O'REILLY®

Wydanie VI

Python

Wprowadzenie

Helion

Mark Lutz

Tytuł oryginału: Learning Python: Powerful Object-Oriented Programming, 6th Edition

Tłumaczenie: Anna Mizerska z wykorzystaniem fragmentów poprzedniego wydania w przekładzie Grzegorza Kowalczyka, Andrzeja Watraka, Anny Trojan i Marka Pętlickiego

ISBN: 978-83-289-2942-5

Authorized Polish translation of the English edition of *Learning Python, 6th Edition*
ISBN 9781098171308 © 2025 Mark Lutz

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/pytho6
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 24.

Pakiety modułów

Dotychczas kiedy importowaliśmy moduły, ładowaliśmy *pliki*. To typowy model użycia modułów i technika, której w początkach naszej kariery programisty Pythona będziemy używać najczęściej. Importowanie modułów to jednak coś więcej, niż dotychczas sugerowałem. Ten rozdział wprowadza **pakiety** modułów — zestaw plików modułów, które zwykle odpowiadają *folderom* (inaczej *katalogom*) na urządzeniu. W tym rozdziale zostały omówione cztery tematy:

- importy pakietów, które podają część ścieżki do pliku w folderze;
- same pakiety, które organizują moduły w folderach;
- względne importy pakietów, które używają kropek w obrębie pakietu, aby ograniczyć wyszukiwanie;
- pakiety przestrzeni nazw, które tworzą pakiet mogący obejmować wiele folderów.

Jak się przekonasz importowanie pakietów zamienia katalog z naszego komputera na kolejną przestrzeń nazw Pythona, z atrybutami odpowiadającymi podkatalogom oraz plikom modułów znajdujących się w tym katalogu. Jak zobaczymy, importowanie pakietów jest czasami wymagane do rozwiązania problemów z operacjami importowania powstających, kiedy na jednym komputerze zainstalowanych jest kilka plików programów o tej samej nazwie.

Pakiety to temat nieco zaawansowany, zatem wielu czytelników może go odłożyć na później, aż zdobędzie doświadczenie z modułami opartymi na plikach. Pakiety oferują łatwy sposób organizacji plików kodu, który unika konfliktów związanych z tą samą nazwą i jest wykorzystywany przez wiele narzędzi z biblioteki standardowej i narzędzi zewnętrznych, z których będziesz korzystać. Chociaż pakiety, podobnie jak wiele innych rzeczy w Pythonie, stały się z czasem złożone, warto poznać podstawy ich zastosowania.

Stosowanie pakietów

Aby załadować elementy w folderze pakietu, należy użyć zwykłych instrukcji importu i narzędzi, które już znasz, ale z podaniem ścieżki nazw, która odzwierciedla ścieżkę zagnieżdżonych folderów. Zacznijmy od oprowadzenia po pakietach.

Podstawy importowania pakietów

Na poziomie podstawowym importowanie pakietów jest całkiem proste — w miejscu, w którym w instrukcji `import` normalnie wstawiamy nazwę pliku, umieszczamy *ścieżkę* nazw rozdzielonych od siebie kropkami. Na przykład to zadziała w instrukcji `import`:

```
import dir1.dir2.mod
```

Tak samo wygląda to w przypadku instrukcji `from`:

```
from dir1.dir2.mod import var
```

I to samo dotyczy już zaimportowanych elementów w wywołaniach funkcji `reload`:

```
from importlib import reload
reload(dir1.dir2.mod)
```

Ścieżka z kropkami w tych instrukcjach ma odpowiadać ścieżce w *systemie plików* na Twoim urządzeniu, prowadzącej do pliku *mod.py* lub innego elementu z nazwą *mod* (jak już wiesz, może to być na przykład plik z kodem bajtowym lub moduł rozszerzający napisany w C). Powyższe instrukcje wskazują zatem, że na naszym komputerze istnieje katalog *dir1*, a w nim podkatalog *dir2* zawierający plik modułu *mod.py* (lub podobny).

Co więcej, takie operacje importowania sugerują, że katalog *dir1* znajduje się w katalogu nadrzędnym *dir0*, dostępnym dla ścieżki wyszukiwania modułów Pythona. Innymi słowy, powyższe instrukcje sugerują, że w systemie plików istnieje struktura przypominająca poniższą (z separatorami w postaci lewych ukośników stosowanych w systemie Windows).

```
dir0/dir1/dir2/mod.py                  # Lub mod.pyc, mod.so i tak dalej
dir0\dir1\dir2\mod.py                  # To samo w systemie Windows
```

Katalog nadrzędny *dir0* musi być dodany do ścieżki wyszukiwania modułów (o ile nie jest katalogiem głównym dla pliku najwyższego poziomu) — dokładnie tak samo, jakby *dir1* był plikiem modułu.

Mówiąc bardziej formalnie, pierwszy od lewej element ścieżki importu jest nazwą *względną* w ramach ścieżki wyszukiwania `sys.path`, którą mieliśmy okazję poznać w rozdziale 22. Od tego miejsca do końca ścieżki instrukcje `import` z naszego skryptu w jawny sposób określają ścieżki katalogów prowadzących do modułów.

Składnia z kropkami używana w pakietach jest niezależna od platformy, ale jest to również dowód na to, że ścieżki folderów w instrukcjach importu stają się obiektami zagnieżdżonymi: `dir1.dir2.mod` przechodzi przez trzy obiekty modułów po imporcie. Ta składnia wyjaśnia też, dlaczego Python zgłasza błąd dotyczący pliku, który nie jest pakietem, jeśli w instrukcjach importu podasz rozszerzenie *.py* — to jest traktowane jako import pakietu!

Pakiety a ustawienia ścieżki wyszukiwania

Jeżeli korzystamy z tej opcji, należy pamiętać, że ścieżki katalogów w instrukcjach `import` mogą być tylko *zmiennymi* rozdzielonymi *kropkami*. Nie można w tych instrukcjach użyć żadnej składni ścieżek specyficznej dla określonej platformy, takiej jak `C:\dir1`, `/Users/ja/dir1` czy

`../dir1` — takie ścieżki nie będą działały. Zamiast tego składni specyficznej dla platformy należy użyć w ustawieniach ścieżki wyszukiwania modułów i określić w ten sposób nazwę katalogu nadrzędnego *zawierającego* Twoje pakiety.

W poprzednim przykładzie katalog *dir0* — nazwa katalogu dodawana do ścieżki wyszukiwania modułów — może być dowolnie długą, specyficzną dla platformy ścieżką do katalogu, prowadzącą do katalogu *dir1*. Zamiast używać niepoprawnej instrukcji, takiej jak poniższa:

```
import C:\Users\ja\kody\dir1\dir2\mod          # Błąd — niepoprawna składnia (Windows)
import /Users/ja/kody/dir1/dir2/mod           # To samo, w systemie Unix
```

dodaj katalog *C:\Users\ja\kody* lub */Users/ja/kody* do zmiennej środowiskowej `PYTHONPATH`, zapisz w pliku *.pth* umieszczonym w strategicznym miejscu lub dodaj ręcznie do ścieżki `sys.path` w samym kodzie, a następnie wpisz w kodzie programu następujące polecenie:

```
import dir1.dir2.mod                          # OK, zmienne i kropki
```

W rezultacie wpisy ze ścieżki wyszukiwania modułów będą zawierały *prefiksy* katalogów specyficzne dla platformy i prowadzące do nazw znajdujących się po lewej stronie instrukcji `import` lub `from`. Takie instrukcje importu same z siebie dostarczają pozostałą część ścieżki katalogu w sposób neutralny dla platformy.

Jeżeli chodzi o proste importowanie plików, nie musisz dodawać katalogu *nadrzędnego* do ścieżki wyszukiwania modułów, jeżeli już tam jest. Zgodnie z tym, co pokazywaliśmy w rozdziale 22., będzie to katalog główny (katalog, w którym pracujesz interaktywnie, lub katalog, w którym znajduje się plik najwyższego poziomu uruchomionego programu), wraz ze standardowym katalogiem biblioteki lub katalogiem *site-packages* instalacji *pakietów* zewnętrznych. Tak czy inaczej, ścieżka wyszukiwania modułów musi zawierać wszystkie katalogi znajdujące się *z lewej strony* argumentu instrukcji importowania pakietu kodu.

Tworzenie pakietów

Aby utworzyć własne pakiety, musisz umieścić pliki modułów i zagnieżdżone foldery w folderze pakietu. Folder pakietu może, ale nie musi, zawierać plik *__init__.py*, który jest uruchamiany przy pierwszym imporcie, oraz plik *__main__.py*, który jest uruchamiany, gdy uruchamiany jest cały folder pakietu. W kolejnych punktach zobaczysz krok po kroku, jak to wygląda w kodzie.

Podstawowa struktura pakietów

W najprostszej formie pakiety to foldery zawierające zwykłe pliki modułów, a być może także zagnieżdżone podfoldery tego samego typu. Utwórzmy jeden jako przykład. Poniżej zostało użyte wcięcie, aby przedstawić zagnieżdżenie folderów, które będziemy wykorzystywać w tym i następnych punktach:

```
dir0/                                 # Katalog w ścieżce wyszukiwania modułów
     dir1/                            # Katalog główny pakietu dir1
          mod.py                      # Plik modułu dir1.mod w pakiecie
```

```
dir2/                          # Zagnieżdżony folder pakietu dir1.dir2
    mod.py                     # Moduł dir1.dir2.mod w pakiecie
```

Te zagnieżdżone foldery można utworzyć w eksploratorach plików lub za pomocą przedstawionych niżej poleceń, działających na większości platform. W systemie Windows zastosuj lewe ukośniki lub użyj znajdującego się wśród materiałów do pobrania dla tego rozdziału folderu, który ma już zbudowaną strukturę:

```
$ mkdir dir1
$ mkdir dir1/dir2                  # W systemie Windows użyj lewych ukośników
```

Należy pamiętać, że nazwy folderów pakietów, podobnie jak nazwy prostych plików modułów, muszą spełniać zasady dotyczące nazw zmiennych, ponieważ po zaimportowaniu stają się zmiennymi. Zajrzyj do rozdziału 11., aby przypomnieć sobie te ograniczenia. W skrócie: używaj liter, cyfr i znaków podkreślenia, a unikaj słów zarezerwowanych.

