: "Zacznę implementować najpierw tego taska, bo dzięki temu odblokuję frontend."
7. Zakładaj dobre intencje When in doubt, ask. Tak w skrócie. Jeśli pracujesz z kimś, to znaczy, że prawdopodobnie ta osoba przeszła podobną weryfikację co i ty. Znaczy to, że jest jakoś kompetentna, takie założenie mam. Do tego zakładam, chyba, że fakty dowiodły inaczej, że wszyscy w zespole mamy dobre intencje. Zatem zanim coś zrobię, jeśli posiadam chociaż cień wątpliwości, co do natury sytuacji, wolę dopytać, co ktoś miał na myśli. Może zły dzień, może źle coś odebrałem, może komuś żona z rana dała popalić. Kto wie.

Uważam, że kierowanie się tymi punktami wyżej Ci nie zaszkodzi. Warto mieć je z tyłu głowy. Bo w IT często nie chodzi o IT a o ludzi i ich problemy. Jak o tym pomyślisz, to za to właśnie ci płacą - za rozwiązywanie problemów biznesowych, problemów określonych ludzi, ułatwianie ich życia. To za to płaci się grube hajsy. Dostarczanie wartości. A to wymaga tego, by mieć odpowiednie cechy i zrozumienie, być określonym człowiekiem. Myślę, że atrybuty wyżej są takimi, które pozwolą ci błyszczeć w zespole jak i firmie. To podstawy bycia normalnym człowiekiem, aczkolwiek czasem rzadko spotykane. Wyróżniaj się nie tylko twardymi umiejętnościami z informatyki, ale również byciem dobrą osobą.

Cholllera, ale poetycko i buńczucznie.

19.4 Kaizen

Parę słów chciałbym rzec o filozofii Kaizen. Co to znaczy? To taka japońska myśl, której motywem przewodnim jest ciągła, ustawiczna praca nad sobą i dokonywanie małych usprawnień, które razem składają się na coś większego.

Generalnie sprawdza się to bardzo w naszym zawodzie jak i w sumie wszędzie.

Codziennie powinniśmy czynić kod odrobinkę lepszym dodając nowe funkcjonalności lub poprawiając istniejące. Codziennie powinniśmy chociaż odrobinkę poczytać, pouczyć się czegoś nowego. Po pewnym czasie zaczyna działać tutaj procent składany i okazuje się, że z tego codziennego, niewielkiego wysiłku, rośnie coś ogromnego. 1% dziennie daje 3700% rocznie po uwzględnieniu procenta składanego. Ale czaszka.

Zatem Kaizen czy The Slight Edge to coś, co powinno być obecne w kulturze twojej pracy.

19.5 Zarządzanie portfelem inwestycyjnym

Tajemniczy nagłówek. O cóż chodzi? Mianowicie o to, że każdy z nas w życiu inwestuje, nawet jeśli nie gra na giełdzie. Jak to? Tak. Ty sam jesteś swoją największą inwestycją. Twoja wiedza, kariera, życie.

Wiedza jest bardzo cenna. To coś w co należy ciągle inwestować, gdyż daje ogromne zwroty zazwyczaj.

19.6 Relacja z prowadzenia pierwszego retro

Ostatnio miałem okazję po raz pierwszy w życiu facylitować retrospektywę, to będzie moje podsumowanie tego doświadczenia.

Pomysł zaczął się od tego, że jeden z founderów firmy dla której wtedy pracowałem zapytał mnie, czy nie chciałbym tego zrobić.

Ja - agile/scrum-sceptic/whatever-sceptic, prowadzący retrospektywę? Brzmi jak przepis na katastrofę.

Cóż, zgodziłem się bez wahania. Pomimo tego, że w większości nie jestem wielkim fanem całego obecnego szumu wokół agile, scrum slaves (uwielbiam ten termin i często go nadużywam) i tak dalej, to dodatkowo jedną przyjemną rzeczą, która przychodzi mi do głowy, gdy słyszę retrospektywę, jest piosenka Jinjera. A tak na poważnie - albo to tylko ja, albo niektórzy ludzie oszaleli, wymyślając coraz bardziej ekstremalne i dziwne praktyki, jeśli chodzi o AGILE, które nie przynoszą większych efektów poza denerwowaniem deweloperów i marnowaniem czasu ludzi z biznesu.

W każdym razie - widząc mnóstwo takiego szumu, od razu staję się sceptyczny. Stąd postanowiłem sprawdzić, co jest po drugiej stronie płotu.

Oczywiście zdawałem sobie sprawę, że jest wiele wartościowych praktyk, rzeczy, nawyków i schematów myślenia, których można się nauczyć od gromady Agile, to nie jest tak, że mówię, że wszystko co agile jest bezużyteczne. Nie interpretuj moich słów w ten sposób. Po prostu niektórzy ludzie robią szalone rzeczy, które nie są Agile i nazywają je Agile. Szczególnie w Polsce tak się wydaje. A to czasem prowadzi do szaleństwa.

Widziałem to na własne oczy, żaden psychoterapeuta nie pomógł później. Odbiegam od tematu.

Ponieważ był to mój pierwszy raz, zaoferowano mi pomoc. Firma załatwiła mi kilka godzin z naszym zaprzyjaźnionym trenerem agile, który pomaga nam w budowaniu pewnych procesów w firmie. Odbyliśmy krótką rozmowę, zanim zaplanowałem swoją retrospektywę.

Dowiedziałem się trochę o ustawieniu sceny, zbieraniu danych, insightach, fajnych narzędziach do wykorzystania i tak dalej. Ta szybka sesja, pomimo moich początkowych wątpliwości co do niej, pomogła mi całkiem sporo. Więc wielkie dzięki tutaj i shout out dla Marcina Konkela!

19.6.1 Ready, set, go

Tak oto nadszedł dzień i godzina. Zaczęło się spotkanie. Zdenerwowałem się, gdy już wszyscy byliśmy na miejscu. Gdyby nie to, że była to moja pierwsza retrospektywa, to ze względu na sposób pracy, a pracujemy w 100% zdalnie, była to retrospektywa zdalna. Osobiście uważam, że komunikacja zdalna jest trudniejsza - trzeba być bardziej precyzyjnym, dokładnym, bardziej komunikatywnym i uważnym. Potrzeba trochę czasu i umiejętności, żeby się do tego przyzwyczaić.

19.6.2 Ustawienie sceny

Zanim zaczęliśmy, postanowiłam ustalić kilka jasnych zasad i powiedzieć na głos moje założenia, które brzmiały: Nie przerywamy sobie nawzajem podczas rozmowy. Biorąc pod uwagę, jak zdalnie prowadzimy retrospektywę, musimy się tego trzymać jeszcze bardziej niż kiedykolwiek. Jeśli masz coś do dodania poza czasem swojej kolejki/mówienia, podnieś rękę do kamery. Prelegent lub moderator powinien to zauważyć i dać Ci szansę. Pierwsza osoba, która zabierze głos w danej rundzie dla danego ćwiczenia jest wybierana przez facylitatora. Osoba ta dostaje trochę czasu na odpowiedź na pytanie, podczas gdy inni słuchają. Po zakończeniu wypowiedzi wskazuje kolejną osobę, która powinna się wypowiedzieć. Po tym jak ustaliliśmy te zasady, wspomniałem o jednej rzeczy.

To była retrospektywa dla naszego zespołu. Jestem częścią tego zespołu, aktywnie uczestniczę w rozwoju i pracy. Dziś jednak pełniłem rolę moderatora. Nie byłem niczym więcej. Dlatego wyraźnie zaznaczyłem, że dziś robię właśnie to - facylituję. Monitoruj, zarządzaj, ułatwiaj i moderuj rozmowy. Niech oni zrobią resztę.

Ponadto, jestem trochę bezpośrednim facetem - zamiast używać jakichkolwiek wymyślnych narzędzi, po prostu poszedłem z rozmową. Żadnych karteczek post-it. Żadnych tablic. Tylko nasze twarze, głosy i ja, cicho notujący całe retro w tle. Tak - przerażające, że faktycznie musisz rozmawiać z ludźmi. Ugh.

19.6.3 Rozgrzewka

Początkowo chciałam nieco rozjaśnić nastrój, skłonić ludzi do rozmowy, bo doskonale zdawałam sobie sprawę z tego, co w danej chwili czują. Niektórzy z nich zostali przerwani w połowie pracy, zostali wyrzuceni z flow, żeby uczestniczyć w tym spotkaniu. Są teraz w innym kontekście, więc nadszedł czas, aby sprowadzić ich tutaj, do tej chwili.

Jak to zrobić? Zadałam wszystkim proste pytanie.

Co, czy to w życiu prywatnym czy zawodowym, sprawiło, że ostatnio jesteś szczęśliwy?

Pozwoliło to zespołowi na nowe, małe spojrzenie na siebie nawzajem. Być może znalezienie nowego sposobu patrzenia na siebie. Wiem, że brzmi to prosto, ale tak właśnie było w rzeczywistości. Wszyscy zaskakująco podzielili się szczęśliwymi małymi momentami ze swojego prywatnego życia i wiecie co? Czułem, że to zacieśniło więzi, które łączą nas wszystkich jako zespół.

19.6.4 Retrospektywa w przeszłości

Przed tą przeprowadziliśmy w przeszłości inną retrospektywę z innym trenerem agile, w wyniku której zadeklarowaliśmy kilka celów. Teraz nadszedł czas, aby je sprawdzić. Zapytałem więc każdą osobę, która miała zdefiniowany cel, o status jej celu, jak przebiega i jaka jest jej opinia, dotychczasowe doświadczenia.

Wszystkie osoby zgodnie stwierdziły, że poprzednie cele były potrzebne, są wykonane lub w trakcie realizacji i jakoś poprawiły naszą pracę. Miło.

19.6.5 Dobrodziejstwa - zbieranie danych

Pracujemy w bardzo wielokulturowym i międzynarodowym zespole, a jednak mimo to jakoś tak wyszło, że na tej retrospekcji byli tylko polscy koledzy z zespołu. Sama będąc Polką, wiem o naszych dziwactwach, a jednym z nich jest narzekanie. To jak hiperinflacja, jak już się zacznie, to nie ma odwrotu.

Aby wprowadzić odpowiednią atmosferę, postanowiłem najpierw porozmawiać o dobrych rzeczach, aby wprowadzić ludzi w dobry nastrój. Zadałem im wszystkim pytanie:

Jakie są pewne rzeczy, które twoim zdaniem robimy, dzięki którym dostarczamy szybciej, lepiej i ogólnie przyczyniamy się do tego, że wykonujemy lepszą pracę lub z czego jesteś ostatnio zadowolony?

Tutaj każdy dzielił się swoją perspektywą, spostrzeżeniami i tak dalej.

19.6.6 Baddies - zbieranie danych

Więc, gdy jeszcze byli w dobrych nastrojach i aura była pozytywna, wrzuciłem bombę, która czasem otwiera bramy piekieł na polskich retrospektywach - co poszło źle? Przejdźmy do konkretów. Dość cukrowania. Po polsku. Cebulowy sposób.

I oto zaczęło się narzekanie. Albo tak mi się wydawało - że się zacznie. Zamiast tego, nie zaczęło.

Zadałem pytanie:

> Co według Ciebie zrobiliśmy ostatnio źle? Co zaobserwowałeś, co sprawia, że dostarczamy mniej, wolniej? Co sprawia, że twoja praca jest mniej przyjemna?

Zamiast bezmyślnego bełkotu, zespół zrobił coś miłego. Przedstawili konstruktywne i mądre spostrzeżenia, spostrzeżenia i komentarze. Szczerze mówiąc, było to dość niesamowite. Czasami jestem zaskoczony tym, jak mądrzy są ludzie, z którymi pracuję.

19.6.7 Happies - wciąż zbieramy dane

Ponieważ lubię zmieniać nastroje tak, jak funkcja sinusoidy zmienia swoją wartość, ponownie zmieniłem temat.

Tym razem postanowiłam ułatwić rozmowę o rzeczach, które robimy, a które sprawiają, że są szczęśliwi. Było to inne pytanie niż pierwsze - dobre rzeczy, które robimy - ponieważ tutaj nie chodziło o dostarczanie, produktywność i generowanie wartości. Chodziło o więzi międzyludzkie i to, jak zachowujemy się wobec siebie, jak wchodzimy w interakcje. Pomyślałem, że byłoby miło, gdyby powiedzieli sobie nawzajem, co zauważają w zachowaniu innych i co doceniają.

Nie wiem jak Wy, ale ja lubię czasem poklepać się po plecach - znak, że ktoś docenia moje starania.

A więc:

> Co doceniasz, że my robimy? Nie w aspekcie produktywności, kodowania czy techniczności, ale po prostu jak człowiek - co sprawia, że chcesz z nami przebywać i spędzać nasz czas razem?

I ten, o rany, ten punkt był dość słodki szczerze mówiąc. Więc jeśli mieliśmy słodkie, to czas na gorzkie ponownie.

19.6.8 Przestań, to boli.

Teraz postanowiłem trochę zmącić wody. Wrzuciłem temat:

> Cele do poprawy. Co powinniśmy przestać robić? Co powinniśmy osiągnąć, żeby się poprawić?

I tu postanowiłem, że ta retrospektywa nie powinna zakończyć się tylko pustym gadaniem. Znaczy tak jasne, poprawiliśmy trochę więzi, poznaliśmy się trochę lepiej i mieliśmy blasta rozmawiając ze sobą (naprawdę!), ale dla mnie to za mało.

Podrzuciłam więc pewien pomysł.

Każdy, jeśli miał takie życzenie, mógł dobrowolnie wybrać jedno lub więcej zadań, swoich pain pointów, które chce rozwiązać do następnej retrospektywy.

Poza tym, musieli też wybrać ~~witchera~~, obserwatora dla swoich zadań. Może właściciel byłby tu lepszym słowem. Ta osoba będzie odpowiedzialna za sprawdzanie ich postępów i tak dalej. Zasadniczo ktoś, kto upewni się, że zadanie zostanie wykonane, poza tą osobą. Ktoś, kto będzie ich trochę nagabywał, np. co drugi dzień zadawał słynne, przynajmniej w naszej firmie, pytanie - Jaki jest status tutaj?

Dlaczego? O ile ufam moim kolegom z zespołu, że w pełni zrealizują rzeczy związane z projektem, to te zadania były z kategorii miłych do zrobienia i w ogóle, ale nie zginiemy, jeśli ich nie zrobimy. Nie były krytyczne, tylko miłe do zrobienia. Stąd dobrze by było mieć jakąś odpowiedzialność i dodatkową motywację do ich wykonania.

To też się udało - prawie każdy zdecydował się na jakieś zadania dla siebie.

19.6.9 Incepcja

Po tym przyszedł czas na małą incepcje. Retrospektywę na retrospektywę. W zasadzie krótka runda informacji zwrotnych dotyczących dzisiejszej retrospektywy, mojej facylitacji, wygenerowanej wartości i tak dalej. Podobno im się podobało, albo tak powiedziała. Tak.

To by było chyba na tyle. Po tym retrospektywa dobiegła końca. Przeżyłem i faktycznie uznałem ją za sensowną. Ale z drugiej strony dość wymagające.

Jestem młody, mam problem z moim ego, wiesz. Ciężko było być na uboczu, przez większość czasu tylko obserwować, być moderatorem, zamiast aktywnym uczestnikiem. Miałem kilka momentów, w których chciałem porzucić moją rolę moderatora i zaangażować się jako członek zespołu, zaczynając moje niekończące się gadanie, wrzucając swoje dwa grosze, ale powstrzymałem się od tego, udawało mi się pozostać w swojej roli.

To było trochę uczące pokory i satysfakcjonujące, szczerze mówiąc.

19.6.10 Podsumowanie

Żebyś wiedział, później te zadania, które zdefiniowaliśmy - zostały wykonane. Więc tak naprawdę nie skończyło się tylko na pustym gadaniu. Podobno ten pomysł ze strażnikami dla zadań też trochę pomógł - może się przyda i tobie? Nie mam pojęcia.

Więc cóż, procesy Agile też mogą być fajne i przydatne, to po pierwsze. Po drugie, wykaż się empatią i sprawdź drugą stronę płotu, jeśli masz jakąś konkretną opinię albo może się w czymś mocno z kimś nie zgadzasz, spróbuj zrozumieć. Może jest tam coś wartościowego, tylko o tym nie wiesz. Na koniec wyjdź ze swojej strefy komfortu. Dzięki temu będziesz się rozwijać. Coaching na całego. Kto ci ukradł marzenia?

Jeśli i Ty przygotowujesz się do swojego pierwszego retro, może ten artykuł będzie dla Ciebie pomocny. Ponadto: protip. Róbcie notatki. Ja tak zrobiłem. Potem, już po retro, zamieniłem je w ładną stronę confluence i zadania na Jira.

To by było na dzisiaj, ja, Olaf, byłem twoim skromnym gospodarzem, mam nadzieję, że ci się podobało.

20 Papierologia i pieniądze

Trochę o papierologii.

20.1 Netto, brutto, umowy, statusy, koszty

Mała notka dla mniej obeznanych: netto to kwota do ręki, brutto, to kwota przed częścią podatków, a całkowity koszt, jaki płaci za ciebie pracodawca to jeszcze zupełnie inna kwota w większości przypadków. Trochę to się inaczej ma podczas rozliczania na B2B, ale to już insza inszość. Opiszę to za chwilę.

Kolejna mała notka, fajny atut przy pierwszej pracy to status ucznia/studenta. Otóż jak szukasz pierwszej pracy albo gdzieś tam jeszcze zaczynasz, to dobrą rzeczą jest fakt, że prawdopodobnie masz poniżej 26 lat i możesz uzyskać lub też posiadasz, status ucznia/studenta.

