

Programowanie z Górkim: Junior Python Developer

Czyli co warto wiedzieć jako nieopierzony junior.

Olaf Górski © 2022

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

Contents

1 Wstęp	12
1.1 O co tutaj chodzi?	12
1.2 Czyli w skrócie	14
1.3 Podziękowania	14
1.4 Dedykacja	17
2 Kilka słów do piratów i o prawach autorskich	18
3 O autorze	19
3.1 Górski, czyli inteligentny zwierz lądowy	19
3.2 Obecnie	19
3.3 Łatwo to nie będzie	20
3.4 Czyli tak reasumując	21
4 Wybór języka	22
4.1 Język niezbędny	22
4.1.1 Materiały	22
4.1.2 Komunikacja	23
4.1.3 Samo programowanie	23
4.1.4 Wyjątki	23
4.2 Jak się za to zabrać?	23
4.2.1 No, to jak?	24
4.2.2 Jak zatem powinno to wyglądać, panie mądralo?	24
4.3 Implementacja tego w praktyce	25
4.3.1 Tłumaczenie	25
4.3.2 Nauka słówek/fraz	25
4.3.3 Słuchanie	26
4.3.4 Mówienie	26
4.3.5 Czytanie	27
4.3.6 DeepL	27
4.4 Podsumowując	27
5 Python, o co chodzi?	28
5.1 Python – o co chodzi?	28
5.2 Jak korzystać z tej książki	28
5.3 Część interaktywna	29
5.4 Python 2 – Python 3?	29
5.5 Krótki opis długiej historii Pythona	29
5.6 Abdykacja Guido	30
5.7 Wężowe cele	32
5.8 Zalety Pythona	33
5.8.1 Ekspresywność	33
5.8.2 Prostota	36
5.8.3 Python językiem dynamicznie typowanym	36
5.8.4 Społeczność	37
5.8.5 Mnogość zastosowań	38
5.8.6 Czytelność	38
5.8.7 Automatyczne zarządzanie pamięcią	39

5.8.8	Wspieranie różnych paradymatów programowania	40
5.8.9	Wiele wspieranych platform	40
5.8.10	Dojrzałość	40
5.8.11	Prostota w integracji z innymi językami	40
5.8.12	Szybkość tworzenia kodu	41
5.9	Wady Pythona	41
5.9.1	Python językiem dynamicznie typowanym	41
5.9.2	Wydajność	42
5.9.3	GIL	43
5.9.4	Wysoka ekspresywność	44
5.9.5	Python nie istnieje w świecie mobile	44
5.9.6	Zbytnia wygoda	44
5.9.7	Tyle	44
5.10	Kto używa Pythona?	45
5.11	Python w porównaniu z	46
5.11.1	Java/C	46
5.11.2	Perl	47
5.11.3	C	47
5.12	Alternatywne implementacje	47
5.12.1	Stackless Python	48
5.12.2	Cython	48
5.12.3	PyPy	48
5.12.4	IronPython	49
5.12.5	Jython	49
5.12.6	Brython	49
5.12.7	MicroPython	49
5.12.8	CLPython	49
5.12.9	TinyPy	50
5.13	Podsumowanie	50
5.14	Pytania	50
6	Ustawiamy środowisko	51
6.1	W czym pisać kod na początku?	51
6.1.1	IDE? Edytor tekstu?	51
6.1.2	Podstawy	51
6.1.3	Jak to u mnie wyglądało	52
6.2	Windows?! Linux?! MacOX?! Co wybrać?	53
6.2.1	Instalacja potrzebnych nam rzeczy za pomocą choco	54
6.3	Instalacja na Linux/macOS	55
6.4	Przechodzimy do programowania! Nareszcie!	55
6.5	Czterej jeźdzy konsoli	55
7	Witaj, Świecie!	57
7.1	Wypisujemy tekst na ekran	57
7.2	Język binarny – jedyne, co rozumie komputer	60
7.3	Jak komputer widzi litery – system binarny	61
7.4	Kodowanie znaków – ASCII	62
7.5	I wtedy UNICODE I UTF-8 wchodzą całe na biało	64
7.6	Podsumowanie	66
7.7	Zadania i pytania	67