Teraz dodaj zagnieżdżone pliki modułów wymienione w przykładach 24.1 i 24.2. Ich kod najwyższego poziomu uruchamia się przy pierwszym imporcie i tworzy atrybuty jak zwykle, a ich tytuły wskazują ich ścieżki (dla spójności podano tutaj separatory uniksowe w postaci ukośników).

Przykład 24.1. *dir1/mod.py*

```
var = 'hakować'
print('Wczytywanie dir1.mod')
```

Przykład 24.2. *dir1/dir2/mod.py*

```
var = 'kod'
print('Wczytywanie dir1.dir2.mod')
```

Użycie podstawowych pakietów

Aby użyć naszego pakietu modułów, zaimportuj jego moduły z poziomu REPL lub innego pliku tak, jakbyś to robił dla prostego pliku *.py*, ale użyj kropek, aby w ścieżce wyszukiwania określić ścieżkę poniżej folderu *dir0* — co w REPL oznacza bieżący katalog:

```
$ python3
>>> import dir1.mod              # Ścieżka pakietu w imporcie
Wczytywanie dir1.mod
>>> dir1.mod.var                 # Uruchomienie kodu modułu
'hakować'

>>> import dir1.dir2.mod         # Jeszcze bardziej zagnieżdżona ścieżka
Wczytywanie dir1.dir2.mod
>>> dir1.dir2.mod.var            # Powtórzenie ścieżki, by uzyskać element na końcu
'kod'
```

Podobnie jak w przypadku prostych modułów najwyższego poziomu, kod modułów zagnieżdżonych w pakiecie jest uruchamiany tylko przy pierwszym imporcie:

```
>>> import dir1.mod              # Nic się nie dzieje, jeśli moduł został już zaimportowany
>>> import dir1.dir2.mod
```

Ogólnie rzecz biorąc, musisz *zaimportować* zagnieżdżony moduł, aby móc używać jego atrybutów, ponieważ moduły odpowiadające folderom nie odwołują się ponownie do systemu plików podczas pobierania atrybutów i zaimportowanie tylko folderu głównego nie wystarczy:

```
$ python3
>>> import dir1                          # Pobiera dir1, ale nie moduły znajdujące się w środku
>>> dir1.dir2.mod.var
AttributeError: module 'dir1' has no attribute 'dir2'

$ python3
>>> import dir1.dir2
>>> dir1.dir2.mod.var
AttributeError: module 'dir1.dir2' has no attribute 'mod'

$ python3
>>> import dir1.dir2.mod                 # Podanie pełnej ścieżki do celu
Wczytywanie dir1.dir2.mod
>>> dir1.dir2.mod.var
'kod'
```

Wszystko to działa tak samo w instrukcji `from`, która, jak widzieliśmy, jest w rzeczywistości importem z dodatkowym kopiowaniem nazw. Jednak wciąż musimy podać folder wraz ze ścieżką w instrukcji importu, aby móc uzyskać dostęp do jego zawartości. Ścieżki wymienione w importach są traktowane dosłownie, a nie pobierane z zmiennych:

```
$ python3
>>> from dir1.mod import var          #  Ścieżki pakietów w instrukcji from
Wczytywanie dir1.mod
>>> var                               # Nie powtarzaj ścieżki do elementu
'hakować'

>>> from dir1.dir2.mod import var
Wczytywanie dir1.dir2.mod
>>> var
'kod'

>>> from dir1 import dir2             # Ścieżki odczytywane dosłownie
>>> from dir2 import mod              # A nie z wartości zmiennych
ModuleNotFoundError: No module named 'dir2'
```

Jedną z potencjalnych zalet użycia instrukcji `from` jest to, że nie trzeba *powtarzać* ścieżki pakietu za każdym razem, gdy chcemy użyć elementu z tego pakietu. W przypadku instrukcji `import` zazwyczaj musisz za każdym razem powtórzyć pełną ścieżkę. Trzeba pamiętać, że `from` pozwala na zapisanie ścieżki tylko raz, a pobranej z niej prostej nazwy można używać wszędzie. Jest to szczególnie przydatne, jeśli struktura katalogu pakietu się zmienia (jak to często bywa w oprogramowaniu):

```
>>> import dir1.dir2.mod                      # import wymaga ścieżek
>>> dir1.dir2.mod.var
'kod'

>>> from dir1.dir2.mod import var             # from może skracać ścieżki
>>> var
'kod'
>>> from dir1.dir2 import mod                 # …i nie trzeba ich powtarzać
```

```
>>> mod.var
'kod'

>>> import dir1.dir2.mod as mod                    # Słowo kluczowe "as" też to potrafi
>>> mod.var
'kod'
```

Jak pokazuje ostatni przykład, rozszerzenie as wprowadzone w poprzednim rozdziale może skrócić ścieżki w taki sam sposób jak `from` — podobnie jak as upraszcza ręczne przypisanie (więcej o as w następnym rozdziale).

Pliki __init__.py

Jeśli podczas importowania folderu pakietu musisz uruchomić kod inicjujący, umieść go w pliku o nazwie *__init__.py* i zapisz w folderze pakietu. Python automatycznie uruchomi kod z tego pliku przy pierwszym imporcie pakietu w trakcie działania programu (lub sesji REPL). Oto jak ten plik wzbogaca strukturę naszego pakietu:

```
dir0/
      dir1/
            __init__.py                  # Uruchamia się przy pierwszym imporcie dir1
            mod__.py
           dir2/
                 __init__.py             # Uruchamia się przy pierwszym imporcie dir1.dir2
                 mod.py
```

Nowe pliki *__init__.py* znajdziesz w przykładach 24.3 i 24.4.

Przykład 24.3. *dir1/__init__.py*

```
var = 'Python'
print('Uruchomiono dir1.__init__.py')
```

Przykład 24.4. *dir1/dir2/__init__.py*

```
var = 3.12
print('Uruchomiono dir1.dir2.__init__.py')
```

Użycie zaktualizowanego pakietu

Te szczególne pliki *__init__.py* są opcjonalne. Ponieważ zapisany w nich kod jest uruchamiany automatycznie przy pierwszym zaimportowaniu katalogu przez program, służą przede wszystkim jako punkty zaczepienia do uruchomienia inicjalizacji wymaganych przez pakiet (stąd skrócone nazwy). W rzeczywistości ich przypisania służą do inicjalizacji *przestrzeni nazw* odpowiadającej folderowi na Twoim urządzeniu — podobnie jak w przypadku plików tworzą one atrybuty obiektu modułu pakietu:

```
$ python3
>>> import dir1.dir2.mod                    # Uruchamia wszystkie pliki __init__.py
Uruchomiono dir1.__init__.py
Uruchomiono dir1.dir2.__init__.py
Wczytywanie dir1.dir2.mod
>>> dir1.var                                # Nazwa w przestrzeni nazw pliku __init__.py
```

```
'Python'
>>> dir1.dir2.var
3.12
>>> dir1.dir2.mod.var                    # Nazwa w przestrzeni nazw zagnieżdżonego pliku
mod.py
'kod'
```

Technicznie rzecz biorąc, pakiety z plikami *__init__.py* nazywane są pakietami „regularnymi", a pakiety bez nich — pakietami „przestrzeni nazw". Zanim dodaliśmy plik *__init__.py*, przykład był pakietem „przestrzeni nazw" zawierającym jeden folder, a dodanie *__init__.py* uczyniło go „regularnym". Szczegółami tego rozróżnienia zajmiemy się później, ale poza tym, że pakiety „regularne" mają pierwszeństwo podczas wyszukiwania modułów, różnica jest dziś głównie zjawiskiem historycznym i nie ma znaczenia w większości zastosowań. Oczywiście pliki *__init__.py* mogą robić więcej niż tylko wyświetlać komunikaty. Ich role zostały omówione dalej w tym rozdziale.

Jak wspomniano wcześniej, `reload` działa również ze ścieżkami pakietów, jeśli już zaimportowałeś odpowiednie ścieżki. Kontynuując poprzednią sesję REPL:

```
>>> from importlib import reload
>>> reload(dir1)
Uruchomiono dir1.__init__.py
<module 'dir1' from '/…/PW6W/r24/dir1/__init__.py'>

>>> reload(dir1.dir2)
Uruchomiono dir1.dir2.__init__.py
<module 'dir1.dir2' from '/…/PW6W/r24/dir1/dir2/__init__.py'>

>>> reload(dir1.dir2.mod)
Wczytywanie dir1.dir2.mod
<module 'dir1.dir2.mod' from '/…/ PW6W/r24/dir1/dir2/mod.py'>
```

Na podstawie informacji wyświetlonych na ekranie można zauważyć, że to wywołanie przeładowuje tylko *pojedynczy* moduł na końcu ścieżki. Aby to poprawić, w następnym rozdziale zakodujemy narzędzie do przeładowywania, które po podaniu głównego katalogu pakietu może przeładować wszystkie elementy na ścieżce pakietu, jeden po drugim (zobacz „Przykład: przechodnie przeładowywania modułów" w następnym rozdziale).