Co to daje? Otóż jeśli masz do 26 lat i masz ten status, to jesteś ubezpieczony i zatrudniając Cię na umowę zlecenie, twój pracodawca nie musi płacić żadnych składek w zasadzie. Powoduje to, że całkowity koszt twojego zatrudnienia maleje i to znacznie, a ty przy tym masz ubezpieczenie. Podam ci przykład wzięty z mojego przypadku.

Zatrudniając mnie i płacąc mi przy tym 3024 zł 'do ręki' miesięcznie, czyli 3024 zł netto, to kwota brutto, która widnieje na umowie zleceniu, wynosiła 3533 zł. Z racji tego, że miałem status ucznia/studenta, to ta kwota brutto była też również finalnym kosztem pracodawcy.

Teraz porównajmy sytuację, kiedy pracujesz na umowie o pracę lub nie jesteś uczniem/studentem. Netto zarabiasz 3024 zł. Kwota brutto na umowie to już 4244,58 zł, a całkowity koszt twojego pracodawcy to aż 5119,46. Czyli 40% kosztów twojego zatrudnienia zjadają podatki, ubezpieczenia i inne daniny dla miłujących nas panujących włodarzy.

Szkoda, że to pracodawca to wszystko opłaca. Gdyby pracownik dostawał pełną kwotę 5100 do ręki i nagle zostawałoby mu 3024 zł po podatkach, to wydaje mi się, że po 10 następnego miesiąca nastąpiłaby w kraju rewolucja.

Jak widzisz, różnica jest ogromna. Opłaca się być uczniem/studentem. Opłaca się tobie i firmie. Dlaczego? Bo skoro ty potrafisz sobie to przeliczyć, to twój szef również. Zatem mając do wyboru porównywalnych kandydatów, jak sądzisz, kogo wybierze szef – sprytny przedsiębiorca? To raz.

Dwa, jeśli jest taka opcja, to możesz przecież się dogadać, że firma więcej ci zapłaci, ale wciąż będzie to mniej, niż mieliby na ciebie wydać, gdybyś pracował na ozusowanej umowie zleceniu lub umowie o pracę.

Czy to znaczy, że musisz studiować? Nie. Jest mnóstwo szkół, gdzie zapisując się, potem wystarczy pojawiać się raz w miesiącu na godzinę czy dwie, może trzy i tyle.

Są też takie placówki, gdzie jedyny czas, podczas którego spotykasz się ze swoimi koleżankami, kolegami z wykładów oraz z wykładowcami, to w swoim wyobrażeniu podczas robienia comiesięcznego przelewu.

Status ucznia/studenta gotowy. Nie musi być to uczelnia, może być to dowolna szkoła policealna czy coś, nie trzeba nawet mieć matury ;) Temat ciekawy, możesz się zainteresować, ale to tylko podobno, tak słyszałem. Ja do takich rzeczy absolutnie nie

namawiam, nigdy nie stosowałem. Broń Boże, gdzież bym śmiał. Jestem praworządnym obywatelem, który przestrzega prawa.

Czy wszyscy powinni olać umowę o pracę, przejść na umowę zlecenie i tak robić? Nie, każda forma zatrudnienia ma swoje wady i zalety, które zaraz omówię, których musisz być świadom, by podjąć decyzję najkorzystniejszą dla ciebie w danej sytuacji.

Bo jak to w ogóle z tymi umowami jest, jakie są między nimi różnice i które będzie dobra akurat dla ciebie? Już wyjaśniam.

Na ogół mamy trzy podstawowe formy współpracy na rynku między podmiotem prawnym (firmą) a osobą fizyczną (pracownikiem), są to umowa o pracę, umowa zlecenie i umowa o dzieło.

20.2 Umowa o dzieło

Tutaj szybko, najtańsza dla pracodawcy forma zatrudnienia i chyba najgorsza dla Ciebie. Umowa o dzieło jest umową, która dotyczy wykonania konkretnej pracy, konkretnego rezultatu, usługi.

Z tego też rezultatu możesz być rozliczany i o ile jest dobrze, to jest dobrze, ale jeśli trafisz na szefa Janusza, skopiesz robotę lub po prostu coś pójdzie grubo nie po twojej myśli, to może zrobić ci się gorąco pod stopami, bo twój szef ma prawo zwyczajnie zareklamować efekt twojej pracy i koniec. To raz.

Dwa, na umowie o dzieło nie obowiązują cię oskładkowania: Ubezpieczenie emerytalne, chorobowe, rentowe, wypadkowe i co najważniejsze, zdrowotne.

O ile ja jakoś zbytnio nie liczę na emeryturę z ZUSu, tak ubezpieczenie zdrowotne wolę jakieś mieć, bo w razie kłopotów wizyta w szpitalu potrafi być kosztowna. Bardzo.

Także przelicz sobie, czy kalkuje ci się ryzykować bez ubezpieczenia lub ewentualnie kupować prywatne, które obejmuje także opiekę szpitalną.

Płatne urlopy czy coś w tym stylu? Raczej zapomnij, chyba że dogadasz z pracodawcą nieoficjalnie. Chorobowe? Pf.

Moim zdaniem umowa o dzieło jest zatem nieciekawą opcją zatrudnienia dla programistów. Nie polecam nawet mimo tego, że nie jestem fanem oddawania daniny państwu. Nie warto w tym wypadku ryzykować.

20.3 Umowa zlecenie

Tu już kapkę lepiej. Zacznijmy od tego, że tutaj umowę rozlicza się nie na podstawie efektu twojej pracy, ale czynności, jakie podjąłeś, by ją wykonać. Czyli nawet jeśli efekt twojej pracy będzie niezadowalający, ale ty dołożyłeś wszystkich starań, by wykonać ją należycie, to jesteś w miarę bezpieczny. A przynajmniej bezpieczniejszy niż na UoD. Czy to znaczy, że możesz być partaczem i odwalać Manianę? NIE, ale zawsze warto mieć kolejną warstwę bezpieczeństwa.

Z urlopami jest podobnie jak przy UoD – jak sobie nie wynegocjujesz w umowie zapisu o płatnym urlopie, to go nie ma. Co z chorobowym? Zależy. Opiszę niżej.

Jeśli o składki chodzi, to mamy dwa przypadki. Pierwszy, gdy masz status ucznia/studenta i jesteś w wieku do 26 lat, drugi w pozostałych przypadkach. Przejdźmy najpierw do drugiego, bo jest prostszy.

Otóż jeśli nie masz statusu ucznia/studenta lub masz powyżej 26 lat, to w przypadku umowy zlecenia, twoja umowa podlega normalnemu oskładkowaniu, w miarę podobnie jak na UoP.

Jest tu jednak pewien element swobody – jeśli sobie życzysz, to pracodawca może zarejestrować cię jako dobrowolnego płatnika składek chorobowych i wtedy faktycznie, masz prawo do płatnych zwolnień, ale kosztem potrącenia kilku złotych z pensji co miesiąc.

W tym pierwszym przypadku zaś, kiedy masz status ucznia/studenta i jednocześnie masz do 26 roku życia, jak już mówiłem, jedynie podatek dochodowy trzeba odprowadzić, więc koszt pracodawcy jest podobny jak przy UoD.

Niestety w tym wypadku nie przysługuje ci jednak płatne chorobowe i nie masz prawa do niego dobrowolnie przystąpić.

Dość ważne – jeśli chodzi o status ucznia, to jest taki fajny myk, który polega na tym, że twój status ucznia nie kończy się z datą ukończenia szkoły a z początkiem następnego roku szkolnego.

Na przykładzie – ja, kończąc liceum w maju, status ucznia miałem do końca sierpnia i do końca sierpnia byłem zwolniony ze składek na UZ. Dopiero od 1 września ten status straciłem i od 1 września włącznie, obowiązywały mnie już normalne warunki.

Nieco inaczej jest jednak w przypadku studentów – tam dzień, w którym opuszczasz mury uczelni, lub zostajesz z niej wyrzucony, jest dniem, gdy kończy się twój status.

20.4 Umowa o pracę

Opcja najbezpieczniejsza, za to najbardziej kosztowna. W skrócie to oczywiste – wszystkie składki, emerytalne, chorobowe, zdrowotne i co tam jeszcze, płatne urlopy i w ogóle.

Jednocześnie opcja, której osobiście nie lubię. Dlaczego?

Bo potężne pieniądze oddaje się państwu. Tych pieniędzy raczej już nigdy nie zobaczysz, przynajmniej jeśli o emerytury chodzi. Kwoty wyliczałem już wyżej. Nie żartujmy – każdy, kto sądzi, że obecne pokolenia, gdzieś w moim wieku i okolicach, doczekają emerytury z ZUS, trochę śnią na jawie, to raz.

Dwa, jak już, to ta emerytura będzie śmiesznie niska, tak niska, że za to nie idzie godnie przeżyć. O ile sami się o nią nie zatroszczymy, to nie ma na co liczyć. Proste.

Sam ten fakt, że tyle pieniędzy jest pobierane z mojej ciężko wypracowanej pensji, budzi we mnie odrazę. Szkoda gadać.

Ma to jednak zaletę jedną – jeśli starasz się o kredyt na mieszkanie, to na UoP jest dużo łatwiej zazwyczaj. W innych przypadkach trzeba się trochę nagimnastykować.

Zwłaszcza na B2B. A jak już o B2B mówimy, to...

20.5 B2B

Teraz parę słów o tajemniczym B2B, czym jest, po co to komu, i dlaczego oraz analiza, kiedy warto przejść na tę formę współpracy. Poświęcam temu zagadnieniu cały rozdział, bo ogółem temat rzeka.

20.5.1 Czym jest B2B

Przeglądając ogłoszenia z ofertami pracy dla programistów, niejednokrotnie można natknąć się na widełki finansowe – to raczej standard w branży, by podawać zakres możliwej do uzyskania pensji.

Przy okazji jest też często podawana forma współpracy. Co raz, to częściej, można zaobserwować, że zamiast znanej wszystkim umowy o pracę, umowy zleceniu, czy też umowy o dzieło, pojawia się coś takiego jak B2B. Co to znaczy?

B2B to skrót od Business to Business – to forma współpracy, w której nawiązujesz stosunek z jakąś firmą nie jako pracownik a jako inna firma. Oznacza to po prostu, że zamiast zostać oficjalnym pracownikiem, stajesz się wykonawcą usługi dla danej firmy, musisz założyć swoją działalność gospodarczą i rozliczać się za pomocą faktur. Ponownie jednak – co to konkretnie znaczy i co to ze sobą niesie?

Zacznę od tego, dlaczego pracodawcy lubią B2B. Otóż sprawa jest prosta – po pierwsze, osoba 'zatrudniona' na B2B, jest prostsza do rozliczenia – jako pracodawca nie masz wtedy pracownika do rozliczenia i utrzymania, ale po prostu koszt – fakturę do opłacenia. Na tym kończy się odpowiedzialność pracodawcy w przypadku B2B.

B2B pozwala też zmniejszyć koszta zatrudnienia. Jak już wcześniej wyliczałem, przy umowie o pracę, kwoty, jakie co miesiąc oddaje się państwu, są przerażające. Wydajesz ponad 5 tysięcy, pracownik dostaje z tego 3. A to same podatki, do tego przecież jeszcze trzeba doliczyć koszta obsługi kadr.

O to, by odpowiednie dokumenty zostały złożone, przelewy wysłane a składki potrącone, dba przy UoP, nasz pracodawca. Te wszystkie czynności zajmują czas i kosztują, generując dodatkowe wydatki.

Dodatkowo przy B2B nie obowiązuje Kodeks Pracy. To ogromna ilość obostrzeń i regulacji, które chronią pracownika, do których musi stosować się pracodawca. W relacji B2B?

Nic takiego nie istnieje.

B2B zdejmuje zatem dużo obowiązków z barków pracodawcy i przerzuca je na twoje. To ty musisz zarejestrować sobie działalność, potem regularnie płacić VAT, ZUS, dochodowy, wystawiać faktury, przechowywać dokumentację i tak dalej.

Możesz zapytać zatem – po co w ogóle ktoś by się na coś takiego godził? Cóż, z większą odpowiedzialnością przychodzą też pewne benefity.

Zanim je wymienię, to zaznaczę, że B2B nie jest formą współpracy odpowiednią dla każdego. Postaram się w miarę rzetelnie opisać wady i zalety tej współpracy, chociaż może być trudno, bo ja akurat jestem fanem i zwolennikiem. Do rzeczy.

20.5.2 Zalety B2B

Bycie na B2B, czyli tak zwanym samozatrudnieniu, niesie ze sobą szereg odpowiedzialności, ale jednocześnie daje pewne możliwości, których nie mamy, pracując w oparciu, o inne formy współpracy. Jakie to zalety? Ja wymienię dwie, moim zdaniem, najważniejsze.

20.5.2.1 Mniej podatków Chyba moja ulubiona zaleta. Wyliczenia dotyczące tego, jak dużo procent naszej pensji wędruje do organów państwa, już sobie liczyliśmy. Dla przypomnienia: zarabiając 5000 zł netto, czyli na rękę, przy umowie o pracę, pracodawca

musi zapłacić za nas 8530 zł. Sporo, co? Te 3,5 tysiąca zjadane jest przez różnego rodzaju podatki. Do tego opłać jeszcze osoby, co kadrami zarządzają. Dramat.

I tu wchodzi B2B, całe na biało, które pozwala nam te podatki trochę zminimalizować. Jak bardzo? Już liczymy.

Rozpatrzę trzy sytuacje: kiedy założymy działalność i korzystamy z ulgi na start, kiedy założymy działalność, ale korzystamy tylko z małego ZUS oraz dla normalnego ZUS.

W pierwszym przypadku, przez 6 miesięcy płacimy jedynie składki na ubezpieczenie zdrowotne, czyli około 500 zł miesięcznie.

Do tego dochodowy, zależnie od tego, jaki wybierzemy, ale dla prostoty założymy, że wybraliśmy zasady ogólne i łapiemy się do 1 progu podatkowego, zatem 12%.

VAT tutaj pominiemy i będziemy rozmawiać o kwocie netto, bez VAT, ponieważ jeśli firma pracodawcy cokolwiek na siebie zarabia, to VAT przy pensjach jest tak naprawdę tylko czymś, co sobie przerzucamy wzajemnie.

Zakładając, że na B2B dogadamy się na jakieś 8000 zł netto, czyli jakieś 500 zł mniej niż normalnie by zapłacił za nas pracodawca na UoP, do ręki zostanie nam wtedy: kwota netto na fakturze - podatek dochodowy - zus, czyli: 8000 zł - 8000 zł * 12% - zus = 8000 zł - ~900 zł - ~500 zł = 6700 zł.

Od 2022 w IT mamy 12% ryczałt albo np. 19% liniowy. Załóżmy liniowy.

Ponad 1700 zł więcej. A ta kwota może jeszcze zostać znacznie zmniejszona, jeśli masz jakieś koszta. Jest różnica? Jest.

Żeby nie było, iż wyciągam kwoty nie wiadomo skąd, to podaję przykład tego, jak pracodawcy sobie przeliczają kwoty na poszczególnych umowach, który wziąłem w ogłoszenia jednej z popularniejszych firm:

```
Widełki:
6k - 10k netto b2b
3,5k - 5k netto UoP
```

Także ja tutaj w zasadzie jeszcze nieco zaniżyłem kwotę, jaką możemy uzyskać na B2B, bo tu 5k netto na UoP, odpowiada 10k netto na B2B.

Im większe twoje przychody, tym większa ta różnica będzie.

Żeby nie było – to tutaj zakładam sytuację, kiedy przysługuje Ci ulga na start lub chociaż mały ZUS. W innym przypadku, kiedy musisz od razu płacić duży ZUS, to powiedzmy, że B2B stanie się opłacalne dopiero gdzieś przy 9-10k netto na fakturze. Wcześniej nie masz co o tym myśleć.

A kiedy ci nie przysługuje mały ZUS lub ulga? W momencie, gdy prowadziłeś działalność w ciągu ostatnich 5 lat lub gdy będziesz świadczył usługi na rzecz swojego pracodawcy. Uwaga! Umowa zlecenie/o dzieło to, w większości przypadków, nie jest stosunek pracy – liczą się tylko te firmy, dla których pracowałeś lub pracowałaś na umowę o pracę.

W każdym razie. Jak widać kwoty, jakie oddajemy państwu, są przerażające. Ja na przykład nie lubię za to, by moje pieniądze szły w głównej mierze na rzecz różnych danin i podatków. Wolę przytulić je do swojej kieszeni. Poza tym jest to sprawa nie tylko korzyści finansowej, ale również, jak dla mnie, idei. Ot co.

20.5.2.2 Koszta uzyskania przychodu Następnym punktem, który przemawia za tym, by przejść na B2B, są tak zwane 'koszta uzyskania przychodu.' Wspominałem już o nich wcześniej. Cóż to takiego? To w zasadzie przedłużenie punktu poprzedniego niejako.

Otóż ty, jako przedsiębiorca, by prowadzić swoje przedsiębiorstwo, potrzebujesz nierzadko różnych rzeczy. W naszym przypadku, jako programistów, będą to chociażby różnoraki sprzęt cyfrowy, materiały szkoleniowe w postaci książek, e-booków, subskrypcji platform learningowych, samochód, który jest nam niezbędny, by dojechać do klienta (firmy). I tak dalej.

Wszystkie te koszta, które jesteśmy w stanie wykazać, że były nam potrzebne, by uzyskać dochód, możemy uznać za koszta prowadzenia naszej działalności. Co za tym idzie, możemy je 'odliczyć od podatku'. Co to znaczy?

W skrócie, to tyle, że od rzeczy kupowanych na firmę możesz sobie odliczyć praktycznie 1/3 ceny (jeśli jesteś VATowcem w innym przypadku około 18 procent), natomiast te pozostałe 2/3 z ceny, to będzie twój rzeczywisty koszt, jaki poniesiesz za nabycie danej rzeczy. Brzmi fajnie, co?

O tym, jakie mogą być koszta, można by książkę oddzielną napisać, bo kreatywność ludzka nie zna granic. Doprawdy.

