zaj3-blok4

October 26, 2015

1 Instalacja oprogramowania w pythonie

1.1 Długa dygresja o ABI

Wyjaśnienie:

- API (*Application Programming Interface*): Definicja funkcji, metod, typów modułu jakiegoś modułu, do której musi stosować się **kod źródłowy** innych modyułów by moduły te mogły współpracować.
- ABI (*Application Binary Interface*): Definicje funkcji, metod, typów, format wywołania funkcji, jakiegoś modułu do której musi stosować się **skompilowany kod** innych modułów by moduły te mogły współpracować.

ABI jest problemem tylko przy tworzeniu bibliotek w językach kompilowanych (C/C++/...). W Javie/Pythonie ten problem nie istnieje na poziomie maszyny wirtualnej.

API i ABI są od siebie niezależne program o tym samym API skompilowany pod windowsem będzie miał inne ABI niż skompilowany pod linuksem. Dwie wersje biblioteki o tym samym API mogą mieć inne ABI.

In [64]:
```
%%cpp

#include <iostream>
#include <cstdint>
// Struktura foo należy do biblioteki foo w wersji 1.0
struct foo{
    uint16_t a;
};

// Moja funkcja która z niej korzysta
int unpack_foo(foo arg){
    return arg.a;
}

int main(){
    foo f = {1};
    std::cout << unpack_foo(f);
}
```

```
b'1'
```

In [65]:
```
%%cpp

#include <iostream>
#include <cstdint>
// Struktura foo należy do biblioteki foo w wersji 2.0
struct foo{
```

```
        uint16_t a, b;
    };

    int unpack_foo(foo arg){
        return arg.a;
    }

    int main(){
        foo f = {1};
        std::cout << unpack_foo(f);
    }
```

```
b'1'
```

Zauważmy że API struktury foo się nie zmieniło. Mój stary program po dodaniu nowego pola do struktury foo, poprawnie się kompiluje.

Problemy zaczynają się kiedy zarówno mój program jak i biblioteka foo są skompilowane a mój program został skompilowany przy użyciu starej wersji biblioteki!

Dlaczego?

Ponieważ **rozmiar struktury jest częścią ABI**. Użytkownik biblioteki musi móc np. zdefiniować tablicę 10 struktur foo, w wersji 1.0 tablica ta będzie miała rozmiar 20 bajtów a w wersji 2.0: 40 bajtów.

Tablica ta może istnieć **na stosie** więc jej długość musi być znana podczas kompilacji.

Dodatkowo do przekazania struktury (jako parametru) do funkcji trzeba znać jej rozmiar.

1.2 Utrzymywanie kompatybilności ABI

Utrzymywanie kompatybilności ABI jest uciążliwe i bardzo utrudnia rozwój biblioteki. Podejście do kompatybilności ABI bardzo zależy od wielu czynników, ale zasadniczo projekty open-source (poza wyjątkami) stwierdzają że utrzymywanie kompatybliności ABI jest zbyt kosztowne, szczególnie że każdy może sobie skompilować kod

Przykłady:

- Twórcy bibliotek standardowych C/C++ starają się utrzymywać ABI programów skompilowanych różnymi wersjami.
- Jądro Linuksa trzyma ABI wywołań systemowych (czyli program skompilowany dla jądra 3.0 powinien działać i dziś).
- Jądno Linuksa nie trzyma ABI modułów jądra (czyli własnościowe sterowniki nVidii skompilowane pod jądro 3.14 nie będą dziać pod 3.15 — jest to główny problem ze wspomaganiem 3D na Linuksie...).
- **Python** nie trzyma ABI swoich interfejsów C, poza bardzo ograniczonym Stable ABI wprowadzonym dość niedawno. Niestety stable ABI jest zbyt "okrojone" by Numpy mógł z niego korzystać.

1.3 Wirtualne środowisko

Wirtualne środowisko to: izolowany interpreter pythona, który pozwala pracować nad projektem bez martwienia się o interakcje z innymi projektami.

