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.

Contents

1 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.

1.1 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 _.

1.2 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 możnaby 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.

1.3 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 sobię 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!

1.4 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.

1.5 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...

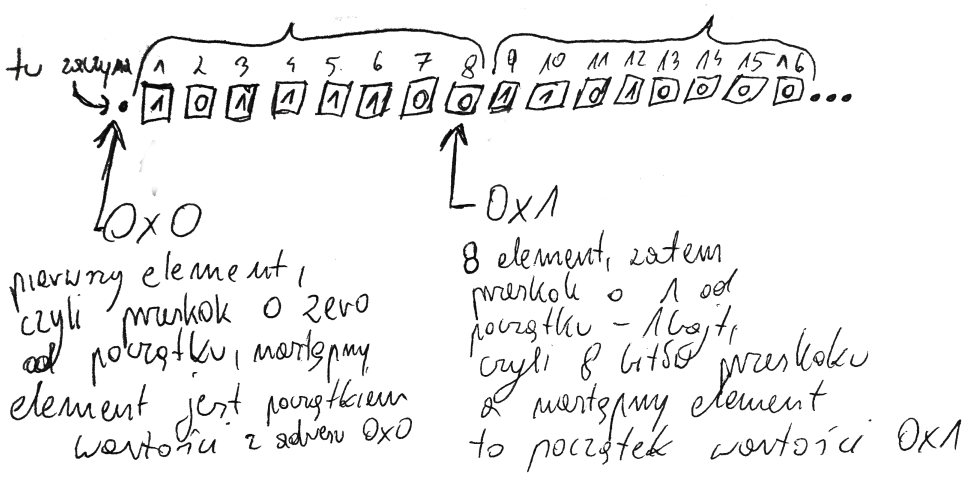

Figure 1: Przedstawienie pamięci

1.6 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!

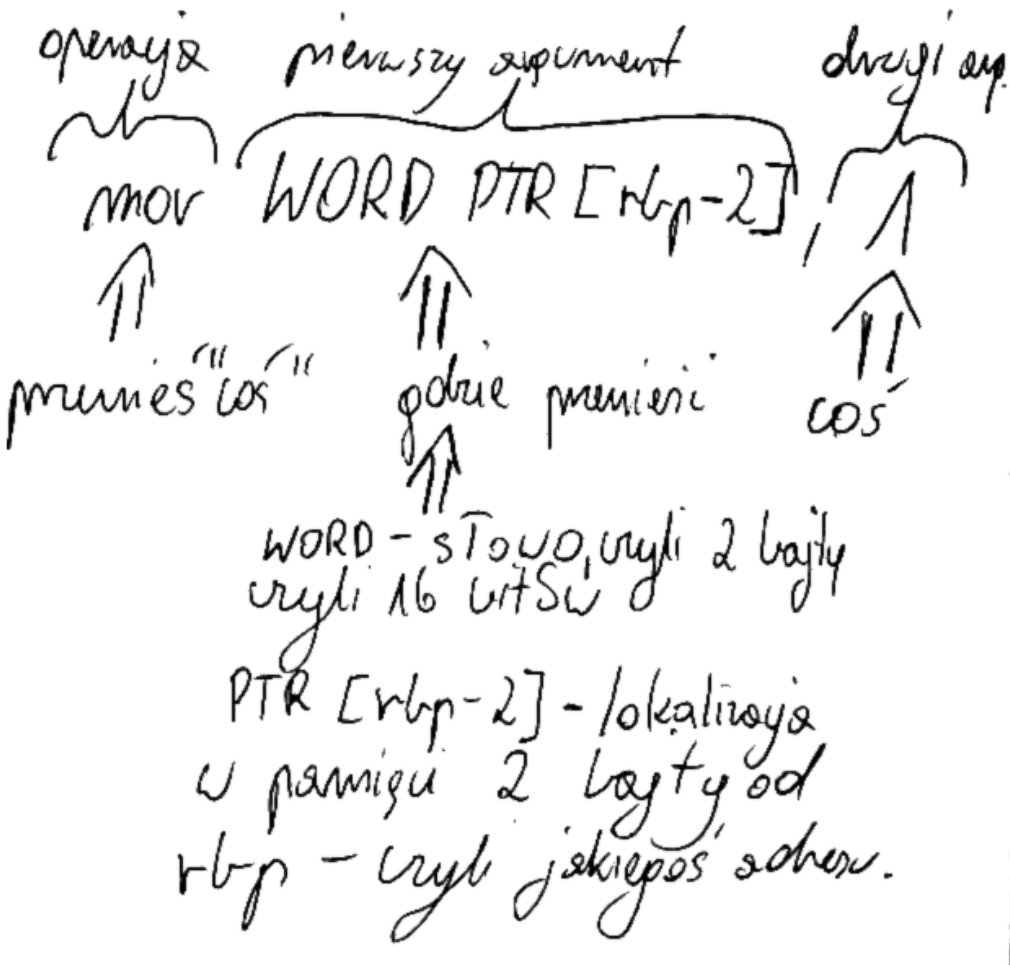

Figure 2: Rozpisane polecenie wyżej

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 implemenacji, 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 zastępywane 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.

1.7 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!

1.8 Inna inoszość

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 sobię 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.

1.9 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.

1.10 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 `pamięc RAM` to pleonazm językowy, gdyż RAM znaczy `Random Access Memory` zatem pisanie pamięc 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.

1.11 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.

1.12 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'.