Skąd wynika ten mechanizm? Z dwóch rzeczy. Po pierwsze, jeśli wystawiasz faktury VAT, to musisz do kwoty netto, która się na nich znajduje, doliczyć podatek VAT. Powstanie wtedy zobowiązanie wobec urzędu skarbowego na określoną kwotę podatku VAT. W momencie, kiedy ponosisz wydatek w celu uzyskania przychodu, ustawodawca przewidział możliwość odliczenia podatku, który sprzedawca zapłacił. Jedną kwotę od drugiej się odejmuje, żebyś nie płacił dwa razy. Powoduje to, że ty finalnie zapłacisz mniej VAT na koniec miesiąca.

Podobnie jest z dochodowym – należną kwotę podatku liczymy od dochodu, a ten z kolei liczymy ze wzoru: przychód – koszta uzyskania dochodu bez VAT = dochód.

Stąd też ta redukcja.

Oczywiście to niesie ze sobą różne wymogi i obowiązki, ale często warto.

Te punkty, to dwie najważniejsze zalety B2B. Czyli tak w skrócie, najważniejszą zaletą B2B jest... kasa. Oprócz tego - wolność i niezależność. To wszystko. I to dlatego ryzykujemy. Czym i jak? Już wyjaśniam.

20.5.3 Wady

A tych trochę jest. Dlatego warto przemyśleć, czy B2B jest wyborem dla nas – nie dla każdego się nadaje.

20.5.3.1 Odpowiedzialność na twojej głowie Pamiętasz, co wspominałem wcześniej? To o składkach, podatkach, urzędach? Przy innych formach współpracy, wszystko to jest na głowie twojego pracodawcy. Ciebie zupełnie nic nie interesuje, poza sprawdzeniem, czy przelew już jest.

W momencie, kiedy przejdziesz na B2B, wszystkie obowiązki formalne, spadają na ciebie. Twój pracodawca jedynie musi pamiętać o tym, by opłacić fakturę, którą mu wystawisz.

Co takiego zatem trzeba robić, jako przedsiębiorca? Trochę tego jest. Pierwsza rzecz to opłacenie składek ZUS. Następna – opłacenie zaliczek na PIT – raz na miesiąc lub raz na kwartał, zależnie jak wybierzesz.

Dalej, jeśli jesteś vatowcem, opłacenie VAT i wysłanie stosownej, comiesięcznej (później zyskasz możliwość płacenia co kwartał, na początku nie ma takiej opcji) deklaracji. Potem jeszcze JPK_VAT, czyli Jednolity Plik Kontrolny. Też trzeba wysłać.

Zazwyczaj dochodzi też prowadzenie KPiR.

Co miesiąc należy też pracodawcy wystawić fakturę za usługi i ją wysłać.

Brzmi, jakby było tego sporo, prawda? Cóż, po pierwsze praktycznie wszystko z tej listy, poza wystawianiem faktury dla pracodawcy, możesz zautomatyzować za jakieś ~100 zł miesięcznie, poprzez zainwestowanie w księgową, która zrobi to za ciebie.

Ewentualnie, można robić to samodzielnie. Osobiście ja tę opcję wybrałem. Dlaczego?

To tylko brzmi skomplikowanie, a tak naprawdę to wszystko ogranicza się do zrobienia przelewu lub kliknięcia 'wyślij deklaracje' w darmowym programie księgowym – resztę robią za nas programy.

Cała moja księgowość zajmuje mi może 10 minut miesięcznie, także szkoda by mi było płacić komuś za coś tak prostego.

20.5.3.2 Brak ochrony Na B2B wszystkie chwyty są dozwolone, czy też raczej powinienem powiedzieć, wszystko zależy od tego, co masz w umowie. A w niej mogą być rzeczy różne – począwszy od braku okresu wypowiedzenia, skończywszy na odpowiedzialności za ewentualne błędy, jakie popełnisz w pracy i szkody, jakie one spowodują.

A będąc osobą, która prowadzi jednoosobową działalność gospodarczą, za swoje błędy odpowiadasz całym swoim majątkiem.

Nie jesteś też chroniony czy chroniona przed chorobą. Nie pracujesz? Nie ma kasy. Przynajmniej domyślnie, bo w umowie, jeśli pracodawca na to pozwoli, można zawrzeć opcję płatnego 'chorobowego', nawet na B2B.

Podobnie z urlopem. Tutaj nie ma płatnego. Nie świadczysz usługi? Nie ma kasy.

Znowu – przynajmniej domyślnie. Zazwyczaj ta kwestia jest rozwiązywana poprzez doliczenie do stawki miesięcznej określonej kwoty, która jest proporcjonalnym ekwiwalentem tego, ile tracisz, jeśli pójdziesz na standardowy urlop przez 20, czy 26 dni w roku. Jak to wygląda?

Załóżmy, że jesteś młodym człowiekiem, tak jak ja, przysługuje ci więc tylko 20 dni urlopu w roku, gdybyś pracował na umowę o pracę.

Załóżmy dalej, że twoja stawka godzinowa wynosi 50 zł netto, czyli 8*50 = 400 zł dziennie.

Liczymy dalej, 400 zł dziennie, 20 dni urlopu, 20 dni * 400 zł/dzień = 8000 zł. A zatem, do stawki godzinowej należy doliczyć: 8000 zł / (12 miesięcy * 168 godzin/miesiąc) = 3,96 zł/h, zaokrąglając, 4 zł/h.

Do tego doliczamy sobie podatek, który musimy od tych 4 zł odprowadzić, czyli, zaokrąglając, 20%, czyli: 4 zł/h + 4 zł/h * 20% = 4 zł/h + 0,8 zł/h = 4,8 zł/h, czyli w zasadzie 5 zł/h.

Czyli, jeśli docelowo chcę zarobić 50 zł netto, to bez urlopów płatnych uwzględnionych w umowie, powinienem zażądać przynajmniej 55 zł/h.

Ja jednak, dla przykładu, wolę nie kombinować w ten sposób, i ustalać sobie z pracodawcą normalny płatny urlop, czy też, jak to oficjalnie w umowie się zapisuje, płatne dni przerwy w świadczeniu usług. Niektórzy pracodawcy dają 20, inni 26, jeszcze inni 30. Wszystko zależy od tego, co jest w umowie – standard.

W każdym razie. Dobitnie pokazuje to jedną rzecz – jak sobie pościelesz, tak się wyśpisz. Na B2B to powiedzenie jest jeszcze bardziej prawdziwe.

Z drugiej strony nie jest też tak, że każdy tylko czeka, aż popełnisz jakąś pomyłkę czy zrobisz jakiś błąd, żeby cię pozwać, naciągnąć na koszta. Nie.

20.5.3.3 Ciężej o kredyt Kolejną wadą B2B jest to, że trochę ciężej o kredyt, zwłaszcza hipoteczny. Przedsiębiorcy są zazwyczaj obłożeni różnego rodzaju obostrzeniami, jeśli chcą wziąć hipotekę.

Pierwszym z obostrzeń, na które się natkniemy, to wymagany okres prowadzenia działalności. Praktycznie każdy bank wymaga tego, byśmy prowadzili swoją działalność gospodarczą nieprzerwanie przez przynajmniej 12 miesięcy, a w dużej ich części, ten okres wynosi 24 miesiące.

Zatem jest to coś ważnego – jeśli chcesz w ciągu następnych kilku miesięcy brać kredyt na mieszkanie, definitywnie nie przechodź na B2B.

W zasadzie to nawet jakikolwiek kredyt. Serio. Podam przykład z mBanku. Mając firmę krócej niż 12 miesięcy, nie dostaniecie absolutnie ŻADNEGO kredytu. Z automatu banku odrzuca taki wniosek. Z ciekawości sprawdziłem. Zawnioskowałem o najmniejszą możliwą kwotę – 500 zł.

Przy dochodach powyżej 10 000 zł netto i moich wydatkach, które są iście studenckie, bo koszta stałe oscylują mi w okolicach 1000 zł, mój wniosek został odrzucony – tylko dlatego, że w chwili wnioskowania działalność prowadziłem krócej niż X miesięcy.

Dla porównania, mając UZ i jakieś 5-6k netto, bez zająknięcia oferowano mi kredyt 30k.

Czasem wymagany jest też nieco większy wkład własny – to ma być gwarancja dla banku, która pokazuje, że faktycznie jesteś w stanie dostarczać pieniądze i spłacać swój kredyt, bo przedsiębiorca to w końcu większe ryzyko.

Dodatkowo hipoteka na B2B to więcej papierologii. Bank zazwyczaj sobie życzy, by dostarczyć poświadczenia od US, ZUS, że nie zalegamy ze składkami czy podatkami, do tego Księga Przychodów i Rozchodów za milion lat wstecz, którą prowadzimy, jakieś zeznania podatkowe. Naprawdę różne rzeczy.

Nie jest jednak tak źle, przynajmniej w naszej branży, bo niektóre banki, widząc, że dla programistów B2B jest tak naprawdę inną formą zatrudnienia, a nie działalnością z prawdziwego zdarzenia, czasem idą nam na rękę i traktują ulgowo. Jest to jednak raczej wyjątek od reguły i rzadko takie preferencyjne traktowanie się zdarza.

Nie demonizujmy tutaj jednak całego procesu z drugiej strony, bo to też głównie formalności, zwłaszcza jeśli bierzemy kredyt w tym samym banku, w którym mamy konto firmowe.

20.5.4 Podsumowanie

Podsumowując, B2B nie jest formą zatrudnienia dla każdego. Ma swoje wady takie jak to, że odpowiedzialność za różne rzeczy spada na twoją głowę, ale za to i zalety, bo z większą

odpowiedzialnością, przychodzi większa władza. W tym aspekcie akurat nad pieniędzmi, jakie wydaje na ciebie pracodawca.

Czy to dobra forma dla ciebie? Nie wiem. Prawie jednak na pewno powiem jedno, że jeśli masz mniej niż, powiedzmy 6, a najlepiej 12, miesięcy stażu, a kwota, jaką proponuje potencjalny pracodawca, opiewa na mniej niż 6-7k netto, to raczej daj sobie spokój i zaczekaj.

Poza tym masz do wyboru UoD, UZ, UoP. Każda ma swoje wady i zalety, które ci przedstawiłem nieco wyżej.

Imo na początku, jeśli masz poniżej 26. lat i dopiero zaczynasz, to nie ma żadnej dyskusji. Ogarniaj status ucznia/studenta i ciśnij UZ.

Jeśli masz powyżej 26. lat i dopiero zaczynasz, zarabiasz poniżej 85k rocznie wal UoP.

Jeśli zarabiasz nieco więcej już, czyli powyżej 100k, zakładaj JDG, zwłaszcza jeśli przysługuje ci Ulga na Start/Mały zus.

Gdy nieco więcej już, np ćwierć bani rocznie, pomyśl nad spółką z o. o. albo coś. Chociaż to raczej nie ten target, tutaj warto zainwestować w doradcę podatkowego.

20.6 Pieniądze

O bardzo ważnej rzeczy, jaką są pieniądze. Bez wstydu i otwarcie. Zacznijmy od czegoś, o czym wspomniałem, w innym rozdziale.

20.6.1 Jak rozmawiać o pensji

Często widzę, że wstydzimy się rozmawiać o pieniądzach. Cóż, moim zdaniem to trochę błąd. Chciwość ludzka rzecz, ale tylko jeśli opatrzona jest pewnymi zasadami i regułami.

Jeżeli to jakaś twoja pierwsza praca, a to sugeruje fakt, że czytasz tę książkę, to cóż, szczerze mówiąc, pieniądze nie są zbyt ważne. W pierwszej pracy tak naprawdę jedyne co robisz to... Uczysz się.

Na początku, przez pierwszych kilka dni/tygodni, nawet ktoś z 10-letnim stażem przynosi straty, bo musi zostać wdrożony do projektu, poznać zespół i tak dalej.

I to jest naturalne. Zatrudniając Juniora, każdy pracodawca musi być świadom tego, że trzeba będzie w niego nieco najpierw zainwestować, żeby potem zaczął być przydatny i przynosić zyski. Zatem jesteśmy świadomi tego, że ten okres wdrażania będzie dłuższy.

Także nie miej wyrzutów, że tylko się uczysz w pracy i to złe, nic bardziej mylnego.

To przemyślany ruch każdej firmy, ty jako pracownik, który się doszkala, wzrastasz na wartości i z czasem, kiedy to już osiągniesz pewien poziom, bardzo szybko zwróci się ta inwestycja w ciebie poprzez pracę, jaką wykonasz.

Bądźmy szczerzy, twój pracodawca musi na czymś zarabiać i tyle. Zarabia może na własnym produkcie, może na outsourcingu, w każdym razie, jakby nie patrzeć, to pośrednio zarabia na tobie i twoich umiejętnościach. Czy to złe? Nie, normalna rzecz.

Niemniej jednak proszę cię, nigdy nie wiń się za to, że czas w pracy spędzasz na nauce, zwłaszcza na początku, to po prostu, jak już wspomniałem, inwestycja pracodawcy w ciebie, która jest dla niego, prawdę mówiąc, dość opłacalna w dłuższej perspektywie czasu.

Nie żyjemy w świecie, gdzie ludzie są bezinteresowni, ale czy to źle? W tym wypadku uważam, że sprawdza się takie powiedzenie - 'wilk syty i owca cała'. Trzeba być wdzięcznym

za to, że ktoś dał ci szansę, ale i również świadomym, że nie jest to altruizm w czystej postaci, a raczej kalkulacja nastawiona na zysk i to kompletnie normalne, nic złego.

Mało tego, powiem jeszcze jedno, jeśli spotkasz kiedyś się w pracy z czymś takim, że każą ci uczyć się jedynie po godzinach, lub nauka rzeczy potrzebnych do wykonania nowego projektu czy czegoś podobnego, nie jest wliczana w godziny pracy, to szczerze, nieciekawa sprawa.

To może sugerować, że trafiłeś raczej do jakiegoś Januszsoftu, gdzie szef to typowy Nosacz. Warto zatem pomyśleć o zmianie pracy. Chociaż jak to mówią, pracy najłatwiej szukać, kiedy jakąś się już ma.

Podsumowując, jeśli dopiero zaczynasz, to stawka nie jest ważna. Czy to jednak znaczy, że masz pracować za przysłowiową miskę ryżu? Absolutnie nie. Trzeba się szanować i poniżej pewnego poziomu jednak nie warto schodzić. Proste. Jaki to jest poziom?

Trudno ocenić, dużo zależy od tego, co potrafisz, ale zakładając, że czytasz tę książkę, to wydaje mi się, że chyba w miarę ambitny z ciebie człek i coś tam będziesz umieć za jakiś czas.

Wyznaczając swoją stawkę, warto brać pod uwagę region, w którym mieszkasz, inne ogłoszenia i widełki, jakie oferują. Zrób mały wywiad. Jeśli krzykniesz za dużo, ale jesteś fajnym kandydatem, to raczej firma będzie chciała renegocjować, a nie od razu cię odrzucić, także nie bój się zbytnio, ale z drugiej strony nie rzucaj kwot z kosmosu, bo to też coś o tobie wtedy świadczy. Zakładanie, że pouczysz się pół roku i cyk #programista15k, jest tak głupie, że nawet nie zamierzam o tym dywagować.

Ponownie, zdrowy rozsądek cię ratuje, chociaż wbij sobie do głowy mocno, przy pierwszej pracy kasa nie jest aż tak ważna. Na pewno jednak nie bądź, jak niektórzy desperaci, co to się deklarują, że nawet pół roku będą pracować za darmo, byle tylko złapać się pazurami jakiejś roboty w IT. Tak, są takie osobniki.

20.6.2 Podwyżka != więcej na wypłatę, o czasie

Teraz kilka słów o tym, że podwyżka nie zawsze oznacza większej kwoty na wypłatę. Dlaczego?

Otóż sprawa jest prosta – musimy zdać sobie sprawę z tego, że my tak w zasadzie sprzedajemy swojemu pracodawcy swój czas i względem niego trzeba przeliczać naszą wypłatę. Nie zawsze uda ci się wynegocjować podwyżkę w postaci monetarnej, bo przykładowo firma nie ma dodatkowych środków, na kogoś o twoim doświadczeniu przewidziana jest konkretnie taka stawka, czy coś podobnego. Możesz wtedy uderzyć pod innym kątem. Jakim?

Chyba najprostszy przykład to redukcja etatu – zamiast pracować 5 dni w tygodniu, poproś o 4. Uda ci się? Właśnie zredukowałeś swój etat do 4/5, zachowując tę samą pensję, czyli efektywnie rzecz biorąc to niejako 20% podwyżki za godzinę twojego czasu. Jak dla mnie fajny wynik.

Innym sposobem na podwyżkę bez podwyżki, może być przejście na tryb pracy zdalnej. Od razu zaznaczam – nie jest to dla każdego i nie w każdej firmie da się taki model pracy zrealizować, o czym jeszcze napiszę w innym rozdziale.

Jednakże jeśli praca zdalna ci odpowiada, w firmie nie ma przeszkód i uda Ci się to wynegocjować – świetnie. Dlaczego?

Bo czas, który poświęcasz na pracę, to nie tylko czas, jaki siedzisz w biurze. Musisz sobie kalkulować również dojazdy. A na tych, często spędzamy bardzo dużo czasu i czasami sprawiają one, że nasza rzeczywista stawka maleje średnio o jakieś 10%, zakładając, że do pracy mamy około 30 minut.

W przypadku pracy zdalnej nie ma tego problemu – czas, jaki spędzasz na dojazdach, wynosi okrągłe 0. Chyba, że do jakiegoś coworkingu chodzisz, ale to wtedy z wyboru, nie z przymusu. Twoja wola.

Czas jest dość, ważny, zatem pomówmy o nim.

