Programowanie
po 6 wykładzie

Andrzej Giniewicz

12.04.2024

W niniejszej części materiałów zajmiemy się obsługą czasu.

1 Czas w komputerze

Sytuacja z czasem w komputerach nie jest prosta[1] i nigdy nie była — a ostatnimi czasy ze względu na powszechność urządzeń mobilnych, jest jeszcze bardziej skomplikowana. Pierwszym problemem jest to, że istnieją dwa standardy zapisu. Wszystko zaczęło się od dwóch systemów operacyjnych: Unix (powstały w 1969 roku) oraz MS-DOS (powstały w 1981 roku na bazie systemu QDOS[2]). Unix już w wersji 4 wydanej w listopadzie 1973 roku zaczął zapisywać czas z uwzględnieniem stref czasowych[3], pamiętając czas i daty jako liczbę sekund od północy 1 stycznia 1970 roku w strefie czasowej UTC[4]. Bardzo szybko Unix stał się systemem dominującym we wczesnych infrastrukturach sieci ARPANET, która potem przerodziła się w Internet — standaryzacja czasu na zależny od czasu uniwersalnego zdecydowanie w tym pomogła, ponieważ w sieci użytkownicy korzystający z zasobów jednego komputera logują się na niego z różnych stref czasowych. Dla odmiany MS-DOS był systemem dla użytkowników domowych i zakładał, że „prosty użytkownik" ustawiający czas w komputerze będzie zdziwiony, że musi wiedzieć, w jakiej znajduje się strefie czasowej. Z tego powodu podjęto decyzję, że czas w MS-DOS będzie ustawiony na czas lokalny, czyli wpisany „po prostu" bez uwzględnienia strefy czasowej. Rozwiązanie takie zakładało, że komputer będzie zawsze pracował w strefie czasowej, w której został skonfigurowany, a jeśli to się zmieni, użytkownik sam przestawi zegarek. Pełne wyjaśnienie w oryginale brzmi wręcz zabawnie.

[1] O niektórych problemach możesz posłuchać tutaj — `https://www.youtube.com/watch?v=-5wpm-gesOY`.

[2] QDOS był skrótem od „Quick and Dirty Operating System".

[3] Informacja ta znajduje się w instrukcji obsługi Unix w wersji 4 dostępnej pod adresem `https://www.krsaborio.net/unix/research/acrobat/7311.pdf`, patrz dokumentacja komendy `time`. Przy okazji — Unix w wersji 4 był pierwszym systemem napisanym w języku C i co ciekawe, twórcą sporej części kodu Unixa był Dennis MacAlistair Ritchie, twórca języka C. Sam język C powstał właśnie do pisania systemów operacyjnych. Nieco więcej o języku C i Unixie możemy przeczytać w wielu źródłach, między innymi w encyklopedii Britannica `https://www.britannica.com/technology/C-computer-programming-language`.

[4] UTC oznacza czas uniwersalny, co odpowiada strefie czasowej Greenwich.

> *And if you explain to them, "No, you see, that time was UTC, not local time," the response is likely to be "What kind of totally propeller-headed nonsense is that? You're telling me that when the computer asks me what time it is, I have to tell it what time it is in London? (Except during the summer in the northern hemisphere, when I have to tell it what time it is in Reykjavik!?) Why do I have to remember my time zone and manually subtract four hours? Or is it five during the summer? Or maybe I have to add. Why do I even have to think about this? Stupid Microsoft. My watch says three o'clock. I type three o'clock. End of story."*[5].

O ile wydaje się to być tylko notką historyczną, to ten podział trwa do dziś. Obecnie większość systemów operacyjnych na komputerach i urządzeniach mobilnych pochodzi od jednego z tych systemów. MacOS, iOS, Linux, Android, BSD, Solaris i wiele innych to systemy z rodziny *nix, które bazują na filozofii i narzędziach wzorowanych na systemie Unix i w większości korzystają z czasu zapisanego w postaci czasu uniwersalnego UTC. Windows natomiast bazuje na MS-DOS i z powodu kompatybilności wstecznej do dziś korzysta z czasu lokalnego, co znacznie utrudnia jego działanie na urządzeniach mobilnych, które często zmieniają strefę czasową z powodu zmiany lokalizacji posiadacza, oraz na serwerach, które wymagają synchronizacji pomiędzy wieloma komputerami pracującymi w chmurze w wielu rozproszonych po świecie centrach danych.