Pliki __main__.py

Na koniec, jeśli chcesz, aby użytkownicy pakietu mogli uruchomić go tak, jakby był programem lub skryptem, dodaj pliki *__main__.py* do każdego folderu, w którym chcesz obsługiwać ten tryb. Python automatycznie uruchomi te pliki za każdym razem, gdy folder, który je zawiera, zostanie uruchomiony jak program. Oto jak ten ostatni element wzbogaca strukturę naszego pakietu:

```
dir0/
       dir1/
       __main__.py                    # Uruchamiany zawsze, kiedy uruchamiany jest dir1
       __init__.py
       mod__.py
       dir2/
```

```
__main__.py                    # Uruchamiany zawsze, kiedy uruchamiany jest dir1.dir2
__init__.py
mod.py
```

Poniżej znajduje się zawartość dodanych właśnie plików *__main__.py* z przykładów 24.5 i 24.6. Są proste, abyśmy mogli skupić się na pakietach:

Przykład 24.5. *dir1/__main__.py*

```
print('Uruchomiono dir1.__main__.py')
```

Przykład 24.6. *dir1/dir2/__main__.py*

```
print('Uruchomiono dir1.dir2.__main__.py')
```

Użycie zaktualizowanego pakietu

Pliki *__main__.py*, znajdujące się w folderze pakietu, są uruchamiane automatycznie dla różnych opcji uruchamiania, w tym bezpośrednio z wiersza poleceń, które wskazują folder zawierający pakiet. W tym trybie pakiet nie musi znajdować się na ścieżce wyszukiwania modułów:

```
$ python3 dir1                          # Uruchamia __main__.py (nie __init__.py)
Uruchomiono dir1.__main__.py
$ python3 dir1/dir2
Uruchomiono dir1.dir2.__main__.py
$ python3 dir1/dir2/mod.py               # Uruchamia zagnieżdżony mod.py
Wczytywanie dir1.dir2.mod
```

Pliki *__main__.py* w pakietach są również uruchamiane w trybie `python -m`, który, jak już wiesz, lokalizuje element na ścieżce wyszukiwania modułów i uruchamia go jako skrypt najwyższego poziomu. W przeciwieństwie do bezpośredniego polecenia uruchamiania ten tryb uruchamia pełny mechanizm importu pakietów i wszystkie pliki *__init__.py* na ścieżce, ale pakiet musi się na niej znajdować.

```
$ python3 -m dir1                       # Uruchamia zarówno __init__.py, jak
                                        # i __main__.py
Uruchomiono dir1.__init__.py
Uruchomiono dir1.__main__.py

$ python3 -m dir1.dir2                  # I pakiet musi być na ścieżce wyszukiwania
Uruchomiono dir1.__init__.py
Uruchomiono dir1.dir2.__init__.py
Uruchomiono dir1.dir2.__main__.py

$ python3 -m dir1.dir2.mod              # Uruchamia mod.py i __init__.py modułu
                                        # nadrzędnego
Uruchomiono dir1.__init__.py
Uruchomiono dir1.dir2.__init__.py
Wczytywanie dir1.dir2.mod
```

Jest to podobne do „pakietów” aplikacji w systemie macOS i w smartfonach, które są w rzeczywistości folderami, lecz mogą być uruchamiane jako elementy wykonywalne. Pliki *__main__.py* pełnią różnorodne role, ale często są używane do zapewnienia interfejsu wiersza poleceń dla

pakietu narzędzi. Dzięki temu możesz zaimportować pakiet, aby używać jego narzędzi w innym programie, ale także uruchomić go jako całość, aby korzystać z narzędzi w samodzielnym programie. Pliku tego można też użyć do umieszczania kodu testowego dla pakietu.

Plik *__main__.py* jest również uruchamiany, gdy znajduje się w pliku ZIP na ścieżce wyszukiwania. Jak wspomniano wcześniej, pliki ZIP są traktowane jak zwykły folder podczas wyszukiwania modułu, ale więcej na ten temat znajdziesz w dokumentacji Pythona.

Trzeba wspomnieć, że importy tych samych pakietów w pliku *__main__.py* mogą wymagać użycia składni `from .` (której się nauczysz wkrótce), gdy będą uruchamiane tylko przez argument `-m`. Ta składnia zawodzi w przypadku uruchomień bezpośrednich w wierszu poleceń, więc może być konieczne uruchomienie obsługujące przypadek, gdy *__main__.py* wymaga importów z tego samego pakietu. Jak się przekonasz, podanie pełnej ścieżki podczas importu dla kodu, który ma służyć zarówno jako pakiet, jak i program, może rozwiązać ten problem.

Do czego służą pakiety?

Teraz, gdy zobaczyłeś, jak stosować i tworzyć pakiety, możesz się zastanawiać, do czego właściwie są one potrzebne. Wbrew temu, co mogłeś słyszeć, pakiety są całkowicie opcjonalne i raczej są zbyt skomplikowane na początku Twojej przygody z kodowaniem, a nawet później możesz pisać niesamowite programy bez ich użycia.

Ogólnie rzecz biorąc, pakiety są przydatne, ponieważ pomagają zapewnić unikatowość nazw w większych programach i w bibliotekach publikowanych do użytku przez innych. Organizując kod jako folder pakietu, możesz być niemal pewny, że nazwy jego plików nie będą kolidować z nazwami innego oprogramowania na urządzeniach, na których są zapisane. Pełnią tę samą rolę, jeśli chodzi o separację przestrzeni nazw, jaką pełnią moduły oparte na plikach i lokalne zasięgi w funkcjach. Pakiety jednak opierają się na systemie plików: ponieważ odwołania do Twojego pakietu używają nazwy folderu pakietu, istnieje mniejsze ryzyko kolizji związanych z tą samą nazwą.

Ponadto pakiety mogą sprawić, że importy będą bardziej opisowe, ułatwią zadanie śledzenia lokalizacji zmiennych w plikach kodu i uproszczą ustawienia ścieżki wyszukiwania modułów. Jedyna sytuacja, w której importowanie pakietów jest *wymagane* do rozwiązania niejasności, pojawia się, kiedy na jednym komputerze zainstalowana jest większa liczba programów z plikami noszącymi te same nazwy. Jest to problem instalacyjny, jednak w praktyce może stać się dotkliwy — szczególnie biorąc pod uwagę tendencję programistów do używania prostych i podobnych nazw plików modułów.

Aby pomóc Ci zrozumieć, dlaczego to wszystko ma znaczenie, zajmiemy się pewnym hipotetycznym scenariuszem wydarzeń.

Historia dwóch systemów

Załóżmy, że programista tworzy w Pythonie program zawierający plik o nazwie *utilities.py* (ze wspólnym kodem narzędziowym), a także plik najwyższego poziomu *main.py* wykorzystywany przez użytkowników do uruchomienia programu. W całym programie pliki wykorzystują

instrukcję `import utilities` do załadowania wspólnego kodu narzędzi. Kiedy program jest dostarczany klientom, stanowi jedno archiwum *.tar* czy *.zip* zawierające wszystkie pliki programu, a po instalacji rozpakowuje wszystkie pliki do jednego katalogu na komputerze docelowym o nazwie *system1*.

```
system1\
   utilities.py                    # Wspólne funkcje i klasy narzędzi
   main.py                         # Uruchamia program
   other.py                        # Importuje utilities w celu załadowania narzędzi
```

Załóżmy teraz, że drugi programista tworzy inny program z plikami o nazwach *utilities.py* oraz *main.py* i ponownie wykorzystuje instrukcję `import utilities` w całym programie w celu załadowania pliku ze wspólnym kodem. Kiedy drugi system zostanie pobrany i zainstalowany na tym samym komputerze co pierwszy, jego pliki zostaną rozpakowane do nowego katalogu o nazwie *system2* na komputerze klienta, tak by nie nadpisały plików o tych samych nazwach z pierwszego systemu.

```
system2\
   utilities.py                    # Wspólne narzędzia
   main.py                         # Uruchamia program
   other.py                        # Importuje narzędzia
```

Jak na razie nie ma żadnych problemów — oba systemy mogą współistnieć i mogą być wykonywane na tym samym komputerze. Tak naprawdę nie musimy nawet konfigurować ścieżki wyszukiwania modułów, by skorzystać z obu programów — ponieważ Python zawsze najpierw przeszukuje katalog główny (czyli katalog zawierający plik najwyższego poziomu), operacje importowania w plikach obu systemów automatycznie zobaczą wszystkie pliki w katalogu danego systemu. Jeżeli na przykład klikniemy plik *system1\main.py*, wszystkie operacje importowania najpierw będą przeszukiwały katalog *system1*. W podobny sposób po uruchomieniu pliku *system2\main.py* jako pierwszy przeszukany zostanie katalog *system2*. Należy pamiętać, że ustawienia ścieżki wyszukiwania modułów są potrzebne tylko wtedy, gdy importujemy pliki pomiędzy katalogami.

Załóżmy jednak, że po zainstalowaniu tych dwóch programów na komputerze decydujemy się użyć jakiejś części kodu z każdego z plików *utilities.py* we własnym programie. W końcu jest to wspólny kod narzędzi, a kod napisany w Pythonie z natury służy do ponownego użycia. W takim przypadku możesz użyć poniższych instrukcji w kodzie pliku utworzonego w trzecim katalogu, tak by załadować jeden z dwóch plików z narzędziami.

```
import utilities
utilities.func('hakować')
```

I teraz zaczynamy dostrzegać problem. Aby taki kod działał, musimy ustawić ścieżkę wyszukiwania modułów w taki sposób, by obejmowała ona katalogi zawierające pliki *utilities.py*. Który katalog należy jednak umieścić jako pierwszy — *system1* czy *system2*?

Problemem jest *liniowa* natura ścieżki wyszukiwania, która zawsze jest przeglądana od lewej do prawej strony, więc bez względu na to, jak długo byśmy się nad tym zastanawiali, zawsze otrzymamy plik *utilities.py* z katalogu wymienionego jako pierwszy (bardziej na lewo) w ścieżce wyszukiwania. W takiej postaci nigdy nie będziemy w stanie zaimportować tego pliku z innego katalogu.

Możemy spróbować zmodyfikować `sys.path` w skrypcie przed każdą operacją importowania, ale to dodatkowa operacja, na dodatek bardzo podatna na błędy, a zmiana ustawień zmiennej `PYTHONPATH` przed każdym uruchomieniem programu w Pythonie jest zbyt nużąca i nie pozwala na używanie *obu* wersji w jednym pliku. Domyślne rozwiązanie sprawia, że jesteśmy w kropce.