Jedyne co w życiu mamy, to czas. Musimy nim mądrze dysponować, bo czasu sobie nie kupimy, nie jesteśmy w stanie sobie go dodać czy sprawić, by biegł wolniej. Każdego dnia nam umyka. #górskiCoelho

Programowanie, to branża, którą można się zachłysnąć, że tak powiem. Relatywnie szybko jesteś w stanie dojść tu do naprawdę fajnych zarobków, zwłaszcza jak na warunki polskie, gdzie sporo osób zarabia minimalną czy tam 2k netto. Wiem to po sobie. Jeszcze w 2017 roku moim głównym zarobkiem była praca fizyczna na budowie. Na początku za całe 9 zł/h, potem 10, by finalnie skończyć z zawrotną stawką 11 zł/h. Nagle bum, rok później zarabiam kilka razy tyle.

To sprawia, że można się nieco 'zafixować' na tych pieniądzach kapkę i zacząć pracować dużo. Za dużo. A to nie one są najważniejsze przecież. Wiem, wiem. Frazes. Co ty wiesz o życiu dzieciaku?

Tak niemniej jednak jest. A my jako programiści handlujemy właśnie naszym czasem zazwyczaj – sprzedajemy pracodawcy godziny swojego życia w zamian za pieniądze. Dlatego znaj ich wartość. Nie daj się opętać pędowi ku kasie, który sprawi, że będziesz pracować po 14 godzin dziennie, bo więcej zarobisz.

Nie warto. Chyba że sprawia ci to przyjemność, nie masz rodziny, a twoje życie społeczne nie istnieje. Wtedy możesz.

Pytanie jednak inne – jak długo jesteś w stanie wytrzymać psychicznie i fizycznie?

20.7 Mój przypadek

Opowiem wam troszeczkę, jak wyglądały moje kolejne kroki na drodze programisty, jeśli o pensje chodzi.

20.7.1 Moje pierwsze zarobki, pierwsza podwyżka

Jak wyglądało to w moim przypadku? Nie wstydzę się mówić o pieniądzach, zatem pozwól, że podam ci kwoty z moich początków programowania.

Pracowałem zdalnie, mieszkając w małej miejscowości, dla firmy z większego miasta. To bardzo rzadko spotykana sytuacja, czyli praca zdalna, kiedy nie miałem żadnego etatowego doświadczenia. Nie spotkałem jeszcze drugiej takiej osoby, która by tak zaczynała, ale mi się jakoś udało. W każdym razie.

Swoją pierwszą pracę podjąłem, zaczynając od stawki 18 zł na godzinę, co w przeliczeniu na 168 godzin roboczych w miesiącu, czyli pełen etat, dawało 3024 zł do ręki. Różnie z tym jednak bywało, bo aż tylu godzin w miesiącu na początku nie robiłem. Listę tego, co umiałem, idąc do pierwszej pracy, opisałem później.

Pierwszą umowę podpisałem na dwumiesięczny okres próbny. Było to w listopadzie 2017 roku. Po jego zakończeniu umowę mi przedłużono z nową stawką, wynoszącą 22 zł na godzinę, netto, czyli w zasadzie 3700 zł na miesiąc. 23% podwyżki po dwóch miesiącach. I to była stawka, którą zaproponował mi szef, spróbowałem wtedy wywalczyć więcej, chciałem rzucić się na 25 zł netto.

Niestety nie udało mi się tyle wynegocjować, ale nie spowodowało to tego, że krzywo na mnie ktoś popatrzył, czy mnie zwolnił. Absolutnie. Zwyczajnie spotkałem się z odmową i obietnicą, że do tematu wrócimy.

Drugą sprawą było to, że nie zabiegałem jakoś mocno o tę podwyżkę, więc to inna sprawa, bo w tamtym czasie to w zasadzie uczyłem się jedynie, a moja efektywność w projekcie była niska.

Jeszcze kolejną sprawą jest to, że dostałem w miarę sporo podwyżki procentowo, bo 23%, natomiast tutaj zaproponowałem 40%, także patrząc na to z perspektywy czasu, to rzuca mi się na usta małe 'XD'.

20.7.2 Następna umowa i B2B

Następna podwyżka przyszła w sierpniu – po 8 miesiącach od podjęcia pracy, kiedy to podpisałem umowę z nową firmą – tutaj moja stawka urosła już do 37,50 zł/h netto, czyli 6300 zł przy 168 godzinach miesięcznie. Tu podwyżka była naprawdę duża, bo prawie 60%. Jak to mówią – najprostszym sposobem na podwyżkę jest zmiana pracy. Trudno się nie zgodzić.

Po miesiącu, z racji tego, że skończył mi się status ucznia/studenta, postanowiłem przejść na inną formę zatrudnienia – B2B.

Tu nastąpiła zmiana warunków, ale stawka została +/- ta sama, z tym że zależnie od tego, jak zrobię podatki, to nieco potrafiła ona wzrosnąć np. do ~40 zł na godzinę, na czysto, do ręki. Wspominam, że do ręki, bo kwota netto na B2B, to nie jest kwota, która nam zostaje :) Na B2B, kwota do ręki powinna się nazywać netto kwoty netto, albo coś.

Następnie, umowa z 3 firmą, którą podpisałem po 3 miesiącach w poprzedniej firmie, opiewała ponownie na taką samą stawkę mniej więcej, bardziej więcej niż mniej, jednakże zagadnienia, przy jakich pracowałem, były o wiele bardziej rozwojowe i wymagające, pieniądze to nie wszystko.

Poza tym z drugą firmą się nie rozstałem, ale przeszedłem na zdalny tryb pracy, robiąc dla nich dodatkowo po godzinach. Moje miesięczne zarobki po przejściu na B2B (czyli ~9 miesięcy doświadczenia) nie schodziły poniżej 10 000 zł netto+VAT, z tego do ręki, przy kreatywnej księgowości, zostawało mi około 8 000.

Czasem trafiało się i 12 000 zł netto, czyli tak+/- 10 000 na czysto. Kwota zależała od miesiąca i tego, ile godzin przepracowałem.

Nigdy jednak nie dochodziło do takich sytuacji, bym przepracował, nie wiem, 300 godzin.

20.7.3 Dalsze losy

Tak spędziłem następne 3 miesiące – pracując na dwa fronty. Potem zrezygnowałem z drugiego etatu, zostałem w jednej pracy, stacjonarnej. W sumie to nie zrezygnowałem, a

hajs z funduszy unijnych się prezesowi skończył. Znaczy hajs na developerów. Zawinął co swoje i zniknął xD

Tam zabawiłem jeszcze przez następne 3 miesiące. W międzyczasie udało mi się znaleźć źródło zleceń dla zagranicznych klientów. Tutaj stawka skoczyła mi już do 60 zł/h netto. Nie było tego za dużo, ale trochę się trafiało.

Trzy miesiące minęły od tamtego momentu, moja umowa z obecną firmą dobiega końca – podpisaliśmy ją początkowo na 6 miesięcy, jako taki swego rodzaju okres próbny, gdyż ogółem naprawdę firma zajmowała się ambitnymi rzeczami i trzeba było zobaczyć, czy dam radę. Cóż, raczej byli ze mnie zadowoleni. Otrzymałem ofertę przedłużenia umowy z nową stawką.

Nowa stawka, którą mi zaproponowano, to 10 000 zł netto miesięcznie, czyli równo 60 zł/h w zasadzie, przy średnio 168 h miesięcznie, z umową, że po 3-4 miesiącach podniesiemy to do 12 500 zł netto plus umowa na czas nieokreślony.

Jednocześnie tak się akurat złożyło, że miałem zupełnie inną ofertę, z innego miejsca. Tam od razu oferowano mi 75 zł/h, czyli jakieś 12 500 zł/netto za miesiąc pracy.

Dni urlopowe tak samo, wszystko inne w zasadzie podobnie.

I w tym momencie trochę się czułem, jakbym złapał Pana Boga za nogi. Nie zrozumcie mnie źle, wcześniej też bez problemu zarabiałem 10, czy 12k miesięcznie, niemniej jednak było to okupione tym, że robiłem sporo nadgodzin. Tutaj natomiast obie stawki dotyczyły podstawowego etatu. Byłem naprawdę zadowolony i ucieszony.

Żadnej z tych ofert nie przejąłem, a z obecnej firmy odszedłem.

Co? Tak.

20.7.4 Odejście

Robiłem to z ciężkim sercem tak naprawdę. Trochę zżyłem się z osobami w firmie, zespołem. W zasadzie to po raz pierwszy w życiu tak się przywiązałem do firmy. W poprzednich tak nie miałem zupełnie.

Kurczę, fajnie było. Super biuro w samym centrum, 15 minut drogi ode mnie komunikacją miejską, inicjatywy, integracje, sprzęt z górnej półki na start, pakiety socjalne, rozwój, firma, która na pracowników nie szczędziła i dbała o to, co jej. Pewnie, czasami bywało ciężko, bo tematy do ogarnięcia w zespole nie były najłatwiejsze, ale ogółem do biura zawsze szedłem bardzo chętnie. Wspaniała społeczność. Odkładając zupełnie projekt czy sprawy technologiczne na bok, nic lepszego sobie wymarzyć nie mogłem, niż to, co miałem tam. Pozdrawiam z tego miejsca serdecznie wszystkich z zespołu Synerise.

A jednak odszedłem. Dlaczego?

20.7.5 Nowa okazja

Otóż pojawiła się przede mną okazja. Okazja wzięcia udziału we współtworzeniu pewnego przedsięwzięcia. Bardzo fajnego przedsięwzięcia, w które wierzę mocno. Z naprawdę łebskimi i ciekawymi ludźmi. Nie zastanawiałem się zatem długo.

I tak oto zacząłem robić to, co robię teraz.

Podsumowując: otrzymałem ofertę 86 zł netto/h, praca kompletnie zdalna. Czyli, jeśli jest miesiąc, który ma 22 dni robocze, oficjalnie dołączałem do grupy #programista15k.

Co prawda nie na UoP, tylko na B2B, ale wciąż. Ogółem bardzo się z tego cieszę i czuję bardzo doceniony finansowo.

Tutaj ponownie wychodzi moje szczęście w życiu i to, że poznaję ludzi, którzy mi pomagają.

Powiem wam jedno, świat IT w Polsce, a w Warszawie to już w ogóle, jest bardzo, bardzo mały. Nawiązując dobre relacje, zostawiając po sobie dobrą opinię i będąc ogółem po prostu dobrym człowiekiem, można sporo zyskać. Do mnie ta okazja spadła właśnie przez coś takiego. Powiem szczerze, cieszę się bardzo.

Tym bardziej, że robota zdalna. Jakby tego było mało, to naprawdę ogarnięty zespół. Szczerze to trochę mi się wydaje, że na taką kasę nie zasługuję, biorąc pod uwagę moje umiejętności, tak naprawdę nie umiem niczego specjalnego honestly. Nie mam pojęcia, jak to się stało.

Będę tylko musiał znowu zaprzyjaźnić się z gołębiem Bartkiem siedzącym na parapecie, bo powrót do pracy zdalnej i brak kontaktu ludzkiego z biura pewnie tego będzie wymagał.

W każdym razie. Oficjalnie doszedłem do poziomu, gdzie zarabiam 10 razy więcej, niż jeszcze 2-3 lata temu. To nie był łatwy okres.

20.7.6 Nic co dobre nie trwa jednak wiecznie

Następny okres, to czas, kiedy moje życie było burzliwe zawodowo. Miałem wtedy 3 lata doświadczenia i odszedłem z dopiero co opisanej firmy po około 1,5 roku. Tu zaczęła się niezła jazda i wariacje, które skończyły się generalnie tym, że jakoś wylądowałem z własną firmą i 7-8 pracownikami na etacie. WHAT THE. Do tego jednocześnie robiłem etat dodatkowo. Moje zarobki tutaj potrafiły być szalone od 100 zł za godzinę do, rekordowo, 600 zł/h. Tak. 600 zł/h. Niezły mix. W tym okresie moje miesięczne zarobki wahały się od 15k do okolic 60-70k miesięcznie. Średnio wychodziło różnie, ale pod koniec przynajmniej 5 dych do łapy zgarniałem, natomiast warto zaznaczyć, że pracowałem kapkę za dużo. Poniosła mnie fantazja, presja, kompleks Boga i zachłyśnięcie się pieniędzmi. W pewnym momencie coś pękło. W 4. kwartale roku 2021. Zszedłem ze sceny i zniknąłem na dość długo. Tak rozpoczęła się moja roczna przerwa w życiorysie, mój upadek, z którego dopiero wracam, rok, gdy zaliczyłem rekordowe personalne wzloty jak i upadki.

Co się wtedy ze mną działo? To długa historia.

20.7.7 Powrót do rzeczywistości

Jak miałem równo 23 lata i 5 lat doświadczenia za pasem zaszła ciekawa sytuacja. Licząc nadgodziny to z 8 tych lat, nawiasem mówiąc. Kilka dni przed urodzinami podpisałem umowę. Plus minus sprowadza się ona do ekwiwalentu w postaci: 36k brutto na UoP, plus 130k EUR w opcjach, vestowanych 5 lat. W praktyce znaczy to tyle, że jeśli przepracuje w firmie przynajmniej rok to dostanę pakiet opcji obecnie wycenianych na 26k EUR. Potem przez następne 4 lata co miesiąc kolejne opcje o wartości 2.1k. Natomiast warto zaznaczyć, że te opcje/udziały były relatywnie warte dość niedużo. Znaczy wtedy wycena to 130k, natomiast relatywnie niedużo do potencjału firmy.

Firma jest w fazie dynamicznego wzrostu i zanim te 5 lat minie to oczekuje, że wycena tychże akcji wzrośnie x5-10, plus dodatkowo spodziewam się w ciągu tych lat wynegocjować więcej akcji. Póki co, po obecnej rynkowej wycenie, uwzględniając wszelakie bonusy etc

wychodzi mi ~46k na UoP brutto miesięcznie. Do tego 3k euro na rozwój na rok. A no i za podpisanie umowy też mi bonus dali. 20k do łapy wleciało. Praca oczywiście zdalna. Plus 26 dni płatnego urlopu.

Mam w życiu więcej szczęścia, niż rozumu. Wróciłem po prawie rocznej przerwie i trafiło się ślepej kurze ziarno.

Tak by się zdawać mogło. Bajka, co? Była, aż do czasu gdy pierwszego tygodnia pracy ktoś przeczytał mój artykuł o tym, iż zarządzać ludźmi trzeba po ludzku, a nie bazując na excelu, który za zgodą i review zespołu od tworzenia treści, umieściliśmy na blogu. Wtedy zostałem zwolniony ze skutkiem natychmiastowym. Ordnung must sein. Zderzyłem się z hipokryzją organizacji gdzie wycierano sobie mordę stawianiem na człowieka, sprawiedliwością i ludzkim podejściem, a gdzie rzeczywistość była przeciwieństwem, gdzie menadżer w pierwszym tygodniu rzucił słowami "Olaf, to twój zespół, nie jestem do nich przywiązany i mnie to nie interesuje, jak ci się coś z nimi nie podoba, to możemy ich zwolnić i zatrudnić kogoś innego.". Tak średnio ludzkie podejście. Zupełnie to nie grało z moim podejściem, które zakłada, że człowiek to coś więcej niż trybik w maszynie.

Tego samego dnia jednak dogadałem się już z inna firmą, którą pierwotnie odrzuciłem. Nie mógłbym być bardziej zadowolony z tego, jak sprawy się potoczyły.

20.7.8 Małe podsumowanie mojej historii zarobkowej

Tutaj streszczona lista mojej historii zarobkowej, którą skopiowałem ze swojego posta na 4programmers.

Data zaczęcia: 12/2017

1. Wiek: 18, doświadczenie: 0, stanowisko: Junior Python Backend Developer, zarobki: 18 zł/h netto UoZ, czyli 3000 przy 168 h.
2. Wiek: 18, doświadczenie: 2 miesiące, stanowisko: Junior Python Backend Developer, zarobki: 22 zł/h netto UoZ, czyli 3700 przy 168 h
3. [Nowa firma] Wiek: 18, doświadczenie: 8 miesięcy, stanowisko: Junior Python Backend Developer, zarobki: 37.5 zł/h netto UoD, czyli 6300 przy 168 h
4. Wiek 18, doświadczenie: 9 miesięcy, stanowisko: Junior Python Backend Developer, zarobki, zależnie od klienta/firmy: 46-50 zł/h netto B2B + płatny urlop, czyli 7900 przy 168 h, czyli przy nadgodzinach raczej >10k netto.
5. [Nowa firma] Wiek: 19, doświadczenie: 1,5 roku, stanowisko: Python Developer + trochę coś tam zaczynam into opsowanie. Zarobki: 86 zł/h netto FV. Praca zdalna.
6. [Nowa firma/Swoja firma] Wiek: 21, doświadczenie: 3 lata, stanowisko: Tech Lead/Engineer/CEO/księgowy/Technical product analyst - własna f. Zarobki: 15-60k/msc netto FV.
7. [Nowa firma] Wiek: równo 23, doświadczenie: ~5 lat, stanowisko: Tech Lead, praca zdalna; Bonus za podpisanie umowy: 20k pln do łapy. Zarobki: 36k UoP Brutto + bonus w postaci opcji o wartości 130k EUR vestowanych przez 5 lat, czyli średnio licząc z opcjami wychodzi około 45k brutto na UoP. Do tego 26 dni urlopu i 3k euro roczny budżet rozwojowy. Bajka, co? Była, aż do czasu gdy pierwszego tygodnia pracy ktoś przeczytał mój artykuł o tym, iż zarządzać ludźmi trzeba po ludzku a nie bazując na excelu, który za zgodą i review zespołu od tworzenia treści, umieściliśmy

na blogu. Wtedy zostałem zwolniony ze skutkiem natychmiastowym. Ordnung must sein.

8. [Nowa firma] Wiek: 23 lata, doświadczenie: ~5 lat, stanowisko: Lead Python Developer, praca zdalna; Zarobki: 36.5k netto na B2B albo 210 zł/h. Cieszę się, że z poprzednią firmą wyszło jak wyszło.