Wirtualne środowisko składa się z:

- Linku do Interpretera Pythona
- Zainstalowanych paczek Pythona
- Skompilowanego kodu C łączączego się z Pythonem

Zachęcam do pracy z pythonem w taki sposób:

- Kompilujecie interpreter pythona
- Tworzycie wirtualne środowisko dla każego projektu

- W każdym projekcie instalujecie oddzielne paczki

Procedura ta wyklarowała mi się po czterech latach pracowania nad długożyjącymi projektami :)
Ma ona jedną zasadniczą wadę: zużywa miejsce na dysku.

```
In [2]: %%bash

        find $HOME -name "python" | wc -l #Ilość interpreterów Pythona które mam w /home
        # W ramach ciekawostek w zeszłym roku było ich 72
```

```
110
```

```
find: `/home/jb/programs/kaskadypy/.git/hooks': Permission denied
```

Są takie alternatywne procedury:

- Używać Pythona i całego oprogramowania z systemowych paczek na Linuksie.

Jest to problem ponieważ wasz program może działać na Django-1.8 a twórca systemu może nagle postanowić zainstalować 2.0 które ma poważne zmiany w API.

- Używać wirtualnego środowiska z pythonem systemowym.

Paczki mogą przestać działać jeśli wasz system zmieni wersję pythona (np. z 3.4.1 na 3.4.2) co zmieni ABI kodu C w pythonie i paczki przestaną działać (powoduje to konieczność rekompilacji środowiska wirtualneg)

- Używać jednego zainstalowanego przez Was interpretera i wirtualnego środowiska w kilku projektach.

Zasadniczo może mieć tą wadę że instalując paczki do projektu A zepsujecie projekt B.

2 Kompilacja Pythona

Najpierw należy pobrać i wypakować Pythona:

```
In [1]: %%capture
        %%bash

        cd /tmp
        rm -rf pyth-example
        mkdir pyth-example
        cd pyth-example
        wget https://www.python.org/ftp/python/3.5.0/Python-3.5.0.tgz
        tar -xaf Python-3.5.0.tgz
```

Dobrym pomysłem jest zainstalowanie zależności interpretera: http://db.fizyka.pw.edu.pl/pwzn/kompilacja-pythona.html.

Polecenie `./configure` konfiguruje kompilację pythona, ma dużo fajnych opcji (polecam `configure --help`). Najważniejszą rzeczą jest skonfigurowanie katalogu **do którego** instalujecie Python, służy do tego opcja `--prefix`.

Następnie należy zbudować i zainstalować projekt.

```
In [2]: %%capture
        %%bash

        cd /tmp/pyth-example
        cd Python-3.5.0
        ./configure --prefix /tmp/pyth-example/python-installed
        make -j8
        make install
```

In [3]:
```
%%bash

/tmp/pyth-example/python-installed/bin/python3 -c "import datetime; print(datetime.datetime.now(
```

```
bash: line 2: /tmp/pyth-example/python-installed/bin/python3: No such file or directory
```

3 Na Windowsie

Na windowsie pobierzcie Pythona 3.4.X: https://www.python.org/downloads/ i zainstalujcie.

3.1 Stworzenie wirtualnego środowiska

By stworzyć wirtualne środowisko należy wywołać polecenie pyvenv przyjmuje ono nazwę katalogu w którym istnieć będzie środowisko

In [4]:
```
%%capture
%%bash

cd /tmp/pyth-example
python-installed/bin/pyvenv venv

ls venv
echo
echo "Zawartość lib"
echo
ls venv/lib/python3.4/site-packages
```

3.2 Włączenie wirtualnego środowiska

Polecenie `source plik` jest odpowiednikiem wykonanie poleceń z tego pliku w aktualnej konsoli. Do aktywowania środowiska wirtualengo Pythona służy skrypt bin/activate.

Po aktywacji środowiska w danej konsoli używany jest Python z danego środowiska z paczkami zainstalowanymi do tego środowiska.

In [5]:
```
%%capture
%%bash

cd /tmp/pyth-example
source venv/bin/activate

which python
```

3.3 Na Windowsie

Uruchomić:

- W PowerShell: /Scripts/Activate.ps1
- W CMD: /Scripts/activate.bat

3.4 Instalacja oprogramowania

pip install nazwa-paczki

pip install nazwa-pliku-wheel

In [ ]: