Programowanie
po 7 wykładzie

Andrzej Giniewicz

19.04.2024

W ostatniej porcji notatek po wykładzie, zajmiemy się instalowaniem i tworzeniem bibliotek oraz skryptów.

Na poprzednim kursie korzystaliśmy z notatników Jupytera. Jest to wygodne narzędzie do pracy interaktywnej, a praca matematyka z komputerem najczęściej jest interaktywna. Gdy przeprowadzamy obliczenia, wykonujemy symulacje lub analizujemy dane, interaktywność jest dużym plusem. Dodatkowym walorem notatników jest łączenie komórek tekstowych i kodu w jednym dokumencie, poprzez co uzyskujemy swojego rodzaju artykuł, który możemy uruchomić[1]. Dodatkowymi zaletami jest to, że licząc coś, czego wcześniej nie liczyliśmy, prowadząc symulacje, których wyników jeszcze nie znamy i analizując dane, które widzimy po raz pierwszy, nie jesteśmy w stanie przewidzieć z góry, jakie kroki są konieczne do wykonania naszego zadania. Gdybyśmy musieli napisać całą aplikację od początku do końca i za każdym razem uruchamiać cały skrypt, jego początek prawdopodobnie uruchomilibyśmy wiele razy, podczas gdy wystarczyłby raz w jednej komórce, po czym dalsze eksperymenty możemy wykonywać w kolejnych komórkach. Taki tryb pracy jest wygodny, gdy chcemy prowadzić badania interaktywnie. Ale to nie zawsze jest prawda. Czasem po opracowaniu pewnej funkcjonalności w notatniku, chcemy móc jej użyć gdzieś indziej. Mamy wtedy dwie opcje — pierwsza, to stworzenie własnej biblioteki, druga, to stworzenie skryptu.

Skrypty i biblioteki, to dwa klasyczne sposoby opakowywania kodu. Biblioteki zawierają klasy, stałe, funkcje i dokumentacje, które mogą być użyte przez osoby piszące programy — autora biblioteki, lub jeśli ją udostępni[2], przez inne osoby. Skrypty są przeznaczone natomiast do bezpośredniego uruchamiania, nie tak jak biblioteki, do wykorzystania w innych programach. Python pozwala na tworzenie zarazem skryptów jak i bibliotek. Na początek, każda osoba może chcieć opakować często używane funkcje w bibliotekę, aby nie kopiować ich z notatnika do notatnika. Ma to więcej zalet, często jakąś funkcję udoskonalamy z czasem. Jeśli będzie w bibliotece, zmieniamy ją w jednym miejscu, a korzystają z tej zmiany

[1] Dokumenty interaktywne z komórkami są jednym z częściej stosowanych filarów powtarzalnych badań. Osoby zainteresowane zachęcam do zgłębienia tego, jak powtarzalne badania i Jupyter pomagają w realizacji celów otwartej nauki — ciekawa książka zatytułowana „The Open Science Training Handbook" dostępna on-line znajduje się pod adresem `https://book.fosteropenscience.eu/`.

[2] O tym jak stworzyć pakiet z własnej biblioteki możemy przeczytać w książce „Python Packaging User Guide" dostępnej pod adresem `https://python-packaging-user-guide.readthedocs.io/`.

wszystkie programy, w których jej użyjemy. Główną zaletą skryptu jest natomiast możliwość napisania aplikacji, która uruchamiana jest z poziomu systemu operacyjnego, nie z poziomu notatnika i przeglądarki. Dowolne programy posiadające interfejs graficzny działają lepiej, gdy uruchamiane są niezależnie od platformy notatników. Również, jeśli chcemy przesłać nasz kod, aby uruchomiony został na innym komputerze[3], skrypty będą odpowiedniejszą formą, niż notatniki.

1 Tworzenie bibliotek

Istnieją dwa rodzaje bibliotek. Pierwszy typ składa się z jednego modułu, czyli jednego pliku `.py`. Drugi nazywa się pakietem i składa się z wielu modułów umieszczonych w jednym katalogu. Aby katalog był pakietem, musi się w nim znajdować plik `__init__.py`, nawet jeśli jest on pusty. My zajmiemy się tylko bibliotekami opartymi na jednym module[4].