21 Wady i zalety pracy jako programista

Ta książka traktuje o programowaniu, prawda? Milion jest takich książek, wszystkie zachęcają do tego, by wejść w świat IT, by programować, bo to przyszłość, bo to życie na poziomie, miliony monet w portfelu, chwała gloria i co tam jeszcze.

No nie, nie zgadzam się. Trzeba polać mały kubeł wody na sporą część osób, które aspirują i marzą o branży it. To nie jest jakaś magiczna, idealna praca, która ma same plusy, żadnych wad i tak dalej. ABSOLUTNIE. Wybij to sobie z głowy.

21.1 Wady

By walczyć z tym przekonaniem, postanowiłem na w treści swej książki napisać dlaczego... nie warto programować.

21.1.1 Zdrowie

Programowanie to długie godziny przy komputerze, często zbyt długie. To nie jest zdrowe. Oczy, kręgosłup, inne rzeczy. Nie bez powodu zaczęło ostatnio funkcjonować takie powiedzenie: sitting is the new smoking.

Siedzący tryb życia i wlepienie w komputer niszczy nasze organizmy i to naprawdę mocno. Jeśli ktoś nie ma na tyle samodyscypliny, by to siedzenie później negować za pomocą siłowni, ćwiczeń czy aktywności ruchowej, to szczerze, na nic mu te pieniądze zyskane dzięki pracy, bo nie będzie miał zdrowia, by się nimi cieszyć.

Kolejną sprawą jest stres. Wierzcie lub nie, ale kiedy w końcu dojdziecie do miejsca, kiedy jesteście w miarę normalnymi programistami i zaczniecie odpowiadać za pewne sprawy, dedlajny i tak dalej, to stres pojawi się nieuchronnie.

Bo jak się nie stresować, kiedy coś popsułeś w kodzie i nagle się okazuje, że klient traci dziesiątki tysięcy każdej godziny, bo zrobiłeś coś głupiego? Co, kiedy okazuje się, że błahy problem, który miałeś zrobić do końca dnia, okazuje się czymś zawiłym, nad czym spędzisz następny tydzień mimo swych deklaracji, że będzie to zrobione w kilka godzin?

Mimo tego, że dla mnie są to obce pytania, gdyż zwyczajnie nie umiem stresować się pracą, mimo tego, że mocno się w nią angażuję, to wiem, że dużo osób takie problemy ma.

Brzmi trochę jak przechwalanie, ale z jakiegoś powodu tak właśnie jest. Po prostu żyję w niezatrwożonym przekonaniu, że ostatecznie dam radę, wszystko się ułoży i będzie dobrze.

21.1.2 Ciągła nauka

Co z czasem, który musisz poświęcać na naukę, nieustanną naukę? Ta branża zmienia się na tyle szybko, że jeśli sam się nie rozwijasz, to wypadasz z gry. Proste. Nie żartuję.

Nie ogarniasz nowinek czy nowych rzeczy przez rok, dwa, trzy i nagle stajesz się trochę bezwartościowy na rynku lub osiągasz stagnację, gdzie twoja wiedza się nie zwiększa, podobnie umiejętności.

A w naszej branży stagnacja oznacza nie tyle, co brak postępu, a regres w zasadzie. Dotyczy to jednak głównie junior developerów. Oni muszą mocno i dużo się rozwijać. Inaczej to nie ma sensu.

Seniorzy czy dobrzy regularowie, mają opanowane pewne podstawy i umiejętności, które są niejako kluczowe i niezmienne, co pozwala im na nieco większe lenistwo.

Zresztą, im większe twoje doświadczenie, tym tak naprawdę potem zaczynasz mniej kodować, więcej czytać kodu, mentorować innych, uczestniczyć w spotkaniach, dokonywać pracy koncepcyjnej, architektonicznej.

Dla jednych fakt przymusu ciągłej nauki jest minusem, dla innych plusem. Ja się akurat z tego cieszę, bo naprawdę interesuje mnie to, co robię, kocham swój zawód, zatem to, że muszę być na bieżąco z pewnymi rzeczami, to niezła wymówka, by posiedzieć trochę więcej przy kodzie.

Jednakże są osoby, dla których programowanie to po prostu praca od 8 do 16 i tyle. I to też jest normalne, prawidłowe i spoko, że się tak wyrażę. Nie każdy musi być kompletnym pasjonatem. Potrzeba też ludzi, którzy po prostu robią swoje i koniec. Mogą być oni dobrymi programistami, nie będą tymi najlepszymi, tą śmietanką, ale wciąż.

Tacy ludzie, to znaczy, przeciętni, albo nie aż tak wybitni, też są potrzebni, to zupełnie normalne. W zasadzie większość osób działa na takiej zasadzie. Pasjonaci to rzadkość.

21.1.3 Słabe projekty

Nie zawsze pracujemy nad fajnymi appkami. Realia są takie, że czasami pojawiają się paskudne projekty, których nikt nie chce robić, ale ktoś jednak musi. I wiesz co?

Ty, jako Junior masz większe szanse na to, że pracodawca wrzuci cię w takie gówno. Dlaczego? Bo jak coś i tak jest skopane, to gorzej tam nic nie zepsujesz, to raz.

Dwa, na początku nie masz jakiegoś zbyt dużego wyboru, o ile nie jesteś dobrym programistą. A w sumie, mam nadzieję, takim się staniesz.

Takie realia niestety. Bo jak już babrzemy się w takim bagnie, że nic nie pomoże, to cóż, w zasadzie można tam wrzucić kompletnego świeżaka, bo co by nie zrobił, to gorzej nie będzie, a wyjdzie taniej.

Podałem tu trochę przykład ekstremalny, ale bardzo prawdopodobny, jeśli trafisz do pracy w Januszsofcie, a tak często bywa w pierwszych pracach.

21.1.4 Stereotypy

IT to też stygmat – przygotuj się na to, że wszyscy znajomi, rodzina i inne osoby, które znasz, będą oczekiwać od ciebie wiedzy wielkiej w dziedzinach, które są choćby najmniej powiązane z programowaniem.

Jaki wybrać telefon, jaki komputer, jaki router, jak go skonfigurować i tak dalej. To nieważne, że nie pracujesz jako admin, devops czy jakiś inny help desk. Każdy oczekuje, że będziesz się znał na wszystkim, co z it/technologią związane. Nie jest to jakaś duża wada, ale po pewnym czasie potrafi być irytujące.

Protip: nigdy się nie oferuj nigdzie, że 'reinstalniesz komuś system', zwłaszcza za darmo. Potem wszyscy będą przychodzić do ciebie ze swoimi problemami technologicznymi i ciebie za nie winić.

Jak już chcesz być dobrą duszą, to weźże, chociażby, tę przysłowiową flaszkę jako rekompensatę.

Poza tym w momencie, kiedy ujawnisz się, jako programista, że tak powiem, duża część osób, przypnie ci pewną łatkę co do tego, ile zarabiasz. U niektórych budzi to zazdrość.

Chociaż to w zasadzie zaleta, bo dzięki temu można łatwo spostrzec, komu zależy na twoim statusie materialnym, a kto jest prawdziwym przyjacielem.

21.1.5 'Rynek pracownika'

Kolejną wadą jest to, że rynek wcale nie jest taki różowy. Co, trenerzy z bootcampów ci naopowiadali, że 60k programistów brakuje, racja?

No nie, bzdury. Bo potrzeba wielu specjalistów, ale... Właśnie, specjalistów. A nie początkujących. Potrzebni są ludzie, którzy już mają wiedzę i 2, 3, 4 czy 5 lat w branży siedzą. Takich początkujących co chcą zostać programistami, bo usłyszeli, że tam dobrze płacą, to jest na pęczki, gdzie nie spluniesz, tam jakiś wannabe programista.

Nie zrozum mnie źle, nie mam nic do osób, które chcą się przekwalifikować czy zostać programistami. Fajnie, niech to robią. Śmieszą mnie jedynie osoby, które uważają, że napisanie hello worlda w pythonie i jednej CRUDowej appki zrobi z nich programistów, którzy po dwóch tygodniach będą zdolni do pracy i to wszystko jest w ogóle takie łatwe proste i przyjemne.

I to nie żart, na większość ogłoszeń, gdzie widnieje słowo klucz 'junior' albo 'brak doświadczenia' potrafi przyjść nawet 500 CV. Widziałem takie przypadki. Podejrzewam, że za niedługo liczba otrzymywanych CV na ogłoszenia juniorskie wzrośnie do czterocyfrowej liczby. Osób, które chcą się dostać, naprawdę jest multum. Problem w tym, że większość z nich nic nie umie lub ma wygórowane oczekiwania i zero pojęcia o branży.

Naprawdę teraz trochę trzeba się wyróżniać, żeby dostać się w jakieś sensowe miejsce. Pewnie, są firmy, które same nawet szkolą programistów i płacą im za naukę, gwarantują pracę, ale to zazwyczaj wiąże się z dwiema rzeczami: lojalką na X lat to raz, dwa – z pracą nad projektami, których nikt się nie chce tknąć. Poza tym zdarza się to baaaaaardzo rzadko, by firma na coś takiego poszła.

Niestety, życie to nie bajka, tutaj się kalkuluje. Inwestycja w pracownika nie jest bezinteresowna – to dokładnie to, co napisałem, czyli INWESTYCJA. Robi się ją z myślą, że się zwróci z nawiązką.

Mówię to bardziej w kierunku osób, które myślą, że o pracę w IT jest niesamowicie łatwo, na rynku jest ogromny głód i w ogóle. Nie. Do was apeluję, do tych, którzy sięgnęli po tę książkę z podejściem, że sobie przeczytają jedną czy dwie książeczki, przeprogramują dwa tygodnie w domku i cyk, 15k na miesiąc. NIE MA TAK.

Pozycję siły to masz wtedy, kiedy na twoim karku jest już kilka/kilkanaście lat doświadczenia, spora wiedza i renoma. Wtedy możesz przebierać w ofertach.

Albo i nie, bo do tych porządnych firm chce dostać się każdy, zatem masz większą konkurencję i może być różnie. Im lepsza firma, tym większe wymagania może stawiać, bo więcej kandydatów ma. Koniec dygresji.

Nie twierdzę tutaj, że jest to też niemożliwe, by dostać pracę bez doświadczenia – popatrz na mój przypadek, ale nie jest aż tak łatwo, jak to niektórzy opisują, że jest. Trzeba to sobie koniecznie wbić do głowy.

Z drugiej strony, nie ma co przesadzać – znalezienie pracy nie wymaga herkulesowego wysiłku. Pod warunkiem, że masz coś w głowie, stosujesz się do zasady NBD oraz opanujesz solidną wiedzę i podstawy.

Same chęci nie wystarczą. Wiem, że to zabrzmi surrealistycznie, ale jest cała rzesza ludzi, która deklaruje się, że będą pracować za darmo nawet przez pół roku, byle tylko

złapać pracę.

Jak zwykle, wszystko zależy od Ciebie, twoich projektów, kontrybucji open source, oraz tego, jaką wiedzę posiadasz. Im większe twe umiejętności, tym większa szansa, że znajdziesz jakąś pracę, ale trzeba też pamiętać, że liczy się również prezentacja oraz sposób szukania.

21.1.6 Wymarzona praca

'Rób to, co lubisz, a nie przepracujesz nawet jednego dnia.' Czy jakoś tak. Nie wiem, kto to powiedział, ale kretyn z niego. Nawet jeśli jara cię programowanie tak jak mnie, to wciąż w pracy czeka cię masa rzeczy, które nie są stricte programowaniem, ale są konieczne przy procesie tworzenia oprogramowania.

Meetingi, spotkania, dyskusje, papierologia, przekonywanie klienta do pewnych rozwiązań czy też jego jakieś fanaberie. Wiele mógłbym tu wymieniać.

Oczywiście, jeśli pracujesz w porządnej firmie, gdzie praca jest sensownie zorganizowana, to część tych elementów zostaje wycięta. Nie da się jednak wyciąć wszystkiego.

Jeśli myślisz, że przyjdziesz do jakiejś pracy i będziesz po prostu siedział przy biurku przez 8h, tworząc zajefajny kod, to jesteś w błędzie.

Jednakże załóżmy, że nie jesteś. Siedzisz sobie cały dzień przed komputerkiem i klepiesz kod. Czy to sytuacja idealna? Nie. Dlaczego?

Po pierwsze, w głównej mierze, praca programisty opiera się na myśleniu. Zanim coś zakodujesz, trzeba to przemyśleć i to solidnie. W zasadzie większość czasu spędzonego w pracy to myślenie i czytanie kodu.

Samo klepanie w klawiaturę to mała część. I bardzo dobrze. Gdyby było inaczej, to nikt by nie był w stanie przepracować 8 godzin. Nawet w tym modelu, gdzie większość czasu to research i myślenie, ciężko być produktywnym przez te 8h.

Tak nie będzie. Po prostu. To nierealne. W programowaniu są ciężkie chwile, dni, kiedy wydaje ci się, że nic nie umiesz i kto wie, po co tu pracujesz – tak zwany impostor syndrome. Wbrew pozorom bardzo popularne zjawisko.

Są też słabe projekty, zadania, do których nie mamy ochoty siadać, ale i tak musimy. Programowanie naprawdę bywa męczące i to bardzo.

I mówię Ci to ja, osoba, która kiedyś pracowała na budowie. W zasadzie czasami tęsknię za tym, jakie to było proste. Czasami z chęcią bym sobie obsadził takie narożniki gdzieś na glifach jakichś. Idealnie, do ładnego kąta prostego. Goldbandzikiem ładnie przejechać. Oj tak. Potem gładź na dwa razy i wyszlifować na lusterko. Nie ma lepszego uczucia.

Pewnie, taka praca męczy fizycznie, czasami bardzo, zwłaszcza jak trzeba dymać na rusztowaniu przy ogromnej wilgotności, bo farba schnie, i przy 35-stopniowym upale w lecie, albo gdy musisz przerzucić milion pustaków lub paletę 25 kilowych worków tynku wnieść na 4 piętro.

Po takim dniu w pracy wracasz do domu, jesz i kładziesz się spać zaraz w zasadzie. Jasna sprawa. Przynajmniej na początku, bo potem się przyzwyczajasz do tego i nie męczy cię to aż tak. Mentalnie za to, jesteś zupełnie wolny.

Ja, podczas takiej pracy, zamykałem się po prostu w swoim świecie i myślami byłem gdzie indziej. Czas mijał bardzo szybko wbrew pozorom. Czasem nawet przyjemnie,

zwłaszcza jak ekipa do pracy była fajna. Do tego masz namacalne efekty swojej pracy. Tworzysz coś, co widzisz, czego można dotknąć, zobaczyć postęp.

Naprawdę. Raczej miło będę wspominał czas spędzony ze szpachelką w ręku. Dużo mnie to nauczyło. Jakbym miał wymienić najcięższy element pracy tego typu, to jest to aspekt finansowy i ograniczenia, które nakłada na ciebie mizerna wypłata. Chociaż i to ostatnio się zmienia, bo teraz z racji mocnego deficytu na rynku, pracując jako fizyczny, jesteś w stanie zarobić grubszą kasę. Serio. Programista15k się czasem chowa.

W każdym razie. Chodzi mi o to, że po takim dniu w robocie, potrafiłem być mniej wyczerpany, niż po dniu pracy jako programista.

Podsumowując, praca programisty, zwłaszcza wtedy, kiedy jesteś zafascynowany tą dziedziną, jest cudowna. Nie jest jednak bez wad, posiada ich nieco. Jedne są prozaiczne, inne poważne, ale są. Mnie, w większości, zupełnie nie przeszkadzają, ale potrafią być uporczywe. Ważne, byś był ich świadom, zanim podejmiesz decyzję, na hurra, że się przekwalifikujesz i tak dalej.

21.2 Zalety

Narzekania już starczy, teraz pora na zalety. Oh boy. Od czego by tu zacząć.

21.2.1 Pitos

Co tu dużo gadać. Hajs się zgadza. Natomiast warto zaznaczyć, że występują skrajne różnice. Otóż znam osoby, które mają podobny staż co ja, a zarobki od 20% do 80% niższe od moich. W IT jest szansa na to, by dobrze zarabiać, dość duża, ale trzeba też włożyć wysiłku i znać się na tym, co się robi. Do tego dochodzą umiejętności miękkie.

21.2.2 Możliwości rozwoju

IT to tak dynamiczna branża, że nigdy nie można narzekać na nudę. Rzeczy ewoluują tu tak szybko, że ścieżki rozwoju i jego możliwości są w zasadzie niezliczone. To mega spoko. Z drugiej strony, jak już wspomniałem o tym, dla niektórych może być to wada. Albo idziesz do przodu, albo umierasz, bo jak stoisz w miejscu to się cofasz. Aż zapachniało Kołczem Majkiem. Every day we stray further away from God.

21.2.3 Ciekawa praca

By napisać kawał kodu i stworzyć sensowny produkt, trzeba rozumieć jak on działa, jaki jest business case, jaki jest kontekst i tak dalej. To znaczy, że tak naprawdę niejako trzeba się danego biznesowego zagadnienia nauczyć. To sprawia, że można na chwilę stać się tak jakby człowiekiem z innej branży i popróbować nowych rzeczy. Dla mnie to niesamowite, że pracując nad nowymi produktami często uczę się bardzo dużo o innych branżach, profesjach. To mega fajne i ciekawe moim zdaniem.

Do tego same technikalia i szczegóły implementacyjne naszej pracy bywają naprawdę niebanalne, zwłaszcza, jeśli ktoś ma tę inżynierską smykałkę.