Współistnienie dwóch systemów korzystających z różnych standardów zapisu czasu na jednym komputerze rodzi mnóstwo problemów. Gdy włączamy jeden z nich, ten przestawia zegarek drugiemu, co może powodować frustrację użytkowników, którym czas nieoczekiwanie zmienia się o godzinę. Szczególnie problem ten może się ujawnić przy przejściu z czasu letniego na zimowy lub odwrotnie, gdy każdy z włączonych systemów chce „skorygować" zegarek, efektywnie przesuwając go nie o godzinę a po godzinie. Aby temu zaradzić, systemy operacyjne kontaktują się z serwerami czasu przy włączaniu komputera, ale jeśli wyłączymy synchronizację czasu lub przy włączaniu komputera nie mamy dostępu do Internetu, co jest częste na urządzeniach mobilnych (Wi-Fi włącza się jako jedna z ostatnich usług), możemy się spodziewać dziwnego zachowania zegarka.

2 Czas w Pythonie

Python posiada dwie reprezentacje czasu, nazywane naiwną oraz świadomą. Reprezentacja naiwna nie posiada informacji o strefie czasowej, natomiast świadoma zapamiętuje czas wraz ze strefą czasową, w której się on znajduje. Na szczęście język oraz biblioteka standardowa `datetime` ukrywają przed nami znaczną część detali implementacji, ale rozróżnienie na czas w reprezentacji naiwnej i świadomej jest istotne, abyśmy mogli określić, jakie operacje na zmiennych zawierających czas są możliwe. Przykładowo, nie można obliczyć, ile sekund różnicy jest pomiędzy czasem w reprezentacji naiwnej i świadomej, ale można odjąć dwa czasy naiwne lub dwa świadome.

[5]Jest to cytat z oficjalnego blogu MicroSoftu, usunięty ze strony, ale dostępny w archiwum dawnych stron Internetowych pod adresem `https://tinyurl.com/blog-microsoft`.

Python w module datetime dostarcza kilku typów: date, time, datetime oraz timedelta. Oznaczają one kolejno: datę, godzinę, datę w połączeniu z godziną oraz różnicę pomiędzy chwilami dowolnego z trzech typów. Obiekty te są niemutowalne, czyli mogą być elementami zbiorów lub kluczami słowników. Najczęściej używanym obiektem jest datetime i na nim się skupimy w niniejszych notatkach. Więcej informacji o pozostałych typach można znaleźć w dokumentacji[6].

Dane dotyczące strefy czasowej pobierzemy z biblioteki pytz dostępnej w Anacondzie. Jest to biblioteka, która zajmuje się sterowaniem strefami czasowymi. Pozwala na przykład podać strefę czasową jako Europe/Warsaw. Dlaczego ma to znaczenie? Otóż strefa czasowa używana w Warszawie się zmieniała[7]. Biblioteka pytz jest bardzo często aktualizowana, ponieważ cały czas mają miejsce zmiany stref czasowych w różnych krajach[8]. Przykładowo trwają dyskusje czy nie powinniśmy zrezygnować z rozróżniania czasu letniego i zimowego, co jeśli będzie miało miejsce, będzie musiało być odnotowane w bibliotece.

Stwórzmy świadomy obiekt datetime wskazujący na moment początku naszych pierwszych zajęć w tym semestrze.

```
import datetime
import pytz

zajęcia = datetime.datetime(2024, 3, 1, 15, 15, 0)
Wrocław = pytz.timezone("Europe/Warsaw")
zajęcia_we_Wrocławiu = Wrocław.localize(zajęcia)

zajęcia_we_Wrocławiu
```

Na początek wykonujemy odpowiednie importy, tworzymy naiwny obiekt czasu i strefę czasową. Następnie lokalizujemy czas, uzupełniając go o strefę czasową.

Stwórzmy teraz czas reprezentujący Sylwestra w Tokio.

```
Sylwester = datetime.datetime(2024, 1, 1, 0, 0, 0)
Tokio = pytz.timezone("Asia/Tokyo")
Sylwester_w_Tokio = Tokio.localize(Sylwester)

Sylwester_w_Tokio
```

[6] Patrz https://docs.python.org/3/library/datetime.html.

[7] Ostatnia zmiana dla Polski miała miejsce w 1996 roku, wszystkie zmiany od 1800 roku można znaleźć w kodzie biblioteki pytz na stronie https://github.com/stub42/pytz/blob/release_2020.4/tz/europe#L2254 do 1988 oraz na stronie https://github.com/stub42/pytz/blob/release_2020.4/tz/europe#L563 od 1988 roku.