Problem ten może rozwiązać właśnie importowanie pakietów. Zamiast instalować programy w niezależnych katalogach wymienionych na ścieżce wyszukiwania modułów osobno, możesz spakować je i zainstalować w *podkatalogach* we wspólnym katalogu głównym. Możemy na przykład zorganizować cały kod tego przykładu w postaci hierarchii katalogów, wyglądającej następująco:

```
root\
     system1\
             __init__.py
             utilities.py
             main.py
             other.py
     system2\
             __init__.py
             utilities.py
             main.py
             other.py
     mycode\                    # Tutaj lub w dowolnym innym miejscu
             myfile.py          # Tutaj nasz nowy kod
```

W takiej sytuacji do ścieżki wyszukiwania dodajemy tylko wspólny *katalog główny*. Jeżeli wszystkie operacje importowania w naszym kodzie są wykonywane względem wspólnego katalogu głównego, możemy zaimportować plik narzędzi z *dowolnego* programu za pomocą importowania pakietu — nazwa katalogu zawierającego plik sprawia, że ścieżka (i tym samym referencja do modułu) staje się unikalna. Tak naprawdę możemy nawet zaimportować *oba* pliki narzędzi w tym samym module, dopóki wykorzystujemy instrukcję `import` i powtarzamy pełną ścieżkę za każdym razem, gdy odwołujemy się do modułów narzędzi.

```
import system1.utilities                    # Importowanie z jednego pakietu
import system2.utilities                    # Importowanie z drugiego pakietu

system1.utilities.function('hakować')       # I użyj nazwy z dowolnego z nich
system2.utilities.function('kod')           # Lub obu!
```

Nazwa modułu zawierającego plik sprawia, że referencja do modułu staje się unikalna.

Warto zauważyć, że w przypadku importowania pakietów musimy użyć instrukcji `import` zamiast `from` tylko wtedy, gdy musimy uzyskać dostęp *do tego samego* atrybutu z dwóch lub większej liczby ścieżek. Gdyby nazwa wywoływanej funkcji była w każdej ścieżce inna, można by było użyć instrukcji `from`, co pozwalałoby uniknąć powtarzania pełnej ścieżki za każdym wywołaniem którejś z funkcji, tak jak opisano to wcześniej; do utworzenia unikalnych synonimów nazw możemy również użyć rozszerzenia `as`.

Z technicznego punktu widzenia w tym przypadku podkatalog *mycode* nie musi się znajdować w *katalogu głównym* — dotyczy to jedynie pakietów kodu, z których będziemy importować. Ponieważ jednak nigdy nie wiemy, kiedy nasze własne moduły będą mogły się przydać

innym programom, możemy równie dobrze od razu umieścić je we wspólnym katalogu głównym w celu uniknięcia problemów z konfliktami między takimi samymi nazwami w przyszłości.

Co ważne, jeżeli będziemy konsekwentnie rozpakowywać wszystkie programy napisane w Pythonie w jednym katalogu głównym, tak jak zaprezentowano to powyżej, konfiguracja ścieżki staje się banalnie prosta — wystarczy do niej tylko raz dodać wspólny katalog główny. Ponadto warto zauważyć, że instrukcje importowania z obu oryginalnych systemów działają bez zmian. Ponieważ najpierw przeszukiwane są ich *katalogi domowe*, dodanie wspólnego katalogu do ścieżki wyszukiwania nie ma znaczenia dla kodu z katalogów *system1* oraz *system2*. Nadal można w nich zastosować polecenie `import utilities` i oczekiwać, że odnajdą w ten sposób własne pliki. Jak zobaczysz dalej, jeśli będziesz używać tych plików *również* jako pakietów, możliwe, że będziesz musiał zastosować importy względne w postaci `from .` (a plik *main.py* mógłby mieć nazwę *__main__.py*).

Na koniec pamiętaj, że nawet jeśli nigdy nie utworzysz własnego pakietu, prawdopodobnie będziesz korzystać z narzędzi z biblioteki standardowej i narzędzi zewnętrznych, które to robią. Oto przykładowe pakiety z biblioteki standardowej:

```
from email.message import Message                  # Analiza tekstu e-maila/ tworzenie e-maila
from tkinter.filedialog import askopenfilename     # Przenośny zestaw narzędzi do tworzenia GUI
from http.server import CGIHTTPRequestHandler      # Narzędzia dla serwera (do Python 3.15?)
```

Poprzez umieszczenie swojego kodu w pakiecie takie narzędzia stają się bardziej samodzielne i unikają konfliktów nazw. To, czy zdecydujesz się zrobić to samo ze swoim kodem, zależy od Ciebie, ale w kolejnych punktach będziemy się zagłębiać jeszcze bardziej w ten temat, na wypadek gdybyś się kiedyś zdecydował na tworzenie własnych pakietów.

Role pliku inicjalizacji pakietu

Teraz, gdy znasz już podstawy i sens korzystania z pakietów, przyjrzyjmy się szczegółom związanym z ich użyciem. Pliki *__init__.py* w pierwszych przykładach były proste, ale te pliki mogą zawierać dowolny kod Pythona, tak jak normalne pliki modułów. Ich nazwy są specjalne, ponieważ ich kod jest uruchamiany automatycznie przy pierwszym imporcie katalogu przez program w Pythonie, co służy głównie jako punkt zaczepienia do wykonywania kroków inicjalizacyjnych wymaganych przez pakiet. Jednak te pliki mogą być również całkowicie puste i czasami mają dodatkowe role.

Pliki *__init__.py* służą jako punkty zaczepienia dla działań odbywających się w czasie inicjalizowania pakietów, deklarują katalogi jako pakiety Pythona, generują przestrzenie nazw dla katalogów i implementują zachowanie instrukcji `from *`, kiedy wykorzystuje się je w połączeniu z importowaniem pakietów.

Inicjalizacja pakietów

Za pierwszym razem, gdy Python importuje coś za pomocą katalogu, automatycznie wykonuje cały kod z pliku *__init__.py* tego katalogu. Z tego powodu pliki te są naturalnym miejscem do wstawienia kodu inicjalizującego stan wymagany przez pakiet, który może

na przykład wykorzystać plik inicjalizujący do utworzenia wymaganych plików z danymi czy otwarcia połączenia z bazą danych. Zazwyczaj pliki *__init__.py* nie są przydatne, jeżeli zostaną wykonane bezpośrednio (do tego służy plik *__main__.py*); są one uruchamiane automatycznie przy pierwszym dostępie do pakietu.

Inicjalizacja przestrzeni nazw modułu

Jak już wspomniano wcześniej, w modelu importowania pakietów ścieżki katalogów ze skryptu stają się po zaimportowaniu prawdziwymi ścieżkami zagnieżdżonych obiektów. Jak widać w poprzednim przykładzie, po zaimportowaniu wyrażenie `dir1.dir2.mod` działa i zwraca obiekt modułu, którego przestrzeń nazw zawiera wszystkie zmienne przypisane przez plik *__init__.py* katalogu *dir2*. Takie pliki udostępniają przestrzeń nazw dla obiektów modułów tworzonych dla katalogów, które w przeciwnym razie nie miałyby żadnego skojarzonego pliku modułu.

Deklaracje użyteczności modułu

Jednym z zadań plików *__init__.py* jest również zadeklarowanie, że dany katalog jest pakietem Pythona. Obecność tych plików zapobiega niezamierzonemu ukrywaniu prawdziwych modułów przez podobne nazwy katalogów, które pojawiają się w ścieżce wyszukiwania modułów. Bez tego zabezpieczenia Python mógłby wybrać katalog, który nie ma nic wspólnego z Twoim kodem, tylko dlatego, że pojawia się wcześniej w ścieżce wyszukiwania. Jak zobaczymy później, pakiety przestrzeni nazw w znacznym stopniu redukują tę rolę, ale uzyskują podobny efekt w sposób algorytmiczny, skanując ścieżkę w poszukiwaniu kolejnych plików. Pliki *__init__.py* w pakietach nadal nadają pakietowi wyższy priorytet niż folderowi o tej samej nazwie znajdującemu się w innym miejscu na ścieżce wyszukiwania.

Zachowanie instrukcji `from *`

Jako zaawansowaną opcję możemy wykorzystać listy `__all__` z plików *__init__.py* do zdefiniowania, co jest eksportowane, kiedy katalog importowany jest za pomocą instrukcji `from *`. W pliku *__init__.py* lista `__all__` ma być listą nazw podmodułów, które powinny zostać zaimportowane, kiedy instrukcji `from *` użyjemy na nazwie pakietu (katalogu). Jeżeli lista `__all__` nie zostanie zdefiniowana, instrukcja `from *` nie załaduje automatycznie podmodułów zagnieżdżonych w katalogu. Zamiast tego załaduje tylko zmienne zdefiniowane przez przypisania w pliku *__init__.py* katalogu, w tym wszystkie podmoduły w jawny sposób zaimportowane przez kod tego pliku. Na przykład użycie instrukcji `from submodule import X` w pliku *__init__.py* katalogu sprawia, że zmienna `X` z podmodułu `submodule` będzie dostępna w przestrzeni nazw tego katalogu bez `__all__`.

Później zobaczysz przykład z listą `__all__`, a więcej informacji znajdziesz w rozdziale 25., gdzie między innymi dowiesz się, że służy ona również do deklarowania eksportów `from *` z prostych plików, nie tylko pakietów, i że jest to część szerszego zagadnienia — **ukrywania danych**. Pliki *__init__.py* mogą również pozostać puste. Dzięki temu podczas wyszukiwania importu Twój pakiet będzie miał pierwszeństwo przed pakietami przestrzeni nazw, ale aby zrozumieć, dlaczego to ma znaczenie, musimy przejść dalej.

Uważaj przy scalaniu klas. Nie powinieneś mylić plików *__init__.py* w pakietach z metodami konstruktora klasy `__init__`, które poznamy w następnej części książki. Pierwsze z nich to pliki z kodem uruchamiane, gdy importy po raz pierwszy przechodzą przez folder pakietu podczas działania programu, a drugie są metodami wywoływanymi w celu utworzenia instancji klasy. Oba komponenty mają role inicjujące i nie są obowiązkowe, ale znacznie różnią się od siebie pomimo podobnych nazw.

Względne importowanie pakietów

Dotychczas podczas omawiania zagadnień związanych z importowaniem pakietów koncentrowaliśmy się głównie na importowaniu plików *spoza* pakietu. Wewnątrz pakietu importowanie plików tego samego pakietu może korzystać z tej samej składni z pełnymi ścieżkami, co importowanie plików spoza pakietu — i jak zobaczymy, czasami nawet powinno. Pliki pakietów mogą jednak również korzystać ze specjalnych, uproszczonych reguł importowania *wewnątrz pakietu*, gdzie zamiast określania pełnej ścieżki do modułu w pakiecie, można zastosować ścieżkę *względną* wewnątrz pakietu.