21.2.4 Zdalna również często

W obecnych czasach praca w IT jest bardzo często zdalna. Niesie to ze sobą pewne wady, o których już pisałem, gdzie cierpi zwłaszcza socjalizacja i życie społeczne, przynajmniej u niektórych, ale jednocześnie daje nam wiele możliwości. Praca zdalna to przede wszystkim ogromna elastyczność w grafiku, życiu. Wygoda - nie tracimy czasu na dojazdy. Natomiast najważniejsze dla mnie w pracy zdalnej jest chyba to, że możemy pracować dla firm z całego świata i wykorzystywać siłę nabywczą mocniejszej waluty z zamożniejszego kraju, mieszkając w miejscu, gdzie lokalna waluta ma słabą siłę nabywczą.

Mówiąc po chłopsku, jak mieszkasz w takiej Polsce, gdzie złotówka to żeton zrobiony z kartonu a rząd regularnie urządza ruchanie tegoż żetonu po same kukle, bez wazeliny, gdzie dolar jest po pinć złoty, to zarabiając w euro czy dolarach będziesz miał tu prawdopodobnie jeszcze większe eldorado niż w porównaniu do twoich kolegów, którzy zarabiają w złotówkach.

Dodatkowo firma, która jest z zagranicy i pracuje w trybie zdalnym też umie policzyć. Lepiej zapłacić inżynierowi z USA, gdzie koszty życia są większe i nie zgodzi się on za pracę za mniej niż 20k USD na miesiąc, załóżmy, czy dać 10k USD polakowi, który pracę wykona prawdopodobnie podobnie albo równie dobrze, będzie mega zadowolony z pensji a mimo tego połowę tańszy bo za 10k USD w Polsce można żyć prawdopodobnie na większym poziomie niż w USA za 20k USD? Oczywiście liczby są zmyślone, ale obrazują sytuacje.

Polska jest chyba jednym z niewielu krajów, gdzie różnice w zarobkach między średnią pensją a pensją w IT są tak ogromne. W Norwegii tego nie ma. W Szwajcarii też niekoniecznie. W Niemczech podobnie. Znaczy jest, oczywiście, ale nie są to różnice rzędu 10x średnia pensja.

21.2.5 Rynek pracownika

Mimo tego, że dementowałem plotki o takowym w części o wadach pracy jako programista, to dotyczyło to nie każdego. Otóż jeśli naprawdę znasz się na swojej robocie i cokolwiek sensownie ogarniasz, to ssanie na rynku jest takie, że głowa mała. Naprawdę. I wcale nie będzie mniejsze.

Recesja idzie. Okej, napompowane dziadostwa, które nie mają prawa żyć na wolnym rynku, umrą. Natomiast pozostałe firmy będą rosły. Będą rosły też chęci na optymalizacje i redukcje kosztów we wszelakich przedsiębiorstwach. Co to oznacza? Często automatyzację procesów, redukcję zatrudnienia, optymalizację istniejących procesów. Znajome rzeczy, często robi się to za pomocą technologii właśnie.

Także moim zdaniem w sektorze IT mamy obecnie BARDZO mocny rynek pracownika, pracownika wykwalifikowanego. Sytuacja ta będzie się tylko pogłębiać moim zdaniem.

21.3 Podsumowanie

Bycie programistą ma swoje wady i zalety. Szczerze, więcej zalet moim zdaniem. Natomiast są też pewne wady, które urealniają ten zawód jak i oczekiwania niektórych ludzi. To nie jest jakiś święty Graal pracy. Nie. To robota jak każda inna, akurat obecnie modna i

lukratywna. Dla niektórych będzie jednak katorgą, która nie jest warta zachodu, gdyż mają lepsze predyspozycje w innych kierunkach i tam też powinni raczej zmierzać.

Wymieniłem tutaj po trochu zarówno wad jak i zalet, może da Ci to kapkę szerszą perspektywę.

22 Praca zdalna

Trochę o pracy zdalnej pogadamy i jak zrobić to po ludzku.

Wiele osób nie zdaje sobie sprawy z tego, że praca zdalna wymaga tego, by pewne kwestie rozwiązać dobrze. Inaczej długoterminowo może być nieprzyjemna. Praca zdalna to nie tylko pracowanie z domu. Dobrze zaimplementowana wymaga zmian nie tylko w lokacji wykonywanej pracy.

22.1 Ergonomia

Pracując zdalnie czasem zapominamy o ergonomii czy higienie pracy. Będąc młodym łatwo jest wpaść w mit niezniszczalności. W końcu nie od razu czuć konsekwencje pewnych akcji. Dopiero po latach pewne rzeczy wychodzą. Mówię to ja, 23 letni człowiek. Przy komputerze spędzam godziny już od wielu lat. W branży od 5. Mimo tego już zdążyłem się przekonać co to znaczy wypalenie zawodowe czy problemy zdrowotne spowodowane pracą. Nie chcę, byś podzielił mój los. Zatroszcz się zatem o ergonomię miejsca z którego pracujesz. Pozwoli ci to nie tylko być zdrowszym, ale i efektywniejszym w swojej pracy w dłuższej perspektywie.

Co to konkretnie znaczy omówię na swoim przykładzie.

22.1.1 Mój ergonomiczny setup

Przeanalizujmy to, jak wygląda moja sytuacja i jakiego sprzętu używam.

1. Monitor zewnętrzny + laptop. Prosta sprawa. Dodatkowy monitor zewnętrzny pozwala mi ułatwić pracę poprzez większą powierzchnię do wyświetlania rzeczy. Fajnie, żeby były to przynajmniej 27", wysokiej rozdzielczości. Dla jednych FullHD wystarcza, dla innych 2K to mało. Ja osobiście skłaniam się ku 2k/4k. 4k może kapkę za dużo, stąd kompromisem wydaje mi się 2k. Porządny monitor nie będzie męczył oczu. U mnie sprawdza się 27" poleasingówka od della, którą nabyłem za 800 zł. Rozdzielczość 2k. Ten dodatkowy monitor naprawdę poprawił i usprawnił moją pracę.
2. Mikrofon Pracując zdalnie fajnie jest mieć dobry mikrofon. Za 200-300 zł można mieć naprawdę porządny sprzęt a mamy gwarancję, że będzie nas dobrze słychać, redukcja szumów etc. Polecam Thronmaxxy. Cena do jakości jest niesamowita. Zawsze miło, kiedy nie ma się problemów ze zrozumieniem współpracownika. Diabeł tkwi w szczegółach.
3. Kamerka Zazwyczaj starczy ta wbudowana czy też dla bardziej wymagających zewnętrzna kamerka lub dla najbardziej wymagających geeków wewnętrzny aparat cyfrowy podłączony do laptopa. Fajnie jest mieć od czasu do czasu kontakt wzrokowy ze współpracownikami. Nie ma co tego jednak nadużywać, nie zawsze kamerka jest potrzebna.
4. Biurko stojące. Jestem strasznie hiperaktywny. Biurko stojące pomaga mi tym zarządzać. Dodatkowo fakt, że od czasu do czasu wstaję, dobrze wpływa na moje zdrowie, plecy i ogólną kondycję fizyczną. Siedzenie jest średnio zdrowe. Przykurcze, problemy z kręgosłupem. Można się załatwić.

5. Porządny fotel Jak już siedzimy dużo, to najlepiej na czymś spoko. Jedni lubią piłki gimnastyczne, inni ergonomiczne fotele, jeszcze inni taborety. Upodobań jest wiele. Ja od siebie polecam jednak porządny fotel typu Ergohuman czy coś w tym stylu, ale tańsze. Wydatek jednorazowy, a popłaca.

Powyższe rzeczy to moim zdaniem podstawy w ergonomicznej pracy zdalnej, które przydatne są, jeśli chcemy zachować zdrowie na lata. Oczywiście, da się pracować i bez tego. Nie musisz od razu do samej nauki zaopatrywać się w krzesło za tysiąc złotych czy mikrofon za trzysta. Nie. Natomiast jeśli jesteś już profesjonalistą, to nie bój się inwestować w swój sprzęt. To narzędzia twojej pracy takie same jak wszystkie inne. Ułatwisz życie sobie i innym.

Wiem, że wydać się to może śmieszne, ale podświadomie nieco bardziej poważnie biorę osoby, które mają fajny setup. Nie jest to wyznacznikiem niczego, ale tak jakoś mam. To sygnalizuje mi pewne rzeczy podświadomie.

Na koniec dnia oczywiście liczy się tylko to, czy dowozisz i nie jesteś dzbanem, ale wciąż.

Oto kilka wskazówek dla osób pracujących zdalnie:

1. Stwórz odpowiednie miejsce do pracy: Zadbaj o to, aby Twoje miejsce pracy było wygodne i ergonomiczne. Unikaj pracy z łóżka lub na kanapie, ponieważ może to prowadzić do problemów z kręgosłupem i innych schorzeń.
2. Ustal granice czasowe: Ważne jest, aby ustalić granice czasowe dla swojej pracy, tak aby móc odpowiednio rozdzielić pracę i życie prywatne.
3. Zadbaj o dobrą komunikację: Praca zdalna wymaga dobrej komunikacji z innymi członkami zespołu i przełożonymi. Upewnij się, że masz dostęp do odpowiednich narzędzi komunikacyjnych, takich jak komunikatory internetowe lub wideokonferencje.
4. Dbaj o swoje zdrowie: Praca zdalna może prowadzić do siedzącego trybu życia, co nie jest zdrowe. Upewnij się, że zachowujesz prawidłową postawę przy komputerze i wstań co jakiś czas, aby się rozprostować i zrobić kilka ćwiczeń.
5. Utrzymuj dyscyplinę: Praca zdalna może być trudna, jeśli nie masz odpowiedniej dyscypliny. Ustal swój plan dnia i trzymaj się go tak, jakbyś pracował w biurze.
6. Zachowaj rutynę: Praca zdalna może być rozpraszająca, więc ważne jest, aby zachować rutynę, taką jak ustalenie stałych godzin pracy i utrzymywanie regularnych przerw.

23 Luźne przemyślenia

Ten rozdział to zbiór moich różnych przemyśleń, apeli, cokolwiek.

23.1 Backend to nie produkt

Chciałem kilka słów napisać nie o programowaniu per se, ale o programowaniu jako pracy. Otóż jestem dość młodym developerem, mam plus minus, w tej chwili, około dwóch lat doświadczenia komercyjnego, a jeśli o sensowną pracę w zespole chodzi, to jeszcze mniej.

To malutko. Lat na karku też za dużo nie mam, bo przecież nawet 20. mi jeszcze nie wybiła. Dlatego też ciągle uczę się wielu rzeczy i jednocześnie widzę, że inni ludzie, z większym stażem, starsi ludzie, mają podobnie — również się uczą, stąd też wrażenie, że warto poruszyć temat, o którym dziś piszę, bo wydaje mi się, że niezależnie od wieku i stażu, dużo osób tego nie rozumie.

23.1.1 Małe oświecenie

Otóż niedawno zaświeciła mi się w głowie lampka — zadaniem zespołu jest stworzenie rozwiązania, produktu, czegoś, co zadowoli klienta, spełni jakiś cel, rozwiąże dany problem. No właśnie.

Co w związku z tym? Piszę to z perspektywy backend developera, otóż: backend nie jest produktem sam w sobie.

Frontend też nie. O co mi chodzi dokładnie? Do pewnego czasu miałem podejście w stylu — nic mnie nie obchodzi, że funkcjonalność X nie została dowieziona na czas/nie działa/są problemy z implementacją — ja swoje zrobiłem, backend działa i jest piękny, to frontend/devops/qa/ktokolwiek nie dopiął i produkt nie działa.

To kompletnie błędne podejście, zrozumiałem to. Po części pomogło zrozumieć mi to, jak wyglądał proces dowożenia aplikacji w naszym zespole w jednej z poprzednich firm, oraz jakie problemy napotykaliśmy po drodze, które wynikały głównie właśnie przez takie myślenie, a z drugiej strony, zostałem lekko skierowany, by spojrzeć na ten problem w moim myśleniu, przez mojego de facto mentora.

Tu też fajna sprawa — zamiast od razu o czymś mówić, pozwolono mi spróbować czegoś na własną rękę, przemyśleć sprawę, a potem dojść do poprawnego rozwiązania samodzielnie — szacunek za to.

23.1.2 Jak powinno być

Kończąc dygresję — gdy piszemy backend/frontend, cokolwiek, nie możemy myśleć tylko o tym, jaki on będzie fajny, piękny i niesamowity z perspektywy czysto backendowej/frontendowej.

Musimy pomyśleć, jak innym będzie się współpracowało z kodem, serwisem, który tworzymy.

Czy rozwiązanie, które akurat tworzymy, będzie wygodne dla frontu? Czy wszystko jasno opisałem tak, by osoba pisząca integrację z API nie miała już żadnych pytań, bo wszystko zostało klarownie opisane w dokumentacji, dodatkowych notatkach? Może mogę im pomóc w inny sposób, rozmawiając z nimi, przed w trakcie i po implementacji?

Dobre pytania, które, uważam, warto przemyśleć. Wydaje mi się, że trzeba uważać na to, by nie wpaść w taki swego rodzaju egoizm, gdzie wydaje nam się, że nasza praca jest najważniejsza, my ją wykonaliśmy dobrze, to ktoś inny jest kompletnie winien i koniec, my mamy czyste ręce. To tak nie działa.

Oczywiście, są odstępstwa od reguły — ktoś celowo sabotuje prace, cokolwiek, ale to tak rzadkie sytuacje, że nie warto o nich wspominać. Programowanie to zazwyczaj praca zespołowa, a kluczowym elementem pracy zespołowej jest... no właśnie praca zespołowa.

Zespół jest bardzo ważny, tak samo, jak ważne jest to, by poszczególne 'elementy' zespołu, dobrze współgrały. Dzięki temu cały mechanizm będzie świetnie działał. A o to chodzi.

Nie zrozumcie mnie źle — nie mówię tu o tym, że cały czas trzeba robić jakiś 'team building', 'integracje', 'meetingi', 'spotkania' czy inne cholerstwa wciskane często na siłę. Nie. Nie jestem adwokatem scruma czy innych podobnych albo mniej podobnych, metodologii różnorakich. Nie.

23.1.3 Apel

Zasadniczym celem tego tekstu jest jedynie zaszczepienie takiej myśli, swoista prośba bym wręcz powiedział.

Brzmi ona następująco: podczas tworzenia swojej części kodu, serwisu, czegokolwiek, pomyśl o osobie, która będzie twój kod czytała, która będzie korzystała z twojego API, może się skonsultuj z nią, opisz wszystko, udokumentuj.

Postaw się na miejscu tego fronta, backenda, opsa.

Być może jesteś w stanie zrobić coś inaczej, co wciąż będzie technologicznie dobrym rozwiązaniem, a pozwoli drugiej stronie zrobić coś szybciej, przy czym nie spowolni to twojej pracy? Kto wie.

Takie tylko pytanie. Warto je sobie chyba zadać. Oczywiście nie ma co przesadzać w drugą stronę — tworzenie bubla programistycznego, bo front poprosił czy cokolwiek — nie, to zły kierunek.

Jak zwykle — trzeba znaleźć jakiś środek, bo przesadzanie w którąkolwiek ze stron jest złe.

Konkludując już: nie bądźmy takimi egoistami. Backend nie jest produktem, frontend też nie. Możesz mieć state-of-the-art backend/front/stronę devopsową, ale jak ten inny komponent nie będzie działał, to produkt będzie ssał.

Takie podejście, gdzie z własnej woli zrobisz więcej dla innych, zwyczajnie sprawi, że twój projekt będzie lepiej działał. Im więcej pomożesz innym, tym więcej będą oni mogli pomóc tobie. Po prostu.

Absolutnie nie znaczy to, że masz wykonywać pracę z kogoś, myślę, że rozumiesz.

To bardzo ważne, zwłaszcza teraz. Wydaje mi się, że obecnie zaczynamy przykładać coraz to większą wagę do tego, jak się z kimś współpracuje, bo inaczej zespół będzie leżał.

Także to bardzo ważne, co już nie raz podkreślałem i jeszcze niejednokrotnie pewnie powtórzę, zwłaszcza tobie, juniorze. Bo często to twój charakter i nastawienie decydują o tym, czy warto w ciebie zainwestować i przyjąć cię do pracy.

Zatem naprawdę weź sobie do serca tę radę.

I co? Chyba tyle.

A jak już przy apelach jesteśmy, to...

23.2 Chciej, pisz dobry kod

Trochę absurdalny nagłówek, prawda?

Owszem, ale nie do końca. Otóż jest jedna bardzo ważna rzecz, o której chciałbym Ci powiedzieć, najlepiej kilka razy, byś ją zapamiętał czy zapamiętała.

Otóż sprawa ma się tak, że...

Niektórzy, po pewnym czasie, dają sobie spokój. Przestają mieć ochotę, przestają się starać. Po prostu mają dość. Nie dążą już do tego, by być lepszym, by kod, który piszą, był lepszy z dnia na dzień. By każda zmiana powodowała, że repo jest w lepszym stanie niż poprzednio.

Wypaleni pracownicy.

Nigdy nie bądź taki. Proszę, to mój osobisty apel.

Staraj się być osobą, która codziennie się rozwija, której się CHCE.

I tylko tyle.

23.3 Obrzydliwe kuriozum w świecie IT

Ostatnio obserwuję obrzydliwy proceder. Proceder upolitycznia świata IT i wytwarzania oprogramowania jako całości. Co mam na myśli?

Regularny proces niszczenia naszej społeczności polityczną poprawnością.

Od dłuższego czasu przyglądam się temu procesowi, ostatnio jednak gdy zobaczyłem, iż do oficjalnego repozytorium Pythona zmergowane zostały branche pull requestów, których to jedynym zadaniem było...

Zmienienie terminologii master/slave w kontekście procesów i tak dalej. Jakby tego było mało, to dość niedawno w grupie tworzącej jądro Linuxa pojawił się obowiązkowy CoC – Code of Conduct, który brzmi... Cóż, dość specyficznie.

Sam Linus, który znany jest ze swojego 'charakterystycznego' zachowania, oraz tego, że każdemu potrafi powiedzieć co myśli, a często myśli dość ostro, ogłosił, że na razie robi sobie przerwę, by 'popracować nad swoim charakterem w świetle pewnych zmian'. No ludzie kochani.

Te dwa wydarzenia sprawiły, że postanowiłem napisać swoje dwa słowa o tym temacie. Otóż wielce mi się to nie podoba. Uważam za kompletnie obrzydliwe, wstrętne i nieakceptowalne, by robić z procesu wytwarzania IT oraz naszego społeczeństwa, jakąś politycznie poprawną abominację, która w niczym nie przypomina piękna swej oryginalnej formy. Z czym konkretnie mam problem?

Oto lista kilku rzeczy.

Zacznijmy może od omawianego przykładu zmiany terminologii Pythona. To nie jest odosobniony przypadek. Podobna sytuacja miała już miejsce kilka lat temu, tyle że w kontekście kodu framework'u Django. Jak widać, zaczyna być to trendem.

Mocno niepokojącym i głupim tak szczerze. Otóż jak pewnie wszyscy wiemy, mamy w świecie IT pewien swoisty słowniczek pojęć, które mają bliżej określone, znane wszystkim, znaczenie. Widząc niektóre słowa, niezależnie od języka, w jakim występują, możemy w dość znacznym stopniu domyślić się, co robi dana funkcja, dana zmienna czy ogółem, dany kod.

Te terminy istniały przez wiele lat, cała nomenklatura na nich bazuje i ich używa, często dość trafnie opisują to, jaką abstrakcję przedstawiają. I teraz pojawia się problem.

Bo przychodzi ktoś, kto w imię politycznej poprawności zaczyna zmieniać te terminy, bo być może kogoś one urażą, mimo tego, że nie mają prawa, gdyż używane są w innych kontekstach, innych znaczeniach, których używa się też w słownikach i tak szczerze to nikt się tym nie przejmuje, poza kilkoma głośnymi krzykaczami, którzy dali się pewnym kwestiom zwariować. Niestety z tego, co widzę, w świecie IT raczej więcej jest osób nieco pasywnych, niezbyt dominujących. Typ introwertyka, intelektualisty, który chce mieć spokój.

O ile ktoś nie wejdzie aż tak bardzo w jego strefę komfortu i jej nie zaburzy, to taka osoba nie będzie się zbytnio czemuś sprzeciwiać raczej. W tym problem. Bo pewne kręgi to widzą i wykorzystują ten fakt na swoją korzyść, by pchać wszędzie swą agendę.

To smutne. Jak dla mnie jest to kompletna głupota, która wprowadza tylko niepotrzebne zamieszanie. Co, jeśli we wszystkich językach zaczną się takie rozterki?

Co, jeśli nagle zacznie funkcjonować kilka różnych nazw zastępczych na 'niewygodne' terminy, bo jedne będę lepsze niż inne? Mniej/bardziej poprawne politycznie? Przecież to jest receptura na katastrofę. To tak jakbyście w kodzie opisywali jeden obiekt kilkoma różnymi nazwami. Nie jest to dobra praktyka, prowadzi do złego kodu i problemów z utrzymaniem, debugowaniem i wszystkim w zasadzie. Ja właśnie tak widzę skutki tych zmian.

Na pewno nie pomoże tutaj fakt, że Guido przecież zaczyna się wycofywać ze świata Pythona, oddał stery, a ostatnio ma w dodatku niezłe jazdy – chodzi mi tu chociażby o jego niedawną deklarację, kiedy to stwierdził, że, nie będzie uczył, udzielając konsultacji, białych mężczyzn, bo oni tacy okropni. Zamiast tego zamierza pomagać tylko uciśnionym mniejszościom.

No ludzie kochani, proszę was...

Następna do omówienia jest utworzenie regulaminu dla społeczności tworzącej jądro Linuxa. Nowo powstały regulamin brzmi jak paplanina wyjęta prosto z mokrego snu wojującej aktywistki LGBT.

Żebym był jasny – jestem za wolnością. Zupełnie mnie nie obchodzi, kim jesteś, co robisz, kogo kochasz i kto ci się podoba. Żyj i daj żyć. Proste.

Z założenia te nowo powstałe zasady brzmią okej, problem jednak w tym, jak zostaną one wdrożone, bo historia pokazała już, jak to się zazwyczaj dzieje. Coś, co ma być narzędziem wolności dla wszystkich, staje się aparatem ucisku.

Chodzi zwyczajnie o to, że SJW (Social Justice Warriors) zyskują tutaj potężne narzędzie, które umożliwi im kompletne pozbywanie się osób, które im nie przypadną do gustu. A im do gustu nie przypada raczej większość niż mniejszość. W zasadzie to panuje tam zasada 'kto nie z nami, ten przeciw nam'. Widzicie tę rozbieżność? Ten dysonans?

W teorii mają oni walczyć o sprawiedliwość społeczną, a praktyce są oni często tyranami, którzy niszczą ludzi. Często dobrych ludzi, kompetentnych ludzi. Jeśli nawet na Linusa udało się wpłynąć, który to jest raczej uparty i kontrowersyjny w swoich poglądach, to co będzie z innymi osobami? Co, gdy jakiś naprawdę porządny deweloper 'urazi' kogoś albo podpadnie kaście SJW?

Czy naprawdę potrzebujemy gdziekolwiek komisji ludzi, którzy będą mówić nam jak się zachowywać, co by przypadkiem kogoś nie urazić? Poważnie?

Idźcie mi z tym cholerstwem stąd, idźcie z naszego świata IT. Wracajcie do swoich wysypisk na uniwersytetach i innych miejsc, gdzie siejecie tę zarazę. Tam sobie bytujcie

ze swoją polityczną poprawnością, nie u nas, w świecie IT. Paszoł won!

Nie psujcie czegoś pięknego, bo świat IT taki jest. Piękny, inkluzywny i otwarty, sam w sobie. Popatrzcie, jak on wyglądał od początku stworzenia – losowi ludzie z Internetu, bez względu na cokolwiek, wiek, pochodzenie, kolor skóry, religię, pracowali razem nad różnymi projektami, tworzyli oprogramowanie, które rozwiązywało kolejne problemy tego świata, te mniejsze i te większe.

Po co robić w tym bałagan, mieszać się i narzucać jakieś ramy? Idźmy dalej.

Pomówmy o tak zwanej 'dyskryminacji pozytywnej', czy jak to tam teraz te dzbany nazywają. O co chodzi? A no o to, że jak jesteś białym, heteroseksualnym mężczyzną, jeszcze nie daj Boże, chrześcijaninem, to trzeba ci jakoś życie utrudnić, bo jesteś uprzywilejowany, 'check your privldż!'. Rozumiecie tę hipokryzję?

Kolejny punkt – parytety w IT. Coraz to więcej firm zaczyna rekrutować na stanowiska tylko i wyłącznie kobiety/czarnych/gejów/łatever, by spełnić normy 'diversity', 'różnorodności'.

Wydawać się wam może, że to przecież Polski nie dotyczy, że to jakieś amerykańskie problemy. Nic bardziej mylnego. Wyobraźcie sobie, że na naszym podwórku również mają miejsce takie rzeczy.

Inne, jeszcze bardziej przerażające, też – jak np. sponsor wycofujący się z finansowania danego wydarzenia, bo normy 'diversity' nie zostały zachowane, czyli na darmowe wydarzenie, na które każdy przychodzi z własnej woli, zapisało się zbyt mało kobiet. Czemu ich nie zmusiliście do przyjścia, co to ma być?! Psychoza. I to jest autentyczny przykład z Polski.

Ile opowieści słyszałem o tym, że na dane stanowisko przyjęto kobietę, mimo że była mniej kompetentna od innych kandydatów? Nie mówię tu, że kobiety są mniej kompetentne. Absolutnie. Po prostu porównując dwóch kandydatów, jeden ma lepsze umiejętności, będzie lepszym pracownikiem, drugi nie. Logiczne jest zatrudnianie lepszej osoby.

W momencie, kiedy robimy z tej logiki prostytutkę i zatrudniamy kogoś słabszego tylko dlatego, że np. jest kobietą, albo jest czarny/żółty/zielony, to coś jest nie tak. Św. Tomasz płacze gdzieś po cichu.

Takie rzeczy właśnie, takie ruchy w open source, w firmach, w społeczności, doprowadzą ostatecznie do pogorszenia jakości kodu, odejścia filarów i oryginalnych propagatorów ruchów open source. Nie widzę przyszłości wesoło, jeśli się nie sprzeciwimy tej zarazie. Patrzmy tylko na jedno. Na wiedzę, na to, co ktoś mówi i jak prawdziwe to jest. Na nic więcej. Tylko to niechaj się liczy i koniec.

I jestem świadom, że puszczając te słowa w eter, publicznie to mówiąc, mogę sobie zaszkodzić, stając się celem dla niektórych.

Wiecie co? Mam to w głębokim poważaniu. Nie zamierzam siedzieć cicho i przytakiwać tylko. Od zawsze miałem niewyparzoną gębę i mówiłem to, co uważałem. Nie zamierzam tego zmieniać.

Jak właśnie czytasz ten tekst i jesteś oburzony czy oburzona, to wiesz co... Trudno. Na tym polega wolność słowa, że każdy może mówić co uważa, nawet jeśli ci się nie podoba to, co mówi. Deal with it.

23.4 Szanujmy czytelników, szanujmy Internet

Zanim przejdziesz dalej, to czysto opiniowy wpis, zero w nim rzeczy technicznych, a jedynie moje narzekanie na pewien temat i moja perspektywa. Jeśli zaglądasz tu tylko dla tematów technicznych, to ten wpis możesz odpuścić, bo to subiektywne podejście w pewnej sprawie.

O co mi chodzi? Popatrzcie na to, jak duża część witryn, zwłaszcza tych polskich, dziś wygląda. Bez AdBlocka są niemożliwe do przeglądania, bo zwyczajnie większość ich powierzchni stanowią reklamy. Jedne w formie video, inne - zwykły obrazek. Automatycznie odtwarzające się reklamy z dźwiękiem. To dopiero zmora. Tak samo jak to, ile transferu potrafi to całe dziadostwo zeżreć.

Milion ciasteczek i szpiegowanie na każdym kroku tego, co robisz, byle tylko zdobyć o tobie dodatkowe informacje, które można sprzedać.

Nachalne zgody na newsletter, ciasteczka, powiadomienia na desktopie, rodo, srodo, popupy z serii: wyłącz adblocka gnoju bo ni momy piniendzy.

Dodawanie wszędzie pluginów fejsbuka, instagrama, snapa czy czego tam jeszcze.

Nieskompresowane grafiki ważące po kilka mega. Tony niepotrzebnego JS'u, który sprawia, że strona ładuje się w ślimaczym tempie.

Dla mnie jest to po prostu okropny brak szacunku do czytelnika. Jak tylko napotykam na taką stronę, to o ile nie muszę, nie korzystam z niej. Dlatego staram się prowadzić swoją w sposób przeciwny do tych opisanych wyżej.

Na moim blogu nie ma żadnych zintegrowanych pluginów do fejsika, instagrama czy czegokolwiek innego, nie ma komentarzy, bo też nie są potrzebne akurat w moim wypadku. Grafik w newsach też raczej nie, chociaż może się coś gdzieś znajdzie, ale na dobrą sprawę jedyny obrazek, który tutaj jest, to moja gęba, która, moim zdaniem, tu być jednak powinna - piszę różne rzeczy, czasem mądre, czasem mniej mądre, ale nie wstydzę się podpisywać pod nimi swoim imieniem, nazwiskiem jak i wizerunkiem.

Na rzecz tego jestem w stanie porzucić anonimowość i sympatię do prywatności - by mój czytelnik wiedział, kto pisze to, co on czyta. Poza tym blog to też dla mnie możliwość budowanie wizerunku profesjonalnego, także dodatkowy plus.

Nie ma tu też, i nigdy nie będzie, reklam. Są jedynie linki do moich profili na linkedinie czy gitlabie. Tylko tyle.

Cała moja strona waży 25.3 kB, a by ją pobrać, wystarczą 4 requesty. Załadowanie jej zajmuje około sekundy. Wszystko serwowane po https. W dzisiejszych czasach to konieczne. I tak, wiem, że ktoś zaraz mi powie, iż to nie jest rok 2005, gdzie każdy kB się liczy, ale to nieważne. Dlaczego ja, jako użytkownik, mam marnować czas, czekając aż twój super nowoczesny SPA landing się załaduje po 10 sekundach? Nie ma opcji.

Średnia globalna prędkość łącza internetowego wynosi obecnie około 70-80 mbps. Mimo tego strony ładują się dalej w takim samym ślimaczym tempie. Jakim? Zazwyczaj jest to od 2 do 10 sekund z medianą na poziomie 6 sekund do całkowitego załadowania strony i 4 sekund do wyrenderowania jakiejś sensownej treści. TO DUŻO. Dla porównania całość mojej strony ładuje się w ~sekundę.

Całość jest cachowana za pośrednictwem cloudflara, więc CDN hula i przyśpiesza. Jak bardzo?

Nie znajdziecie również tutaj google analytics, czy jakiegokolwiek innego ustrojstwa do śledzenia tego, ile osób czyta ten blog i co tutaj robicie. Nie ma to dla mnie znaczenia - w

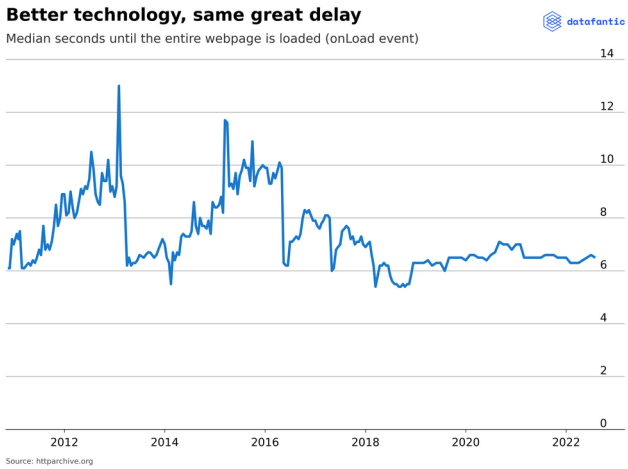

Figure 6: Średnia prędkość kompletnego załadowania strony

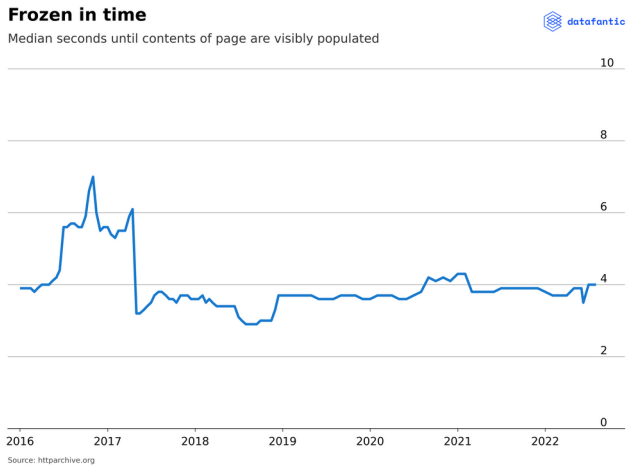

Figure 7: Średnia prędkość do wyrenderowania contentu

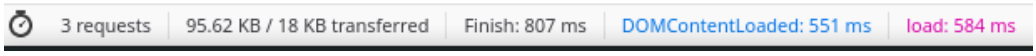

Figure 8: Tymczasem u grskiego na chacie...

Figure 9: 2022 rok i strona bez JSa. Da się? Da.

sumie to w ogóle nie ma znaczenia w przypadku blogów osobistych. Jedyne ciasteczko, które ode mnie dostajecie, to chyba to z cloudflara.

Jestem w stanie znieść artykuły czy treści sponsorowane, spoko, jakiś product placement. Normalna sprawa, trzeba zarobić. Są różne formy monetyzacji zasięgów czy treści, których można użyć, które sprawią, że i twórca zarobi, i czytelnik coś z tego wyniesie. I o to się rozchodzi, bo gdy jest inaczej, jest to po prostu nic innego, jak brak szacunku do czytelnika i tyle. Takich rzeczy nie szanuję.

To tak konkretnie, o cóż chodzi? O to, byś zastanowił się czasem, jako twórca, czy naprawdę potrzeba tej dwu megowej grafiki jako tło twojego bloga. Czy koniecznie wszędzie muszą być buttony do udostępniania na fejsie, insta, snapie albo łerewer. Czy koniecznie trzeba atakować użytkownika reklamami, newsletterami i innymi bzdetami.

To taki mój apel. Dbajmy o Internet, jak on wygląda i jak się prezentuje. To nasze wspólne dobro.

Do poczytania, źródło wykresów: https://www.datafantic.com/how-much-time-do-we-waste-waiting-for-websites-to-load/

23.5 Pierwsze dni pracy jako programista

Kilka słów o tym, jak pierwsze dni wyglądały, przynajmniej u mnie.

Zacznę może od tego, że zanim rozpoczęła się moja praca, miałem w głowie kilka obaw. Pierwszą z nich chyba było to, że sobie nie poradzę czy, że obowiązki mnie przytłoczą. Naturalne, chociaż niesłuszne. Dlaczego?

To zadaniem firmy i jej procesu rekrutacyjnego jest sprawdzenie, czy się nadajesz. Jeśli dostałeś się do pracy, a w CV nie kłamałeś i mimo tego sobie nie radzisz, to jest to wina firmy, nie twoja. Proste, zawiodła tu procedura rekrutacyjna, a nie ty. A co do zwykłego stresu, to chyba jest on normalny, nie ma się co przejmować. Jestem zdania, że wszędzie są ludzie, a z ludźmi idzie się dogadać.

Dobra, to jak to w końcu było?

Cóż, wbrew pozorom... Tak szczerze to spokojnie.