[8] Gdyby Polska zrezygnowała ze zmiany czasu z zimowego na letni, musiałoby to zostać odwzorowane w bibliotece pytz. Co ciekawe, należałoby to zrobić z dużym wyprzedzeniem, jeśli nie chcemy spowodować licznych problemów. Wyobraźmy sobie, co by się działo, gdyby nagle połowa systemów bankowych miała już wykonaną aktualizację informacji o strefach czasowych a połowa nie, przez co część transakcji byłaby rejestrowana ze złą godziną z przyszłości. Mogłoby to prowadzić do licznych błędów i problemów. Taki zabieg wymaga bardzo precyzyjnego planowania i będzie kosztowną operacją.

Możemy teraz sprawdzić dokładnie jaka jest różnica czasu pomiędzy zajęciami we Wrocławiu a Sylwestrem w Tokio.

```
print(zajęcia_we_Wrocławiu - Sylwester_w_Tokio)
```

W odpowiedzi uzyskamy informację, że daty te dzieli 60 dni, 23 godziny i 15 minut, czyli jadąc ze średnią prędkością 8,36km/h powinniśmy być w stanie pokonać dzielące nas do Tokio 12237,31km drogą lądową (innymi słowy, gdyby ktoś spędził Sylwestra w Tokio, zdążyłby wrócić na zajęcia z programowania, jadąc na rowerze).

Tę samą datę możemy wykorzystać w więcej niż jednym wyrażeniu. Możemy na przykład sprawdzić, jaka jest różnica pomiędzy Sylwestrem we Wrocławiu i w Tokio.

```
print(Wrocław.localize(Sylwester) - Sylwester_w_Tokio)
```

W odpowiedzi uzyskamy informację, że Sylwester we Wrocławiu jest 8 godzin później niż w Tokio.

Zwróćmy uwagę, że różnica pomiędzy czasami w obu powyższych przykładach, jest wartością typu `timedelta`. W `timedelta` niezależnie pamiętana jest liczba dni, sekund i milisekund, przez co dla ujemnych wartości możemy zobaczyć nietypowe wyniki, na przykład różnica „minus 8 godzin" to „minus 1 dzień plus 16 godzin".

Spróbujmy teraz pobrać obecny czas z zegarka komputera i użyć go do stworzenia czasu świadomego, podając strefę czasową. Po dodaniu strefy czasowej sprawdzimy, która teraz jest godzina w Tokio.

```
teraz = datetime.datetime.now()
tu_i_teraz = Wrocław.localize(teraz)
teraz_w_Tokio = tu_i_teraz.astimezone(Tokio)

print(teraz_w_Tokio)
```

Sprawdzanie, jaki jest obecnie czas w innej strefie czasowej, możemy wykonać za pomocą metody `astimezone`.

Oprócz podania czasu wprost jako parametru do `datetime` oraz pobrania obecnego czasu metodą `now`, możemy tworzyć daty na podstawie napisów. Służy do tego metoda `strptime`[9].

Ponieważ istnieje wiele sposobów zapisu dat, metoda ta przyjmuje jako argument napis zawierający datę oraz format. Do najważniejszych formatów należą

[9] Nazwa `strptime` pochodzi od „*STRing Parse TIME*".

Wartość	Kod
rok	`%Y`
miesiąc	`%m`
dzień	`%d`
godzina	`%H`
minuta	`%M`
sekunda	`%S`

Spróbujmy stworzyć obiekt `datetime` dla daty i godziny wskazującej wschód słońca we Wrocławiu w najbliższy międzynarodowy dzień pizzy, zapisanej jako napis `"09.02.2025 07:13:00"`. Zapisane są w niej kolejno dzień, miesiąc, rok, godzina, minuta i sekundy — co możemy wyrazić formatem `"%d.%m.%Y %H:%M:%S"`.

```
datetime.datetime.strptime("09.02.2025 07:13:00", "%d.%m.%Y %H:%M:%S")
```

Odwrotną operacją służącą do niestandardowego formatowania czasów jest `strftime`[10].

```
datetime.datetime.strftime(tu_i_teraz, "%d.%m.%Y %H:%M:%S")
```

Więcej informacji o formatowaniu znajdziemy w dokumentacji[11]. Jako ćwiczenie sprawdź, ile czasu upłynie, od końca ostatnich zajęć, do wschodu słońca w najbliższy międzynarodowy dzień pizzy.

[10] Nazwa `strftime` pochodzi od „*STRing Format TIME*"

[11] Na przykład na stronie `https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes`.