Względne i bezwzględne importowanie pakietów

Kod jest prosty. Importy uruchamiane przez pliki używane jako część pakietu mogą korzystać ze specjalnej składni, takiej jak ta poniżej, która działa tylko w plikach używanych jako komponenty pakietu i tylko w instrukcjach `from`, a nie `import`:

```
from . import module              # Import modułu z tego pakietu (tylko)
from .module import name          # Import nazwy z modułu z tego pakietu
from .. import module             # Import modułu równorzędnego z folderu nadrzędnego
from ..module import name         # Import nazwy równorzędnego modułu względem nadrzędnego
                                  # modułu tej nazwy
```

Ta składnia nie miałaby sensu w instrukcji `import`, ponieważ ta instrukcja przypisuje moduły do prostych nazw, a nie ścieżek. Jednak w instrukcji `from`, gdy źródło importu zaczyna się od kropek (lub jest tylko kropką), to import jest uznawany za *względny* — identyfikuje element względnie wobec folderu samego pakietu, w którym ten element jest zawarty.

Ta nazwa wywodzi się od względnych ścieżek do *plików*, które odnoszą się do pliku względem bieżącego katalogu roboczego, co omówiono w rozdziale 9. Podobnie jak w przypadku nazw plików kropka oznacza bezpośrednio otaczający folder pakietu, a dwie kropki oznaczają folder o poziom wyżej (każda dodatkowa kropka oznacza kolejny poziom wyżej, choć jest to rzadko używane). Tutaj jednak odwołujemy się do elementu względnie wobec pakietu importującego, a nie do pliku z zawartością względnie wobec bieżącego katalogu roboczego.

Co ważne, względne importowanie w obrębie pakietu przeszukuje *tylko* wskazany pakiet. Foldery wymienione w `sys.path` nigdy nie są sprawdzane, jak podczas zwykłego importowania (wstępne sprawdzenie modułów wbudowanych i zamrożonych jest pomijane). Ponadto składnia importu względnego może być używana *tylko* wtedy, gdy plik jest wykorzystywany jako

część pakietu, w przeciwnym razie nie zadziała. Ma to zarówno dobre, jak i złe konsekwencje, które za chwilę omówimy.

Importowanie w plikach wykorzystywanych jako część pakietu może być jednak kodowane bez kropek jak zwykle, zarówno w instrukcji `import`, jak i `from`:

```
import module                   # Import modułu z folderu zawartego w liście sys.path
from module import name         # Import nazwy z tego samego modułu
```

Bez wiodących kropek importowanie jest uznawane za **bezwzględne** (ang. *absolute*) — identyfikuje element znajdujący się w folderze wymienionym w `sys.path`. Jest to normalne zachowanie importów w Pythonie i w rzeczywistości nie ma nic „bezwzględnego" w sensie analogii do ścieżek plików (bezwzględne ścieżki plików to pełne ścieżki w systemie plików). Te „bezwzględne" importy są tak naprawdę **względne** (ang. *relative*) wobec wpisu w ścieżce wyszukiwania modułów, ale takie określenie się już przyjęło.

Należy również podkreślić, że importowanie bezwzględne w obrębie pakietu nie sprawdza automatycznie samego pakietu — pomija pakiet i przechodzi bezpośrednio do przeszukiwania folderów z `sys.path`. Jak widzieliśmy, lista `sys.path` może obejmować folder pakietu dzięki bieżącemu katalogowi roboczemu sesji REPL lub folderowi domowemu skryptu najwyższego poziomu oraz ustawionej zmiennej `PYTHONPATH` lub innym ustawieniom. Jednak bezwzględne importowanie samo w sobie nie sprawdza pakietu dla importów uruchamianych w plikach pakietu.

Importowanie względne — za i przeciw

Dlaczego kropki? Krótko mówiąc, importy względne zapewniają, że importy pakietu będą ładować jego własne moduły. Ponieważ importy względne przeszukują *tylko* sam pakiet, nie załadują przez przypadek niepowiązanego modułu o tej samej nazwie, znajdującego się gdzieś indziej na maszynie hostującej, wymienionego w `sys.path`. To z kolei czyni pakiet bardziej samodzielnym. Bez importów względnych taki niepowiązany moduł mógłby zakłócić działanie kodu pakietu. Dzięki importowaniu względnemu pakiety są mniej zależne od ustawień ścieżki wyszukiwania klienta, których nie da się przewidzieć ani kontrolować.

Minusem jest to, że ten model jest propozycją typu „*wszystko albo nic*". W praktyce musisz zdecydować, jaką rolę mają pełnić Twoje pliki: pakietu czy programu. Importy względne zapewniają widoczność samego pakietu i muszą być używane jako pakiet, natomiast importy bezwzględne nie są ograniczone do pakietów, ale nie dają widoczności dla samego pakietu. Ta kombinacja wydaje się sytuacją bez wyjścia, która ogranicza użyteczność kodu.

Zatem jeśli tworzysz bibliotekę narzędzi, może to nie mieć znaczenia, gdyż możesz przyjąć importy względne w całym swoim pakiecie i dodać plik *__main__.py*, aby uruchomić pakiet jako program z przełącznikiem `-m`. Jeśli jednak piszesz bardziej tradycyjną kolekcję kodu, której chcesz używać dowolnie, opcje wydają się ograniczone.

Jednym z prostych rozwiązań tego dylematu jest całkowite *unikanie* importów względnych i używanie pełnych, jednoznacznych i „bezwzględnych" ścieżek importu pakietu w całym kodzie.

Oznacza to użycie importów, które podają ścieżkę do pożądanego elementu, począwszy od folderu głównego pakietu i z jego uwzględnieniem:

```
import package.module                    # Import modułu z tego pakietu
from package import module               # To samo co from . import module
form package.module import name          # To samo co from .module import name
```

Aby to zadziałało, folder główny pakietu musi znajdować się w folderze wymienionym w `sys.path`, ale to będzie *zwykły* przypadek i klienty nie będą miały możliwości zaimportowania kodu pakietu. To również nie wspiera egzotycznych importów bezpośrednich, takich z użyciem dwóch kropek dla katalogów nadrzędnych, ale wydają się one raczej rzadkie (obecnie tylko jedno narzędzie z biblioteki standardowej ich używa).

Dzięki temu rozwiązaniu, które pomija importy względne, foldery z kodem mogą być stosowane bardziej elastycznie, a zarówno bezpośrednie polecenia w wierszu poleceń, jak i przełącznik trybu modułu `-m` mogą być używane do uruchamiania plików w pakiecie jako programów. Następny punkt pokaże, jak to zrobić.

Importy względne w działaniu

Aby pokazać w praktyce teorię omówioną powyżej, wróćmy do prostego pakietu, który utworzyliśmy na początku tego rozdziału. Jak pamiętasz, jego plik *__main__.py* uruchamia się automatycznie, gdy folder jest uruchamiany jako pakiet, ale będzie również pełnił rolę dowolnego pliku najwyższego poziomu w folderze uruchamianym jako skrypt. Kiedy ostatnio widzieliśmy ten plik, zawierał tylko polecenie `print`. Oto przypomnienie, jak wyglądały pliki, których będziemy tutaj używać, abyś nie musiał się cofać:

```
# dir1/mod.py
var = 'hakować'
print('Wczytywanie dir1.mod')

# dir1/__main__.py
print('Uruchomiono dir1.__main__.py')
```

Wystarczy tej teorii, pokażemy zatem kilka prostych przykładów, aby zademonstrować w praktyce koncepcję importów względnych.

Zwykłe importowanie — rozgrzewka

Ten plik *__main__.py* wcześniej działa, ale sprawa się komplikuje, kiedy go uruchamiamy jako skrypt najwyższego poziomu, gdyż wtedy próbuje zaimportować inny moduł znajdujący się w tym samym folderze pakietu, jak w przykładzie 24.7.

Przykład 24.7. *dir1/__main__.py (zmieniony)*

```
import mod
print('Uruchomiono dir1.__main__.py:', dir1.mod.var.upper())
```

Napisany w ten sposób plik może być uruchamiany bezpośrednio z wiersza poleceń (zarówno jako folder, jak i skrypt), ale nie za pomocą przełącznika trybu modułu `-m`, który aktywuje mechanizmy pakietów (zwróć uwagę na dane wyjściowe z *__init__.py* w tym trybie):

```
$ python3 dir1                                  # Uruchomienie bezpośrednio z wiersza poleceń
Wczytywanie dir1.mod
Wczytywanie dir1.__main__.py: HAKOWAĆ
$ python3 dir1/__main__.py                      # Uruchomienie jako folder lub plik
Wczytywanie dir1.mod
Uruchomiono dir1.__main__.py: HAKOWAĆ

$ python3 -m dir1                               # Uruchomienie z przełącznikiem trybu modułu
Uruchomiono dir1.__init__.py
ModuleNotFoundError: No module named 'mod'
$ python3 -m dir1.__main__                      # Jako główny katalog pakietu lub moduł jawny
Uruchomiono dir1.__init__.py
ModuleNotFoundError: No module named 'mod'
```

Pierwsze dwa polecenia działają, ponieważ plik *__main__.py* jest uruchamiany jako normalny program spoza pakietu i znajduje importowanego „sąsiada”, gdyż w `sys.path` wpisano katalog główny, czyli folder pliku najwyższego poziomu. Problem z ostatnimi dwoma poleceniami polega na tym, że używają one pliku jako części pakietu. Instrukcja `import mod` w pliku *__main__.py* jest wtedy interpretowana jako importowanie *bezwzględne*, co powoduje *pominięcie* otaczającego pakietu. Stąd te błędy.

Importowanie względne — przygoda

Naszą pierwszą reakcją może być dodanie importów *względnych*, aby zadowolić system pakietów, jak w przykładzie 24.8.

Przykład 24.8. *dir1/__main__.py (zmieniony)*

```
from . import mod
print('Uruchomiono dir1.__main__.py:', mod.var.upper())
```

Ten kod naprawia działanie `-m`, ale *psuje* bezpośrednie polecenie w wierszu poleceń:

```
$ python3 -m dir1
Uruchomiono dir1.__init__.py
Wczytywanie dir1.mod
Uruchomiono dir1.__main__.py: HAKOWAĆ
$ python3 -m dir1.__main__
Uruchomiono dir1.__init__.py
Wczytywanie dir1.mod
Uruchomiono dir1.__main__.py: HAKOWAĆ

$ python3 dir1
ImportError: attempted relative import with no known parent package
$ python3 dir1/__main__.py
ImportError: attempted relative import with no known parent package
```

Pierwsze dwa polecenia działają, ponieważ flaga `-m` wywołuje zachowanie pakietu, co umożliwia użycie składni importu względnego (z kropką), która przeszukuje pakiet i znajduje docelowy moduł. Problem z ostatnimi dwoma poleceniami polega na tym, że Python nie pozwala na użycie importów względnych w trybie niebędącym pakietem, co z góry skazuje te uruchomienia na niepowodzenie, niezależnie od ustawień ścieżki wyszukiwania. Stąd te błędy.

Stąd wynika, że musimy *wybrać*, czy nasz kod ma pełnić rolę pakietu, czy nie.

Importowanie bezwzględne — rozwiązanie

Jak już wspomniano, łatwym sposobem na obejście tego ograniczenia jest użycie zwykłych importów pakietowych, które jawnie określają bezwzględne ścieżki importu (które, nawiasem mówiąc, są tak naprawdę względne wobec `sys.path`), jak w przykładzie 24.9.

Przykład 24.9. *dir1/__main__.py (zmieniony)*

```
import dir1.mod
print('Uruchomiono dir1.__main__.py:', dir1.mod.var.upper())
```

Jak również wspomniano wcześniej, katalog główny pakietu, `dir1`, zazwyczaj będzie na `sys.path`, ponieważ jest to jedyny sposób na korzystanie z jego kodu spoza pakietu (klienty muszą importować za pomocą ścieżek, które również zaczynają się od korzenia pakietu `dir1`).

Aby to tutaj pokazać, jawnie dodamy katalog główny pakietu, używając względnej ścieżki z kropką (w kategoriach *systemu plików*) dla bieżącego katalogu (w typowym użyciu zamiast tego może być ścieżka bezwzględna lub folder *site-packages* dla zainstalowanych pakietów). Musimy to ustawić tutaj, ponieważ folder pliku najwyższego poziomu na ścieżce wyszukiwania w trybie bezpośredniego polecenia znajduje się o jeden poziom *poniżej* folderu głównego pakietu, a zarówno tryb bezpośredni, jak i `-m` potrzebują dostępu do katalogu głównego pakietu:

```
$ export PYTHONPATH=.                    # Lub podobnie poza systemem Unix — zobacz rozdział 22.
$ python3 dir1
Uruchomiono dir1.__init__.py
Wczytywanie dir1.mod
Uruchomiono dir1.__main__.py: HAKOWAĆ
$ python3 dir1/__main__.py
Uruchomiono dir1.__init__.py
Wczytywanie dir1.mod
Uruchomiono dir1.__main__.py: HAKOWAĆ

$ python3 -m dir1                        # Oba tryby działają, importy bez kropki
Uruchomiono dir1.__init__.py
Wczytywanie dir1.mod
Uruchomiono dir1.__main__.py: HAKOWAĆ
$ python3 -m dir1.__main__
Uruchomiono dir1.__init__.py
Wczytywanie dir1.mod
Uruchomiono dir1.__main__.py: HAKOWAĆ
```

Teraz uruchomienia *zarówno* bezpośrednio w wierszu poleceń, jak i z przełącznikiem `-m` działają, ponieważ nie korzystamy z narzędzia, które ograniczałoby użyteczność kodu tylko do jednego trybu. W rezultacie mamy kod w *trybie dualnym* — może być używany zarówno jako program, jak i pakiet.

Jeśli wolisz nie powtarzać ścieżek importu przy każdym odniesieniu do elementu, oba tryby uruchamiania również działają, jeśli w pliku *__main__.py* umieścimy jedną z poniżej podanych wersji kodu:

```
import dir1.mod as mod
print('Uruchomiono dir1.__main__.py:', mod.var.upper())

from dir1 import mod
print('Uruchomiono dir1.__main__.py:', mod.var.upper())
```

```
from dir1.mod import var
print('Uruchomiono dir1.__main__.py:', var.upper())
```

Możesz również użyć `from *` w tym schemacie, jeśli dodasz `__all__` do pliku inicjalizacyjnego w katalogu głównym pakietu (z zastrzeżeniem wszystkich standardowych ostrzeżeń o wadach from * omówionych w rozdziale 23.):

```
# dir1/__init__.py
__all__ = ['mod']

# dir1/__main__.py
from dir1 import *
print('Uruchomiono dir1.__main__.py:', mod.var.upper())
```

Jeżeli jesteś pewien, że Twój kod będzie wykorzystywany jedynie jako część pakietu, możesz używać importów względnych we wszystkich jego plikach, aby uzyskać dostęp do modułów tego samego pakietu. Co więcej, rozwiązanie z użyciem ścieżek bezwzględnych można zastosować *jedynie* do plików, które mają być uruchamiane jako *skrypty* najwyższego poziomu. Inne pliki, które są tylko importowane, mogą używać importów względem pakietu.

W obu przypadkach importy względne mają tę zaletę, że moduł o tej samej nazwie zawarty w `sys.path` nie zastąpi przypadkowo kluczowego modułu w pakiecie. Jednakże importowanie względne ogranicza plik do ról jedynie pakietowych, a jeśli chcesz, aby pliki pakietu wspierały *zarówno* uruchomienie, jak i importowanie, importy względne nie są najlepszą opcją.

Należy również zauważyć, że pomijamy tu pewne szczegóły ze względu na ograniczone miejsce. Na przykład tryb `-m` wyjątkowo sprawdza również bieżący katalog roboczy dla importów bezwzględnych, przez co nie musimy ustawiać zmiennej `PYTHONPATH` dla tego trybu. Ponadto niektóre źródła sugerują, że niepowodzenia importu względnego w skryptach wynikają z ich nazwy `'__main__'`, ale pliki uruchamiane z `-m` mają tę samą nazwę i używają innych protokołów, które są zbyt złożone, aby je tutaj omówić.

Ponieważ importy względem pakietu są zarówno podatne na zmiany, jak i głównie przeznaczone dla programistów tworzących biblioteki kodu na dużą skalę dla innych, zainteresowanych tych tematem odsyłamy do dokumentacji Pythona po więcej szczegółów. Moduły są, delikatnie mówiąc, narzędziami bogatymi w funkcje, ale dobrą wiadomością jest to, że pakiety przestrzeni nazw, temat ostatniego podrozdziału, nie dodają żadnej nowej składni, lecz jedynie pozwalają modułowi obejmować wiele folderów. Przejdźmy dalej, aby zobaczyć, jak to działa.

Pakiety przestrzeni nazw

Teraz, gdy wiesz już wiele o pakietach i importowaniu względnym, pozostała do omówienia jeszcze jedna kwestia związana z pakietami. Ewolucyjna ścieżka modułów w Pythonie ostatecznie doprowadziła do tego, co jest znane jako **pakiety przestrzeni nazw** — pakiety, które mogą składać się z jednego lub więcej folderów znajdujących się w różnych miejscach na ścieżce wyszukiwania modułów.

Chociaż pakiety podzielone na różne foldery są raczej nieczęsto stosowane w praktyce, uogólnienia wprowadzone w celu wsparcia pakietów przestrzeni nazw w *algorytmie* wyszukiwania

importu (procedurze) umożliwiają również tworzenie pakietów bez plików *__init__.py*. Ponieważ ten zrewidowany algorytm to teraz po prostu wyszukiwanie importu, pakiety przestrzeni nazw są trochę przypadkowym tematem.

Aby zrozumieć miejsce pakietów przestrzeni nazw w szerszym kontekście modułów, a także zmiany, które one wprowadziły, potrzebujemy krótkiej lekcji historii.

Modele importowania w Pythonie

Python ma cztery modele importu. Poniżej wymieniono je od najstarszych do najnowszych, z reprezentatywnymi, ale niepełnymi przykładami (jak widzieliśmy, `from` pozwala również na użycie symbolu wieloznacznego *, a `reload` przeładowuje dowolny przekazany do niego moduł, lecz to są tylko tematy dodatkowe):

Podstawowe moduły

Pierwszy, oryginalny model importowania: import plików i ich zawartości względem ścieżki wyszukiwania modułów `sys.path`:

```
import moduł
from moduł import nazwa
```

Podstawowe pakiety

Pierwszy, oryginalny model importowania pakietów, który dodaje rozszerzenia ścieżki katalogu względem ścieżki wyszukiwania modułu `sys.path`, gdzie każdy pakiet jest zawarty w jednym katalogu i ma plik inicjujący:

```
import folder.folder.moduł
from folder.moduł import nazwa
```

Import względem pakietu

Model zastosowany do importu wewnątrz pakietów, omawianego w poprzednim podrozdziale, wraz z jego względnymi lub bezwzględnymi schematami wyszukiwania dla importów z użyciem kropek lub bez:

```
from . import moduł
from .moduł import nazwa
```

Pakiety przestrzeni nazw

Najnowszy model pakietu przestrzeni nazw, który jest nadal zależny od `sys.path`, ale pozwala pakietom składać się z wielu katalogów i nie wymaga pliku inicjującego:

```
import dowolny_folder.dowolny_folder.moduł
from dowolny_folder import nazwa
```

Pierwsze dwa modele są całkowicie samowystarczalne, trzeci zaostrza kolejność wyszukiwania i rozszerza składnię dla importów wewnątrz pakietu, a czwarty całkowicie odwraca niektóre podstawowe koncepcje i wymagania poprzedniego modelu pakietu. W praktyce spowodowało to, że Python posiadają teraz tylko dwa warianty pakietów:

- Oryginalny model, obecnie znany jako **pakiety regularne** lub po prostu **zwykłe** (ang. *regular packages*).
- Model alternatywny, znany jako **pakiety przestrzeni nazw** (ang. *namespace packages*).

Oryginalne i nowe modele pakietów nie wykluczają się wzajemnie i mogą być używane jednocześnie w tym samym programie. W rzeczywistości nowy model pakietów przestrzeni nazw działa jako *opcja rezerwowa*, używana tylko wtedy, gdy normalne moduły i standardowe pakiety o tej samej nazwie nie są obecne w ścieżce wyszukiwania modułów. Pomimo ich efektownej nazwy pakiety przestrzeni nazw są w rzeczywistości jedynie modyfikacją wyszukiwania importu, o czym przekonasz się w kolejnych punktach.

Uzasadnienie dla pakietów przestrzeni nazw

Pakiet przestrzeni nazw nie różni się zasadniczo od zwykłego pakietu; to po prostu nieco inny sposób tworzenia pakietów. Co więcej, na najwyższym poziomie nadal są one względne w stosunku do ścieżki `sys.path`: pierwszy komponent ścieżki pakietu przestrzeni nazw (zapisywanej z kropkami) musi nadal znajdować się w normalnej ścieżce wyszukiwania modułów.

Jednak pod względem budowy fizycznej oba rodzaje pakietów mogą się znacznie od siebie różnić. Zwykłe pakiety nadal muszą być zapisane w jednym katalogu i posiadać plik *__init__.py*, który jest uruchamiany automatycznie. W przeciwieństwie do nich nowe pakiety przestrzeni nazw nie mogą zawierać pliku *__init__.py* i mogą składać się z wielu katalogów wykorzystywanych podczas importu. W rzeczywistości *żaden* z katalogów tworzących pakiet przestrzeni nazw nie może mieć pliku *__init__.py*, ale zawartość zagnieżdżona w każdym z nich jest traktowana jak pojedynczy pakiet.

Uzasadnienie dla istnienia pakietów przestrzeni nazw jest zakorzenione w celach *instalacji* pakietów, które mogą wydawać się nie do końca oczywiste, chyba że jesteś odpowiedzialny za takie zadania. Krótko mówiąc, pakiety przestrzeni nazw radykalnie rozwiązują problem związany z możliwością wystąpienia kolizji wielu plików *__init__.py* podczas scalania części pakietu poprzez całkowite usunięcie tego pliku. Ponadto dzięki zapewnieniu standardowej obsługi pakietów, które można podzielić na wiele katalogów i umieszczać w kolejnych wpisach w ścieżce `sys.path`, pakiety przestrzeni nazw zwiększają elastyczność instalacji i zapewniają zintegrowany mechanizm zastępujący wiele niekompatybilnych ze sobą rozwiązań, które były używane do tej pory.

Chociaż jest jeszcze zbyt wcześnie, aby ocenić rzeczywistą przydatność pakietów przestrzeni nazw, przeciętni użytkownicy Pythona zwykle określają je jako bardzo użyteczne i alternatywne rozszerzenie zwykłego modelu pakietu — takie, które nie wymaga plików inicjujących i pozwala na użycie dowolnego katalogu kodu jako pakietu do zaimportowania. Aby przekonać się dlaczego, przejdziemy do szczegółów wyszukiwania importów.

Algorytm wyszukiwania modułu

Aby naprawdę zrozumieć, w jaki sposób pakiety przestrzeni nazw zmieniają i rozszerzają wyszukiwanie importów, musimy zajrzeć pod „maskę”, aby zobaczyć, jak działa operacja importowania.

Jak widzieliśmy, Python przeszukuje zestaw potencjalnych folderów. Dla składników importów znajdujących się *najbardziej z lewej* zestawem kandydatów są wszystkie katalogi wymienione

na ścieżce wyszukiwania modułów `sys.path`. Dla składników pakietów, które są zagnieżdżone w importach *pakietów* lub wymienione w importach *względnych*, zestawem kandydatów jest tylko sam pakiet. Podczas gdy foldery pakietów mają tylko jedną instancję danej nazwy, ścieżki wyszukiwania mogą mieć ich wiele.

Sposób, w jaki wyszukiwanie importu wybiera elementy spośród tych kandydatów, jest bardziej subtelny, niż się wydaje. Dla każdego folderu (*nazwa_katalogu*) w zestawie kandydatów wyszukiwania importu Python testuje różne dopasowania do zaimportowanej nazwy w następującej kolejności (używając / dla separatorów folderów i {…} do wskazania wyboru):

1. Jeżeli zostanie znaleziony plik **nazwa_katalogu\nazwa***__init__*.py*, importowany i zwracany jest zwykły pakiet.
2. Jeżeli zostanie znaleziony plik **nazwa_katalogu\nazwa**.*{py, pyc* lub inne rozszerzenie modułu}, importowany i zwracany jest zwykły pakiet.
3. Jeżeli zostanie znaleziony katalog **nazwa_katalogu\nazwa**, zostanie on zapisany, a skanowanie będzie kontynuowane w następnym katalogu ze ścieżki wyszukiwania.
4. Jeżeli żaden z powyższych plików lub katalogów nie zostanie znaleziony, skanowanie będzie kontynuowane od następnego katalogu ze ścieżki wyszukiwania.

Jeżeli skanowanie ścieżki wyszukiwania zakończy się bez zwracania modułu lub pakietu według kroków 1. lub 2., a co najmniej jeden katalog został zapisany w kroku 3., wówczas tworzony jest *pakiet przestrzeni nazw*. W przeciwnym razie zgłaszany jest błąd.

Tworzenie pakietu przestrzeni nazw odbywa się natychmiast i nie jest odraczane do momentu wystąpienia importu na poziomie niżej. Nowy pakiet przestrzeni nazw jest obiektem modułu pozbawionym atrybutu `__file__`, ale ma atrybut `__path__` ustawiony na iterowalną listę ścieżek katalogów, które zostały znalezione i zarejestrowane podczas skanowania w kroku 3.

Atrybut `__path__` jest następnie wykorzystywany w późniejszych, głębszych dostępach do przeszukiwania wszystkich składników pakietu — każdy katalog w ścieżce `__path__` pakietu przestrzeni nazw jest przeszukiwany, gdy poszukiwane są bardziej zagnieżdżone elementy, podobnie jak to miało miejsce w przypadku katalogu zwykłego pakietu.

Patrząc z innej strony, atrybut `__path__` pakietu przestrzeni nazw pełni tę samą rolę dla komponentów niższego poziomu, co `sys.path` dla elementów ścieżek importu pakietów; staje się „ścieżką nadrzędną" pozwalającą na dostęp do położonych niżej elementów przy użyciu tej samej czteroetapowej procedury, którą omówiliśmy przed chwilą.

W efekcie pakiet przestrzeni nazw jest rodzajem **wirtualnej konkatenacji** katalogów zdefiniowanych za pomocą wpisów w ścieżce wyszukiwania modułów. Jednak po utworzeniu pakietu przestrzeni nazw nie ma funkcjonalnej różnicy między nim a zwykłym pakietem; obsługuje wszystko, czego nauczyliśmy się dla zwykłych pakietów, w tym importowanie względne wewnątrz pakietu.

Co ważne, ponieważ *pojedynczy katalog* pozbawiony pliku *__init__.py*, ale zagnieżdżony w folderze będącym kandydatem do wyszukiwania, przez krok 3. tego algorytmu jest klasyfikowany jako pakiet przestrzeni nazw, każdy taki katalog kwalifikuje się jako pakiet. Jedyną

różnicą między takim pojedynczym folderem a zwykłym folderem pakietu jest to, że ten pierwszy ma niższy *priorytet w wyszukiwaniu*. Zarówno foldery o tej samej nazwie z plikiem *__init__.py*, jak i proste moduły umieszczone w dowolnym miejscu na ścieżce wyszukiwania są wybierane na początku przez kroki 1. i 2.

Innymi słowy, chociaż zmiany wprowadzone w celu wsparcia pakietów przestrzeni nazw sprawiają, że pliki *__init__.py* stają się opcjonalne w folderach pakietów, dodanie takiego pliku, nawet jeśli jest pusty, zapewnia, że pakiet zostanie wybrany zamiast folderu o tej samej nazwie znajdującego się później na ścieżce wyszukiwania (może to również przyspieszyć import, kończąc skanowanie ścieżki wyszukiwania na kroku 1., ale jest to jednorazowe zdarzenie). To, czy ten fakt, czy inne role pliku *__init__.py*, które poznaliśmy wcześniej, uzasadniają dodanie tego pliku, będzie oczywiście zależało od Ciebie i tego, co chcesz osiągnąć.

Pakiety przestrzeni nazw w akcji

Widzieliśmy już, jak działają pakiety przestrzeni nazw w praktyce. Pierwszy krok w przykładzie na początku tego rozdziału utworzył pakiety przestrzeni nazw składające się z *jednego folderu*, ponieważ ich foldery nie miały jeszcze plików *__init__.py*. Ponownie każdy folder zagnieżdżony w folderze na ścieżce wyszukiwania `sys.path` kwalifikuje się jako pakiet, o ile nie jest ukryty przez zwykły pakiet o tej samej nazwie z plikiem *__init__.py* lub prosty moduł znajdujący się gdzieś indziej na ścieżce.

Aby zobaczyć większy efekt łączenia folderów w „wirtualnych" pakietach przestrzeni nazw, przeprowadźmy szybką demonstrację. Na początek utwórz dwa moduły w zagnieżdżonej strukturze katalogów, które mają podkatalogi o nazwie *sub* znajdujące się w różnych katalogach nadrzędnych, *part1* i *part2* (wcięcia pokazują poziom zagnieżdżenia):

```
ns/
    part1/
              sub/
                    mod1.py
    part2/
              sub/
                     mod2.py
```

Zauważ, że nie ma tutaj plików *__init__.py* — jak wspomniano wcześniej, pliki te nie mogą być używane w pakietach przestrzeni nazw, ponieważ to pliki inicjalizujące są główną różnicą w porównaniu do zwykłych pakietów umieszczonych w jednym folderze. Na potrzeby tego przykładu możemy utworzyć foldery za pomocą takich poleceń konsoli (w systemie Windows zamień / na \ i pomiń `-p` lub użyj eksploratora plików albo, jeśli wolisz, skorzystaj z gotowych folderów znajdujących się w materiałach do pobrania dla tej książki):

```
$ mkdir -p ns/part1/sub          # Dwa podfoldery o tej samej nazwie w różnych folderach
$ mkdir -p ns/part2/sub          # I podobnie w systemie Windows
```

Moduły na końcu tych ścieżek znajdziesz w przykładach 24.10 i 24.11. Te moduły wyświetlają informacje podczas importu jako ślad.