23.5.1 Spokojne pierwsze dni

Pierwsze dni w swojej pracy tak naprawdę spędziłem na instalowaniu potrzebnych mi rzeczy, czekaniu na dostęp do odpowiednich serwisów, ustawianiu środowiska, poznawaniu struktury firmy i tego, jak pracujemy, swoich współpracowników. Generalnie jeśli o programowanie chodzi to praktycznie nic. Naprawdę, pierwsze dni to spokój.

23.5.2 Wdrażanie

Dopiero gdy wszystko miałem już ustawione, to zacząłem coś programować. Na sam początek mój Project Leader, żeby mnie niejako wdrożyć do technologii, których używamy, przydzielał mi kolejne, stopniowo trudniejsze, zadania do zrobienia. Bardzo fajna sprawa. Dzięki temu później, gdy już zacząłem coś robić przy samym projekcie, to pewne rzeczy były znajome, to pozwoliło mi się czuć pewniej i spokojniej, to pomocne, zwłaszcza kiedy zaczynasz w nowym środowisku, uczysz się nowych rzeczy i chcesz dobrze wypaść.

Tutaj o technologiach nie będę się rozpisywał, bo są one inne dla każdego – zależy jakie stanowisko, jaki projekt i tak dalej. Jednak napiszę niżej o pewnych rzeczach, które chyba są uniwersalne. W każdym razie.

23.5.3 Ważna sprawa

Tutaj wychodzi też, jak ważny jest dobry Project Leader czy też przełożony/mentor. Ja trafiłem dość dobrze akurat, dzięki czemu naprawdę, moim zdaniem, moje wdrożenie w nowej rzeczywistości i pracy poszło dość przyjemnie. Wiadomo, trochę stresu mi towarzyszyło, bo zależało mi na tej pracy, ale mimo wszystko czułem się dobrze.

23.5.4 Atmosfera

Zaskoczeniem była nieco dla mnie atmosfera i sposób komunikacji w firmie. Otóż na początku miałem wyobrażenie, że będziemy wszyscy sobie nieco 'panować' na początku – per pan/pani, oficjalna mowa i tak dalej, ale... Nic bardziej mylnego. Od samego początku dość luźna atmosfera mnie ogarnęła. Nie wiem, jak wygląda to w innych firmach, gdyż ja pracuje w raczej średniej wielkości firmie, gdzie w sumie po prostu większość ludzi znasz i tak to właśnie wygląda.

Nikt się tu też raczej nie denerwuje. Ludzie są po prostu... pomocni. Był to dla mnie mały szok, bo wcześniej pracowałem trochę w budowlance, a tam ludzie bywają, no cóż, różni, ale nie tu. Dlatego też

23.5.5 Nie bój się prosić o pomoc

Na samym początku miałem wrażenie, zanim zacząłem, że będę trochę tak zdany na siebie – po prostu dostajesz zadanie i koniec, rób. Nie umiesz? Wypad.

Otóż po zaczęciu pracy okazało się, że nie – nic bardziej mylnego. Po pierwsze od czegoś są osoby starsze stażem i często chętnie pomagają, dzielą się wiedzą. Ba, częściej niż rzadziej za to mają przecież płatne. Można o pomoc poprosić.

Wiadomo, nie można latać i pytać o najdrobniejszą rzecz, ale gdy już naprawdę nie ma innego wyjścia, sami nie jesteśmy w stanie znaleźć rozwiązania i marnujemy czas, okazuje się, że są osoby chętne a nawet wyznaczone do pomocy.

Nie bójcie się zatem. Pytajcie. Nikt głowy ci nie ukręci za to, że czegoś nie umiesz, nie wiesz. Spokojnie. Jak pójdziesz, powiesz 'Hej, pomożesz mi z X? Próbowałem już tego i tamtego, ale jakoś mi nie wychodzi.' to problemu nie będzie raczej, ale jeśli będziesz latał do swojego przełożonego czy innego członka zespołu z najdrobniejszą pierdołą i liczył na gotowca, to... No cóż, będzie to irytujące na pewno.

Pamiętaj tylko żeby jakoś tam zapisywać rozwiązania tych problemów. Głupio jest pytać po raz któryś o to samo. Robienie notatek to świetna sprawa. Czasami wystarczy też po prostu przejrzeć historię wiadomości.

23.5.6 To oczywiste, że czegoś nie będziesz umiał

Jako juniorzy czy generalnie młodzi programiści, mamy raczej pewien zapas dystansu od szefa w prezencie. Chodzi mi o to, że raczej oczekiwane jest, iż nie będziemy wiedzieć

wszystkiego, czasem zdarzy się jakiś błąd, czy w jakimś temacie trzeba cię będzie dokształcić. Wydaje mi się, że każdy poważny pracodawca bierze to pod uwagę, zatrudniając nowicjusza/juniora.

To też powód, dla którego należy dostosować swoje wymagania. Wiadome jest, że trzeba mieć nieco szacunku do samego siebie i coś zarobić, po pracować za przysłowiowe dwa złote, to nie można, ale błędem jest też myślenie, że nauczysz się pisać Hello Worlda w JS'ie i ktoś ci potem da 15k netto za samo oddychanie. No nie, tak to nie działa.

W każdym razie.

Dlatego, jeśli czegoś nie umiecie, coś zrobicie źle, to nie próbujcie nieudolnie maskować swoich błędów czy niewiedzy, tylko mówcie szczerze i przyznawajcie się do błędów.

23.5.7 Szczerość

Właśnie o szczerości chciałem jeszcze coś powiedzieć. Nigdy nie kłamcie. Po prostu. Nakłamiesz w CV? Wyjdzie na rozmowie. Potem wieść się niesie, IT to wbrew pozorom wąskie grono, i będziesz spalony. Coś zepsujesz, nakłamiesz i to zatuszujesz? Też wyjdzie, prędzej czy później, zazwyczaj prędzej. Czegoś nie umiesz, ale oficjalnie to w sumie twoje zadanie jest już prawie zrobione? Nie zadziała, bo ostatecznie go nie skończysz i tylko sprawisz, że wszystko się niepotrzebnie przedłuży.

Wydaje mi się, że dużą częścią obowiązków w pierwszej pracy, to po prostu bycie szczerym i odpowiedzialnym za swoje czyny, nic więcej.

23.5.8 Rzeczy uniwersalne

Chciałbym tutaj wspomnieć o pewnych rzeczach, które możesz sobie obczaić nawet przed przyjściem do pracy, bo na 90% w twojej firmie też będą używane. Zazwyczaj to proste rzeczy, które od razu ogarniasz, ale im więcej znajomych interfejsów, tym pewniej się poczujesz i łatwiej będzie ten cały natłok informacji ogarnąć, więc wspomnę o kilku technologiach, z którymi warto się zapoznać gdzieś tam. Ba, może nawet na rozmowie jakoś tym zaplusujesz ;)

23.5.8.1 Jira/Confluence Z tego co wiem, dość popularnym systemem zarządzania taskami, jest Jira. Jak ona wygląda i co warto o niej wiedzieć, możesz łatwo znaleźć na internecie. Generalnie rocket science to tu nie ma – po prostu taki dashboard do łatwiejszego ogarniania co kto ma do zrobienia. Confluence to taka swego rodzaju baza wiedzy/dokumentacji.

23.5.8.2 Git/VCS W każdej firmie używany jest jakiś VCS, powinieneś już o tym wiedzieć. Jak trafiłeś gdzieś, gdzie się takich rzeczy raczej nie używa, a projekty przekazuje np. w archiwach czy wysyła mejlem, to... No coś jest nie tak. Generalnie VCS to konieczność i standard.

Obecnie takim standardowym VCS'em jest Git. Chyba większość go używa. Zdarzają się inne, ale to raczej rzadkość.

Do Git'a są różne 'serwisy', które świadczą usługi utrzymywania repozytoriów, czasami firmy robią to same, na swoich serwerach. W każdym razie najprawdopodobniej będziesz

korzystał z repo na GitHubie, GitLabie czy BitBucketcie. Możesz sobie przejrzeć, jak wyglądają te serwisy i zajrzeć co i jak.

Tak swoją drogą, to jeśli jeszcze nie masz konta na GitHubie czy gdzieś, ze swoimi projektami i kodem, to polecam ci się weń szybko zaopatrzyć. Podobnie, jeśli Gita nie znasz. Podstawy powinno się znać.

23.5.8.3 Slack/Google Meets/Microsoft Teams/Zoom Standardowe narzędzia do komunikacji. Do tego dochodzi jeszcze firmowy e-mail, który też pewnie otrzymasz, więc przed możesz sobie ogarnąć jak ustawia się Thunderbirda czy innego klienta pocztowego, ale to też raczej dla wybranych. Standardem raczej jest, że po prostu dostajesz nowego Gmaila albo Outlooka.

23.5.9 Podsumowanie

Tak właśnie wyglądały moje pierwsze dni w pracy. Czy w każdym przypadku tak to wygląda? Nie wiem, ale raczej podobnie. Rzeczy, które tu opisałem, mogą się co niektórym wydawać oczywiste, ale... No nie dla wszystkich takimi są.

Do obejrzenia: https://www.youtube.com/shorts/0P1RuLgxPNE

23.6 Nootropy - doping mózgu

Czasami zdarzy się tak, że jednak świat jest przeciwko nam i musimy z pewnych względów być wydajni, niekoniecznie wtedy, kiedy chcemy. Każdy podejmuje decyzje za siebie, jesteśmy dorosłymi ludźmi. Są pewne substancje i sposoby na to, by podbić naszą wydajność i efektywność w pracy. Opiszę tutaj dwie z nich, które są warte uwagi i rozważenia w kryzysowych sytuacjach.

Pierwszym lekiem jaki opiszę jest modafinil. Czym jest modafinil? Kojarzycie NZT z filmu Limitless z Bradleyem Cooperem? Trochę coś takiego, ale bez efektów ubocznych. Oficjalnie to lek na narkolepsje, który wzmaga czujność i koncentrację, zapobiegając zasypianiu. W skrócie - jedna tabletka, siadasz i przez 15h zamykasz się w pokoju i pracujesz. Modafinil jest lekiem i nie wykazuje właściwości euforyzujących jak na przykład kokaina, która niby ma podobne działanie. Pobudza mózg i neuroplastyczność, wzbudza koncentrację, daje energię do działania. 200 do 400mg zazwyczaj.

Nie będę wchodził w szczegóły tego, jak działa ten lek jak jest metabolizowany etc., napiszę tylko z doświadczenia, że działa. Siadasz, robisz milion więcej razy roboty. Tak po prostu. Nie polecam łączyć z kofeiną bo może być za mocno.

W Polsce oficjalnie średnio można dostać w aptekach, nieoficjalnie idzie kupić za przyzwoite pieniądze. Uwaga: nie namawiam tutaj do spożywania. Wspominam tylko, że istnieje coś takiego, na mnie działa, i może nieco tłumaczę jak to możliwe, że niektórzy menadżerowie czy prezesi pracują po 16h dziennie i mają dalej kopa. :-)

Są inne leki podobne do modafinilu, lecz np. jak adderal, aczkolwiek adderal jest tak naprawdę solą amfetaminy. Modafinil nie. Modafinil król nootropów jest jak lew, jak król dżungli.

Metylofenidat - kolejny stwór do poczytania.

Osobiście zdarza mi się modafinil zażywać kiedy potrzebuję siąść i bardzo dużo wydajnie popracować lub się czegoś trudnego nauczyć jak na przykład teraz, kiedy piszę te słowa.

24 Epilog

Chyba tyle z rzeczy, które miałem do powiedzenia.

Trochę się tego o dziwo nazbierało. Kurka, ponad 200 stron. Nie jest źle.

Wzruszający moment.

24.1 I co teraz?

Oprócz przeczytania tej książki, równocześnie lub nawet przed przeczytaniem polecam zapoznać się z następującymi pozycjami/zasobami:

1. https://realpython.com/ <- Tutaj świetnie podsumowane są wszelakie tematy, czy to list comprehensions, czy typy, czy printowanie, czy pętle. Szukaj tam wszystkiego i to tam doczytuj o szczegółach, których ja tu nie poruszyłem.
2. Learning Python, 5th edition - Mark Lutz <- Koniecznie przewertuj. Rozdziały możesz czytać wybiórczo, sam nie przebrnąłem przez całość, pomijając nieinteresujące mnie rozdziały, natomiast ta księga to cholerna encyklopedia. Polecam bardzo. Oprócz Real Python, gdzie masz treści nieco spłycone, tutaj rzeczy opisane są krok po kroku i baaardzo obszernie. Uważam, że jest to idealna baza do dalszego rozwoju i powinno się tę pozycję czytać równolegle lub nawet przed moją książką.
3. CS50 - chyba najlepszy kurs z podstaw Informatyki. Równolegle do tej książki oraz Learning Python.
4. Effective Python, 2nd edition - Brett Slatkin <- Świetna książka do przerobienia po tym, jak przeczytasz obecną pozycję oraz Learning Python to zajrzyj tutaj. Podszlifujesz swojego Pythona całkiem znacząco.
5. Fluent Python - Luciano Ramalho <- Książka podobna do Effective Python. Opcjonalnie.
6. Designing Data Intensive Applications - Martin Kleppmann <- Jak już skończysz tę książkę, cs50, Learning Python (część) oraz Effective Python, to fajnie rozszerzyć horyzonty i zajrzeć właśnie tutaj. Świetna książka o projektowaniu nie tyle samego kodu co i całego systemu wręcz.
7. Understanding Software - Max Kanat-Alexander <- Kolejna rzecz do przeczytania.
8. Zalando API Guidelines <- przejrzyj chociaż pobieżnie
9. PyCoder's Weekly / Python Weekly / UnKnown news <- Fajne newslettery, które co tydzień na mejla przysyłają ci zbiór ciekawych linków/artykułów i narzędzi ze świata Pythona i nie tylko.
10. python-awesome, django-awesome, fastapi-awesome <- repo na githubie, które zbierają listę popularnych, sprawdzonych i ciekawych projektów, które niejednokrotnie ułatwią Ci życie.
11. Sebastian Opałczyński <- polecam twórczość i różne projekty, też robi różne edukacyjne temaciki. Kursy, szkolenia etc. Kilogramy czystej wiedzy. To osoba, która ukształtowała mnie w dużym stopniu i wyciągnęła mnie za uszy do miejsca w którym jestem. Polecam. Gorąco.
12. Remote - Jason Fried <- o pracy zdalnej, która dziś jest nieodzownym elementem.
13. Blog: The Pragmatic Engineer

Pozycje poszerzające horyzonty, ale które można przeczytać nieco później.

1. Zrozumieć Programowanie - Gynvael Coldwind <- jeśli wydaje ci się, że w tej książce liznąłeś niskopoziomowych podstaw, to przeczytaj to wyżej. Pozycja wybitnie wartościowa, ale oznaczyłbym ją jako opcjonalną, bo do łatwych nie należy i można się zrazić. Natomiast warto jednak przerobić ten tom. Polecam też inną twórczość Gynvaela, bardzo inspirująca i znana postać na scenie.
2. Pragmatyczny programista - Thomas Hunt <- co ja się będę rozpisywał. Klasyk.
3. Bezpieczeństwo Aplikacji Webowych - Securitum <- Świetna książka o bezpieczeństwie w kontekście aplikacji webowych od światowej klasy ekspertów. Swoją droga ciekawostka - Polska ma jedną z mocniejszych reprezentacji jeśli o security idzie. Poland stonks.
4. Modern Software Engineering - David Farley <- O nowoczesnych sposobach wytwarzania oprogramowania.
5. Extreme Programming - Kent Beck <- Trochę o tym jak wygląda proces dostarczania oprogramowania i o sposobach kolaboracji.
6. Deep Work - Cal Newport <- o pracy w skupieniu.

Co do praktyki, to stay tuned - za niedługo powinienem wypuścić zbiór zadań, który warto przerobić by nabyć trochę praktyki. Zaglądaj na bloga/linkedin.

Uważam, że jeśli sumiennie przeczytasz i przerobisz materiały powyżej, zdobędziesz solidną wiedzę, pozwalającą ci startować już w jakiś rekrutacjach. Wymagane jest jednak to, by samodzielnie faktycznie wykonywać zadania, czytać ze zrozumieniem, tworzyć analogie i porównania, być w stanie się wypowiedzieć w danym temacie. Dodatkowo zaznaczę, iż wiem, że niektóre zadania obecnie mogą wydawać się trywialne, ale wciąż, zrób je. Chodzi o to by jak najbardziej przyzwyczaić cię do pisania kodu i zaznajomić ze standardową biblioteką Pythona.

24.2 To już jest koniec, nie ma już nic

Yup. Dziękuję za przeczytanie i powodzenia. Osobom, które zakupiły tę książkę oferuję mały bonus. Jeśli potrzebujesz się doradzić w jakiejś sprawie, to możesz do mnie podbić. Postaram się pomóc.

Mejl: olafgorski@pm.me 4programmers: @grski wykop: @grskii (dwa i) hejto: @grski github: @grski

Śmiało podbijajcie z pytaniami dotyczącymi kariery czy też może z prośbą o przeprowadzenie próbnej rozmowy rekrutacyjnej. Nie obiecuję, że każdemu pomogę, ale postaram się ile mogę. Rzetelnie też w razie czego ocenię Twoją wiedzę i dam feedback.

Co bardziej ogarniętych może gdzieś postaram się polecić. Kto wie.

Pamiętaj, że mi też ktoś kiedyś pomógł, a ja chcę przekazać pałeczkę dalej. Znaczy to tyle, że jestem tylko człowiekiem, a nie nie wiadomo kim. Jeśli do mnie napiszesz do spokojnie, nie gryzę. Postaram się pomóc najlepiej jak potrafię.

Nie zapomnij o https://github.com/grski/junior-python-exercises gdzie robiąc pull requesty do repozytorium możesz uzyskać dodatkowe wskazówki plus będzie to zawsze coś, co wzbogaci twojego githuba o kontrybucje ;)

Dziękuję, za przeczytanie.

Tyle ode mnie, miłego dnia i cześć.