7.8	Odpowiedzi	67
8	Zmienne, wprowadzenie	69
8.1	Zapamiętywanie wartości	69
8.2	Nazwy zmiennych	69
8.3	No po co mi to wszystko?	71
8.4	Znowu trochę teorii	72
8.5	Heksadecymalne liczby	73
8.6	Kiedy przestać czytać?	74
8.7	Wszystko fajnie, ale w Pythonie inaczej	77
8.8	Inna inoszość	78
8.9	Zarządzanie pamięcią	78
8.10	Podsumowanie	79
8.11	Zadania i pytania	80
8.12	Odpowiedzi	80
9	Typy danych	82
9.1	Liczby	82
9.1.1	Krótką charakterystyką	82
9.1.2	Liczby całkowite	83
9.1.3	Przykładowy sposób reprezentacji liczb ujemnych	83
9.1.4	Liczby zmiennoprzecinkowe i niedokładność ich reprezentacji	84
9.1.5	Liczby złożone	86
9.1.6	Operacje na liczbach	86
9.1.7	Konwersje liczbowe	87
9.1.8	Przykłady podstawowych operacji na liczbach	89
9.2	Łańcuch znaków	90
9.2.1	Krótką charakterystyką	90
9.2.2	Pojedyncze znaki	92
9.2.3	Zmienne w tekście	92
9.2.4	Używanie zmiennych w tekście - wydajność	92
9.2.5	Przykłady podstawowych operacji na stringach	95
9.3	Bajty	96
9.3.1	Krótką charakterystyką	96
9.4	Typ logiczny/Boolowski	96
9.4.1	Krótką charakterystyką	96
9.4.2	Wartości prawdziwe vs wartości fałszywe	97
9.4.3	Przykłady podstawowych operacji na typie boolowskim	97
9.5	Listy	97
9.5.1	Krótką charakterystyką	97
9.5.2	Lista od strony niskopoziomowej	98
9.5.3	Referencje i wartości	98
9.5.4	Dynamiczny rozmiar list	99
9.5.5	Alokowanie ponad potrzeby - chciwy i przebiegły wąż	99
9.5.6	Dostęp do elementów listy	99
9.5.7	Negatywne indeksy	100
9.5.8	Cięcie list	100
9.5.9	Dlaczego indeksujemy od zera	102
9.5.10	Jakby to wyglądało, gdybyśmy indeksowali od 1?	104
9.5.11	Przykłady podstawowych operacji na listach	104

9.6	Krotki/Tuple	104
9.6.1	Krótka charakterystyka	104
9.6.2	Wydajna bestia	105
9.6.3	Przykłady podstawowych operacji na krotkach	105
9.7	Słowniki	105
9.7.1	Krótka charakterystyka	105
9.7.2	Jak przebiega proces dodawania elementów do dicta?	106
9.7.3	Kolizja hashy	106
9.7.4	Co może być kluczem?	107
9.7.5	Pass by value & Pass by reference	107
9.7.6	Kopia płytka i kopia głęboka a klucze w słowniku	109
9.7.7	dict.values() keys() items()	109
9.7.8	Przykłady podstawowych operacji na słownikach	109
9.8	Zbiory	110
9.8.1	Krótka charakterystyka	110
9.8.2	Przeszukanie szybsze niż na warszawskiej Woli	110
9.8.3	Przykłady podstawowych operacji na zbiorach	111
9.9	Podsumowanie	111
9.10	Pytania i zadania	112
10	Pętle i iteracja	113
10.1	Pętla krokowa	113
10.1.1	Krótka charakterystyka	113
10.1.2	Obiekt iterowalny	113
10.2	Zwykła pętla	115
10.2.1	Krótka charakterystyka	115
10.2.2	Przykłady użycia pętli	115
10.3	Składanie	115
10.4	Generatory	116
10.5	Walrus	117
10.6	Podsumowanie	118
10.7	Pytania i zadania	118
11	Funkcje	119
11.1	Zwykłe funkcje/metody	119
11.1.1	Śledzik na raz	122
11.2	Funkcje anonimowe/lambda	124
11.2.1	Gdzie używamy lambd najczęściej?	125
11.2.2	Przykłady	125
12	Klasy i OOP	127
12.1	Klasy	127
12.1.1	super() i MRO	129
12.2	Classmethods, staticmethods	130
12.3	Menadżery kontekstu	131
12.4	Typehints	132
12.5	Docstrings	133
12.6	Operator is	133
12.7	134