Otwórzmy Jupytera i wybierzmy z menu `File -> New -> Text file`. Następnie wybierzmy `File -> Save File As...` i w oknie, które się pojawi, wpiszmy `biblioteka.py`. Plik domyślnie utworzy się w katalogu, w którym jest uruchomiony Jupyter, czyli w katalogu roboczym. Katalog roboczy jest jednym z katalogów, w którym Python szuka modułów, więc powinien znaleźć go bez trudu. Wpiszmy teraz jakąś zawartość do modułu.

```
def dodawanie(x, y):
    return x+y
```

Funkcja ta nie jest może szczytem naszych możliwości, ale wystarczy dla ilustracji. Zapiszmy plik `biblioteka.py` i przejdźmy do notatnika (możemy utworzyć nowy lub wybrać istniejący). W nowej komórce wpiszmy.

```
from biblioteka import dodawanie
print(dodawanie(3, 4))
```

Naszym oczom powinna ukazać się liczba 7. Właśnie stworzyliśmy naszą pierwszą bibliotekę, której możemy używać w wielu różnych notatnikach! Zastanów się, czy podczas realizacji list zadań, trafiła się taka sytuacja, że fragment kodu był kopiowany pomiędzy nimi? Czy jest to fragment, który warto wyciągnąć do osobnej biblioteki?

Warto wskazać pewne ograniczenie — jeśli zmienimy kod biblioteki, Jupyter nie zaimportuje drugi raz tej samej biblioteki. Najłatwiej wtedy zrestartować jądro w menu `Kernel -> Restart Kernel...`, co spowoduje wyczyszczenie całego środowiska i import ponownie zadziała. Niestety tracimy wtedy dostęp innych zdefiniowanych w środowisku zmiennych i musimy ponownie wywołać niektóre komórki.

[3] Na innym komputerze lub nawet superkomputerze, takim jak dostępne we Wrocławskim Centrum Sieciowo-Superkomputerowym, szczegóły na — `https://www.wcss.pl/?c=static_kdm&cid=19`.

[4] Więcej o tworzeniu złożonych pakietów można przeczytać w materiałach zatytułowanych „The Joy of Packaging" `https://python-packaging-tutorial.readthedocs.io/en/latest/setup_py.html`.

Kod biblioteki nie powinien wykonywać instrukcji, ponieważ jeśli jest w nim na przykład jakiś `print` na poziomie pliku, tekst będzie się pojawiał za każdym razem, gdy moduł zaimportujemy. Zobacz, co się stanie, jeśli w module umieścisz (niezalecany) kod

```
def dodawanie(x, y):
    return x+y

print(”tego tu być nie powinno”)
```

Po wprowadzeniu zmiany zapisz plik, wróć do notatnika i spróbuj zaimportować funkcję `dodawanie`. Pamiętaj zresetować jądro, jeśli wcześniej importowałeś już tę bibliotekę.

Komendy wykonywane w pliku `.py` są charakterystyczną cechą skryptów.

2 Skrypty

Skrypty w Pythonie to pliki z rozszerzeniem `.py`, które posiadają w sobie komendy, które zostaną wykonane po uruchomieniu skryptu. To, co było niezalecane dla bibliotek, jest wręcz najważniejszym elementem skryptu. Skrypt, który ma tylko definicje, efektywnie zdefiniuje funkcje i nic nie zrobi. Zmodyfikujmy wiadomość w pliku `biblioteka.py`.

```
def dodawanie(x, y):
    return x+y

print(”Witaj w skrypcie”)
```

Po wprowadzeniu zmian zapiszmy plik i otwórzmy nowy terminal wewnątrz Jupytera klikając `File -> New -> Terminal`. W okienku, które się pojawi, uruchamiamy skrypt

```
python biblioteka.py
```

Po wciśnięciu klawisza Enter, na ekranie pojawi się napis. Więcej sposobów na uruchamianie skryptów, poznamy podczas kursu „Zaawansowane techniki programowania” na drugim semestrze.

3 Skrypt i biblioteka w jednym

Często chcemy, aby jakiś moduł był jednocześnie skryptem i biblioteką. Na przykład, gdy uruchomimy go jako skrypt, może wyświetlać okno, w którym wybieramy myszką pliki do zamiany kodowania, a gdy zaimportujemy go jako bibliotekę, może udostępniać nam funkcje do wykonywania tych samych czynności z poziomu naszego własnego programu. Oczywiście, nie chcemy, aby przy każdym imporcie pojawiało się nowe okienko, ani aby do odpalenia okienka, trzeba było umieć programować i napisać własny program, który

wszystko uruchamia. Z pomocą przychodzi specjalna zmienna `__name__`, którą Python ustawia na wartość "`__main__`", gdy program jest uruchomiony jako skrypt. To pozwala nam na napisanie fragmentu kodu, który uruchamia się lub nie, zależnie od tego, jak wykorzystamy nasz moduł. Zmodyfikuj plik `biblioteka.py` aby wyglądał następująco

```
def dodawanie(x, y):
    return x+y

if __name__ == "__main__":
    print("Witaj w skrypcie!")
```

Spróbuj teraz zaimportować go w notatniku oraz uruchomić jako skrypt. Powinien zadziałać zgodnie z oczekiwaniami w obu przypadkach.

Dokładnie ten sam mechanizm wykorzystywany jest przez wszystkie biblioteki. Zarazem te standardowe, jak na przykład moduł `math`, przez te dodatkowe, ale zainstalowane razem z dystrybucją Anaconda, jak moduł `numpy`, oraz takie, które doinstalujemy samodzielnie, odnajdując je na stronie `https://pypi.org/`, na której znajduje się ponad 200 tysięcy projektów.

4 Instalacja dodatkowych bibliotek

Ogólna zasada instalacji jest taka, że jeśli pakiet jest dostępny w oficjalnym kanale dystrybucyjnym Anacondy (i korzystamy z Anacondy), wybieramy instalację za pomocą komendy `conda`. Nie jest to co prawda standardowa metoda, ale zwykle dostarcza bardziej zoptymalizowanych bibliotek, niż w przypadku innych metod instalacji. W naszym przypadku chcemy zainstalować bibliotekę PyGame, która jest w Anacondzie, ale jedynie starsza jej wersja i w dodatku jest to nieoficjalna dystrybucja (kanał dystrybucyjny „cogsci"). Jeśli chcemy spróbować, opis instalacji jest na `https://anaconda.org/cogsci/pygame`. Posługiwanie się aplikacją `conda` użytą do instalacji jest opisane w dokumentacji[5]. My jednak wykorzystamy instalację za pomocą standardowego menadżera pakietów.

Uruchommy terminal w Jupyterze i wpiszmy komendę

```
python -m pip install -U pygame --user
```

Linia ta uruchamia moduł `pip` zainstalowany w Pythonie jako skrypt i przekazuje opcje `install`, która ściągnie z sieci bibliotekę i ją zainstaluje. Opcja `-U` powoduje, że jeśli już mamy bibliotekę, zostanie zaktualizowana do najnowszej wersji. Następnie pojawia się nazwa biblioteki. Opcja `--user` spowoduje, że biblioteka zostanie zainstalowana tylko dla obecnego użytkownika, zatem nie będzie wymagać uprawnień administratora. Po wciśnięciu klawisza Enter zobaczymy pasek postępu i po zakończeniu instalacji, możemy sprawdzić, czy wszystko działa, uruchamiając jedno z dołączonych do biblioteki PyGame aplikacji demo

[5] `https://docs.conda.io/projects/conda/en/latest/user-guide/getting-started.html`

```
python -m pygame.examples.aliens
```

Jeśli pojawiło się nowe okno a w nim gra, instalacja przebiegła pomyślnie.