13 Ekosystem narzędzi Pythonowych	135
13.1 Plik README - co powinien zawierać i jak wyglądać	135
13.1.1 Technologia	135
13.1.2 Jakie sekcje powinien zawierać dobry plik README?	135
13.1.3 Tytuł	135
13.1.4 Opis projektu	135
13.1.5 Stos technologiczny	136
13.1.6 Instrukcja tworzenia środowiska lokalnego	136
13.1.7 Deployment	136
13.1.8 Autorzy	136
13.1.9 Podsumowanie	136
13.2 pdoc3	136
13.3 Pycharm/Visual Studio Code	136
13.4 Robienie notatek	137
13.5 Pyenv, poetry i inne nicponie	137
13.5.1 Pipetools	139
13.6 Walimy pythona sprzętem	139
13.6.1 Pipeliny	139
13.6.2 Co sprawia, że kod jest dobry?	140
13.6.3 Ciemna strona mocy - black	140
13.6.4 Isort	141
13.6.5 Importy jak wódka - Absolutne	141
13.6.6 Bandit	141
13.6.7 autoflake	142
13.6.8 pyupgrade	142
13.6.9 bumpversion	142
13.6.10 Git hooks & pre-commit	142
13.6.11 Podsumowanie	143
13.7 Pytania i zadania	143
14 Bazy danych	144
14.1 SQL	144
14.2 Relacyjne bazy danych	144
14.2.1 Tabele	145
14.2.2 Indeksy	145
14.2.3 Relacje	146
14.2.4 Normalizacja	146
14.2.5 Transakcje i współbieżność	147
14.2.6 Subskrypcje/Powiadomienia	147
14.2.7 Uprawnienia i bezpieczeństwo	147
14.2.8 Profilowanie	148
14.2.9 Kolejność kolumn	148
14.2.10 Podsumowanie	148
14.3 Tenanty i co to takiego	148
14.3.1 Perspektywa młodego	148
14.3.2 Perspektywa dziada	152
14.4 ORM	154
14.4.1 Czym jest ORM?	154
14.4.2 ORM vs czysty SQL	155
14.4.3 Podsumowanie	155

15 Internet	157
15.1 Droga żądania	157
15.2 CDN	158
15.2.1 Wprowadzenie	158
15.2.2 I wtedy wchodzi CDN, cały na biało	158
15.2.3 Szczegóły	159
15.2.4 Czym jest hit rate, lifetime?	159
15.2.5 Czy CDN to jeden ogromny serwer?	160
15.2.6 Podsumowanie	160
15.3 Cache	160
15.4 Chmura	161
15.5 Docker	161
15.6 Docker-compose	162
15.7 Docker Hub	163
16 Rekrutacja	164
16.1 Jak wygląda proces	164
16.1.1 Konkrety	164
16.1.2 Jakie pytania się pojawią?	165
16.1.3 Nie tylko wiedza	166
16.1.4 Rezultat	167
16.1.5 Poćwicz	168
16.1.6 Negocjacje	168
16.2 Studia a rekrutacja	168
16.2.1 Czy tak zwany papier jest nic niewarty?	169
16.2.2 Czy żałuję nie pójścia na studia?	169
16.2.3 Czy brak studiów nas przekreśla?	169
16.3 Jak szukałem pierwszej pracy	169
16.3.1 Spamer pierwszej wody	170
16.3.2 Jakieś rezultaty	170
16.3.3 Pierwsze rozmowy	170
16.3.4 Zdziwionko	170
16.3.5 Zdziwionko dwa	170
16.3.6 Okres próbny	171
16.3.7 Podsumowanie	171
16.4 CV	171
16.4.1 Moje ostatnie CV	171
16.4.2 Analiza	171
16.4.3 Zdjęcie	174
16.4.4 Podsumowanie	175
16.5 Co umiałem idąc do pierwszej pracy - studium przypadku	175
16.5.1 Googlование	175
16.5.2 Angielski	175
16.5.3 Algorytmy i struktury danych	176
16.5.4 Narzędzia	177
16.5.5 Słowniczek	177
16.5.6 Języki programowania	179
16.5.7 Frameworki	180
16.5.8 Wzorce projektowe	180
16.5.9 Testowanie	180

16.5.10 Systemy operacyjne	180
16.5.11 Jak już o systemach mowa	181
16.5.12 Debugowanie	181
16.5.13 Dobre praktyki	181
16.5.14 VCS - o gicie	182
16.5.15 Projekty	182
16.5.16 Blog	183
16.5.17 GitHub	183
16.5.18 Podsumowanie	183
17 Przykładowe pytania i zadania	184
17.1 Django	184
17.2 Python	184
17.3 Git	187
17.4 Http/Rest	187
17.5 Bazy danych	188
17.6 Ogólne koncepty programowania	188
17.7 Struktury danych	188
17.8 Kamień papier nożyce	188
17.9 Operacja na liczbach	189
17.10 Statystyki z logów	189
17.11 Statystyki zapytań	191
17.12 Książka zleceń	192
17.12.1 Format danych wejściowych	192
17.12.2 Format danych wyjściowych	192
17.13 Skracacz adresów	194
17.14 Generator statycznych stron	195
17.15 Jednolinijkowiec	199
18 Studium przypadku różnych aplikacji	201
18.1 Skracacz linków	201
18.2 Jak jedna cyferka może zepsuć aplikację - studium	202
18.2.1 Uff	202
18.2.2 Ścieżka pliku	203
18.2.3 Co się okazuje?	203
18.2.4 Zajrzyjmy do bazy	203
18.2.5 Format?	204
18.2.6 Eureka	204
18.2.7 Podsumowanie	204
19 Kultura	206
19.1 Trochę o menadżerach/liderach	206
19.1.1 Poziom pierszy: mistrz Excela	206
19.1.2 Poziom drugi: empatyczny akolita excela	207
19.1.3 Poziom trzeci: człowiek	208
19.2 Kultura projektowania	210
19.3 O wartościach	211
19.4 Kaizen	214
19.5 Zarządzanie portfelem inwestycyjnym	214
19.6 Relacja z prowadzenia pierwszego retro	214

19.6.1 Ready, set, go	215
19.6.2 Ustawienie sceny	215
19.6.3 Rozgrzewka	215
19.6.4 Retrospektywa w przeszłości	216
19.6.5 Dobrodziejstwa - zbieranie danych	216
19.6.6 Baddies - zbieranie danych	216
19.6.7 Happies - wciąż zbieramy dane	217
19.6.8 Przestań, to boli.	217
19.6.9 Incepcja	218
19.6.10 Podsumowanie	218
20 Papierologia i pieniądze	219
20.1 Netto, brutto, umowy, statusy, koszty	219
20.2 Umowa o dzieło	220
20.3 Umowa zlecenie	220
20.4 Umowa o pracę	221
20.5 B2B	221
20.5.1 Czym jest B2B	222
20.5.2 Zalety B2B	222
20.5.3 Wady	224
20.5.4 Podsumowanie	226
20.6 Pieniądze	227
20.6.1 Jak rozmawiać o pensji	227
20.6.2 Podwyżka != więcej na wypłatę, o czasie	228
20.7 Mój przypadek	229
20.7.1 Moje pierwsze zarobki, pierwsza podwyżka	229
20.7.2 Następna umowa i B2B	230
20.7.3 Dalsze losy	230
20.7.4 Odejście	231
20.7.5 Nowa okazja	231
20.7.6 Nic co dobre nie trwa jednak wiecznie	232
20.7.7 Powrót do rzeczywistości	232
20.7.8 Małe podsumowanie mojej historii zarobkowej	233
21 Wady i zalety pracy jako programista	235
21.1 Wady	235
21.1.1 Zdrowie	235
21.1.2 Ciągła nauka	235
21.1.3 Słabe projekty	236
21.1.4 Stereotypy	236
21.1.5 'Rynek pracownika'	237
21.1.6 Wymarzona praca	238
21.2 Zalety	239
21.2.1 Pitos	239
21.2.2 Możliwości rozwoju	239
21.2.3 Ciekawa praca	239
21.2.4 Zdalna również często	239
21.2.5 Rynek pracownika	240
21.3 Podsumowanie	240

22 Praca zdalna	241
22.1 Ergonomia	241
22.1.1 Mój ergonomiczny setup	241
23 Luźne przemyślenia	243
23.1 Backend to nie produkt	243
23.1.1 Małe oświadczenie	243
23.1.2 Jak powinno być	243
23.1.3 Apel	244
23.2 Chciej, pisz dobry kod	245
23.3 Obrzydliwe kuriozum w świecie IT	245
23.4 Szanujmy czytelników, szanujmy Internet	248
23.5 Pierwsze dni pracy jako programista	251
23.5.1 Spokojne pierwsze dni	251
23.5.2 Wdrażanie	251
23.5.3 Ważna sprawa	252
23.5.4 Atmosfera	252
23.5.5 Nie bój się prosić o pomoc	252
23.5.6 To oczywiste, że czegoś nie będziesz umiał	252
23.5.7 Szczerość	253
23.5.8 Rzeczy uniwersalne	253
23.5.9 Podsumowanie	254
23.6 Nootropy - doping mózgu	254
24 Epilog	255
24.1 I co teraz?	255
24.2 To już jest koniec, nie ma już nic	256

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 zespołe, gdzie masz pewność, że w przyszłości NA PEWNO nikt, kto polskiego nie zna, nie będzie czytał tego kodu (czyli nigdy), to okej. Teoretycznie można 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 sektora publicznego, realizowanych przez pewne duże korporacje, ale to nie do końca są projekty, w których chcesz się znaleźć. Zazwyczaj.

Czyli sprawia 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 – nazwę zmiennej nie powinien być cały poemat. Do tego w nazwach raczej używamy tylko liter, cyfr(rzadziej), podkreślenia. Tutaj konserwatywnie i bez szalu. Zwięzle, 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. Googlujecie 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. Ciągle 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.auth:
        to = rslv(stings.T0)
        if to == r.pt:
            raise VEr(
                "Redirection loop detected. Check that your"
                "REDIRECT_URL doesn't point to a login page."
            )
        return HttpResponseRedirect(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 głup 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 rzuć się 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 przejakrawiam 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. Założmy sobie, że mamy jakieś imię, które chcemy zapamiętać.

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

`Dobry wieczór. Coś się... coś się popsuło i nie było mnie słyszać, 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...")
```

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 – założmy, 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 zimportować. 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.

Otoż 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 wcześniej 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 chwilę 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 dłuża linijka z ponumerowanymi komóreczkami. Wyobraź sobie niesamowicie długi rzad 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 rozłazy? 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 mniej 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 zróbc analogicznie w systemie heksadecymalnym. Jak samodzielnie się nie uda, to pogoglądaj. 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...

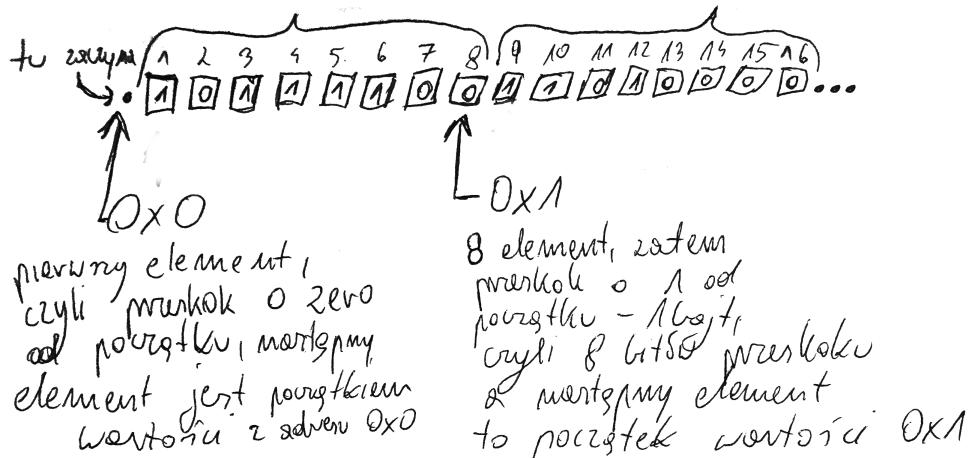


Figure 1: Przedstawienie pamięci

8.6 Kiedy przestać czytać?

Bo adres początkowy ma, ale końcowego tak niezbyt. Co teraz? Już mówię. Znowu wróćmy 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!

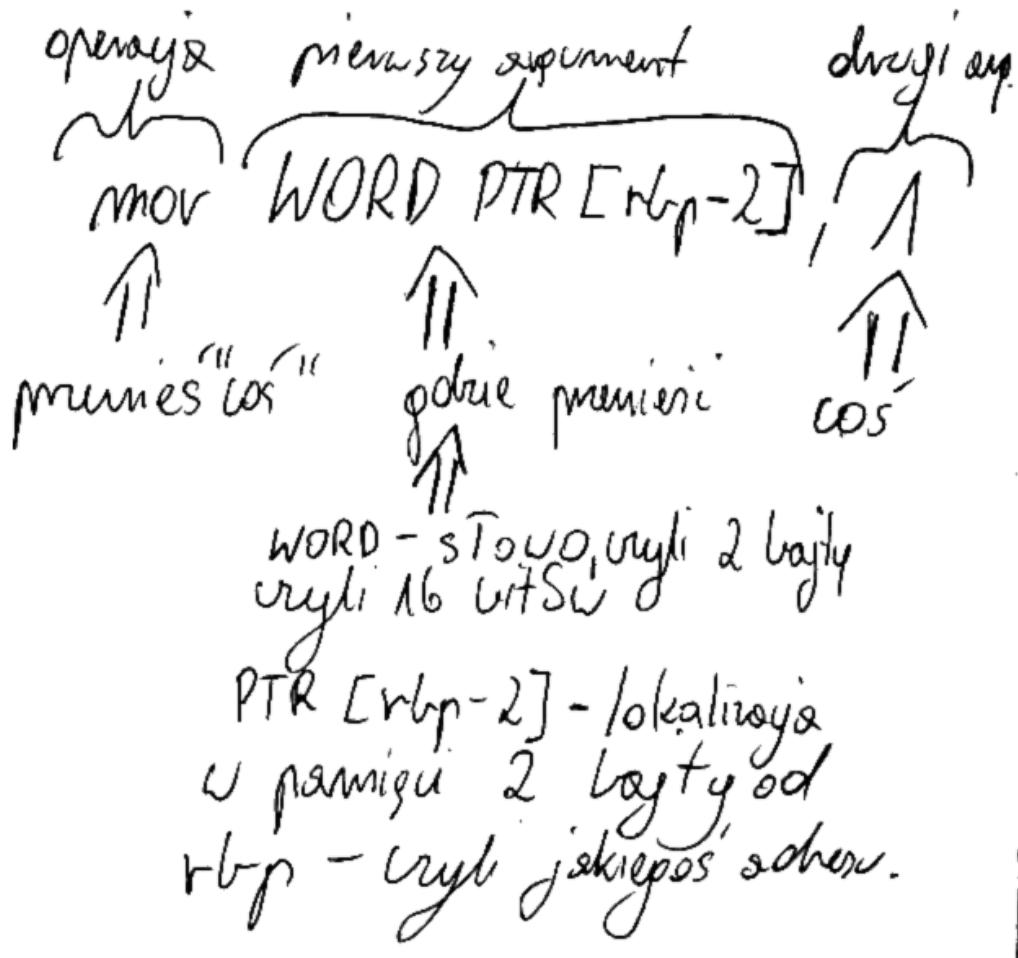


Figure 2: Rozpisane polecenie wyżej

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 niemniej, 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 komplikacji znika coś takiego jak `a`. Jego wystąpienia zastępuywane 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 komplikacji i tak dalej.

W Pythonie sprawa wygląda jednak trochę inaczej, jak już mówiłem. Wynika to z różnych natur tych dwóch języków, statycznie komplikowany C i dynamicznie interpretowany Python. Ta wiedza, którą ci tu przekazuję, jest jednak uniwersalna, bo o ile na wyższym poziomie, czyli tym, co jest płytka 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 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.

Otóz 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 komplikacja? Ale jak to można by zapytać. O co chodzi?

Otóz kod Pythonowy, który napiszesz też jest kompilowany, ale nie do kodu maszynowego. Python kompiluje się do bytecode'u 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 komplikacji. 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 dłużna binarna nawijka. Binarnie 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 komplikacji 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?

Otóz 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 jakiś **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 zlego!

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 niewielkimi 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 podstawię.
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’.