Przykład 24.10. *ns/part1/sub/mod1.py*

```
print('Wczytywanie ns/part1/sub/mod1')
```

Przykład 24.11. *ns/part2/sub/mod2.py*

```
print('Wczytywanie ns/part2/sub/mod2')
```

Jeżeli dodamy *zarówno part1*, jak i *part2* do ścieżki wyszukiwania modułów, katalog *sub* staje się pakietem przestrzeni nazw obejmującym oba katalogi, z dwoma plikami modułów dostępnymi pod tą nazwą, pomimo że znajdują się one w osobnych katalogach fizycznych. Oto zawartość plików i wymagane ustawienia ścieżek w systemie Unix (w systemie Windows użyj `set` i `:`, lub zajrzyj do rozdziału 22., gdzie znajdziesz więcej wskazówek). Użyto tutaj ścieżek względem bieżącego katalogu roboczego, ale w praktyce bardziej prawdopodobne byłoby zastosowanie dwóch pełnych ścieżek bezwzględnych:

```
$ export PYTHONPATH=ns/part1:ns/part2
```

Po bezpośrednim zaimportowaniu pakiet przestrzeni nazw staje się *wirtualnym połączeniem* poszczególnych składników katalogu i umożliwia poprzez normalny import dostęp do dalszych zagnieżdżonych części za pomocą jednej, złożonej nazwy (jak zwykle ścieżki zostały skrócone za pomocą wielokropka dla oszczędności miejsca):

```
$ python3
>>> import sub
>>> sub                                  # Pakiety przestrzeni nazw: zagnieżdżone ścieżki wyszukiwania
<module 'sub' (namespace) from
['/…/PW6W/r24/ns/part1/sub', '/…/PW6W/r24/ns/part2/sub']>

>>> from sub import mod1
Wczytywanie ns/part1/sub/mod1
>>> import sub.mod2                      # Zawartość z dwóch różnych katalogów
Wczytywanie ns/part2/sub/mod2

>>> mod1
<module 'sub.mod1' from '/…/PW6W/r24/ns/part1/sub/mod1.py'>
>>> sub.mod2
<module 'sub.mod2' from '/…/PW6W/r24/ns/part2/sub/mod2.py'>
```

Dzieje się tak również wtedy, gdy *natychmiast* importujemy nazwę pakietu za pośrednictwem przestrzeni nazw — ponieważ pakiet przestrzeni nazw jest tworzony przy pierwszym użyciu, czas przeszukiwania ścieżek jest nieistotny:

```
$ python3
>>> import sub.mod1
Wczytywanie ns/part1/sub/mod1
>>> import sub.mod2                      # Jeden pakiet składający się z dwóch katalogów
Wczytywanie ns/part2/sub/mod2

>>> sub.mod1
<module 'sub.mod1' from '/…/PW6W/r24/ns/part1/sub/mod1.py'>
>>> sub.mod2
<module 'sub.mod2' from '/…/PW6W/r24/ns/part2/sub/mod2.py'>

>>> sub
<module 'sub' (namespace) from
```

```
['/…/PW6W/Chapter24/ns/part1/sub', '/…/PW6W/r24/ns/part2/sub']>
>>> sub.__path__
_NamespacePath(
['/…/PW6W/r24/ns/part1/sub', '/…/PW6W/r24/ns/part2/sub'])
```

Co ciekawe, *import względny* działa również w pakietach przestrzeni nazw, tak jak w przykładzie 24.12.

Przykład 24.12. *ns/part1/sub/mod1.py (zmieniony)*

```
from . import mod2
print('Wczytywanie ns/part1/sub/mod1')
```

Dodana instrukcja importu względnego odwołuje się do pliku w pakiecie, mimo że sam plik, do którego się odwołuje, znajduje się w *innym katalogu*:

```
$ python3
>>> import sub.mod1                    # Import względny mod2 z innego folderu
Wczytywanie ns/part2/sub/mod2
Wczytywanie ns/part1/sub/mod1
>>> import sub.mod2                    # Już zaimportowane przez mod1.py — nie jest ponownie
uruchamiany
>>> sub.mod2
<module 'sub.mod2' from '/…/PW6W/r24/ns/part2/sub/mod2.py'>
```

Jak widać, pakiety przestrzeni nazw są pod każdym względem takie jak zwykłe pakiety z jednym katalogiem, z wyjątkiem dzielonego magazynu fizycznego. To właśnie dlatego foldery z kodem bez pliku *__init__.py* są dokładnie takie same jak zwykłe pakiety, z tym że nie mają kodu inicjującego, który mógłby być uruchamiany automatycznie — są po prostu instancją pakietu przestrzeni nazw z jednym katalogiem.

Jeśli chcesz się dowiedzieć więcej na ten temat, sięgnij po inne źródła dotyczące Pythona. Pakiety przestrzeni nazw są potencjalnie użytecznym narzędziem, ale większość osób uczących się Pythona prawdopodobnie więcej skorzysta na opanowaniu pakietów, które odwzorowują się na rzeczywiste foldery, zanim podejmie się wyzwania, jakim są te pakiety wirtualnie rozprzestrzenione po katalogach.

Podsumowanie rozdziału

W niniejszym rozdziale wprowadziliśmy model *importowania pakietów* Pythona — opcjonalną, lecz przydatną metodę jawnego wymienienia części ścieżki katalogów prowadzących do modułów. Importowanie pakietów nadal odbywa się względem katalogu ze ścieżki wyszukiwania modułów, jednak zamiast pozostawić Pythonowi ręczne przejście ścieżki wyszukiwania, skrypt może podać resztę ścieżki do modułu w sposób jawny. Pakiety mogą być tworzone przy użyciu prostego folderu i mogą sprawić, że importy będą bardziej znaczące, uprościć ustawienia ścieżki wyszukiwania importów oraz rozwiązać niejednoznaczności, gdy istnieje więcej niż jeden moduł o tej samej nazwie — nazwa otaczającego katalogu czyni je unikatowym.

Omówiliśmy również model *importów względnych*, stosowany wyłącznie w pakietach. Jest to sposób wymuszenia importu plików zdefiniowanych w tym samym pakiecie polegający na

zastosowaniu wiodących kropek w ścieżce importu w instrukcji `from`. Na koniec zbadaliśmy *pakiety przestrzeni nazw*, które pozwalają, aby pakiet składał się z wielu katalogów fizycznych; jest to w pewnym sensie „ostatnia deska ratunku" podczas wyszukiwania importów, która całkowicie usunęła wymagania dotyczące plików inicjalizacyjnych w pakietach znajdujących się w jednym folderze.

W kolejnym rozdziale omówimy kilka bardziej zaawansowanych zagadnień związanych z modułami, takich jak użycie atrybutu `__name__` czy `__getattr__`, importowanie nazw oraz narzędzia przydatne podczas pracy z modułami. Jak zawsze jednak zamkniemy bieżący rozdział krótkim quizem sprawdzającym wiadomości w nim przedstawione.

Sprawdź swoją wiedzę — quiz

1. Jaki jest cel umieszczania pliku *__init__.py* w katalogu pakietu modułu?
2. W jaki sposób możemy uniknąć powtarzania pełnej ścieżki pakietu za każdym razem, gdy odnosimy się do zawartości pakietu?
3. Które katalogi wymagają, by znajdował się w nich plik *__init__.py*?
4. Kiedy w przypadku importowania pakietów musimy użyć instrukcji `import` zamiast `from`?
5. Jaka jest różnica między instrukcją `from pkg import name` a `from . import name`?
6. Czym jest pakiet przestrzeni nazw?

Sprawdź swoją wiedzę — odpowiedzi

1. Plik *__init__.py* służy do deklarowania i inicjalizacji zwykłego pakietu modułu. Python automatycznie wykonuje jego kod za pierwszym razem, gdy w procesie importujemy moduł za pośrednictwem katalogu. Przypisane zmienne pliku stają się atrybutami obiektu modułu utworzonego w pamięci i odpowiadającego temu katalogowi. Nie jest on opcjonalny — nie możemy importować modułów za pomocą składni pakietów, jeżeli katalog nie zawiera tego pliku.
2. Aby bezpośrednio skopiować zmienne z pakietu, należy użyć instrukcji `from` lub rozszerzenia `as` w połączeniu z instrukcją `import`, zastępując ścieżkę krótszym synonimem. W obu przypadkach ścieżka wymieniana jest tylko w jednym miejscu, czyli instrukcji `from` bądź `import`.
3. Podchwytliwe pytanie! We wcześniejszych wersjach Pythona każdy katalog wymieniony w instrukcjach `import` i `from` musiał zawierać plik *__init__.py*, ale teraz nie jest już to obowiązkowe, gdyż wprowadzono udogodnienia dla pakietów przestrzeni nazw w algorytmie wyszukiwania modułów. Jak wspomniano w odpowiedzi 1., te foldery nadal pełnią ważne, choć opcjonalne, role, takie jak między innymi zwiększenie priorytetu podczas wyszukiwania importu dla folderu.
4. W przypadku pakietów instrukcji `import` musimy użyć w miejsce `from` jedynie wtedy, gdy potrzebujemy uzyskać dostęp do tej *samej zmiennej* zdefiniowanej w więcej niż jednej

ścieżce. Dzięki instrukcji `import` ścieżka sprawia, że referencje stają się unikalne, natomiast instrukcja `from` pozwala na wykorzystywanie tylko jednej wersji danej nazwy zmiennej (o ile do zmiany nazwy nie użyjesz rozszerzenia `as`).

5. Instrukcja `from pkg import name` definiuje import *bezwzględny*: pakiet `pkg` jest wyszukiwany z pominięciem katalogu pakietu, w którym występuje ta instrukcja, to znaczy przeszukiwana jest wyłącznie ścieżka `sys.path`. Instrukcja `from . import nazwa` definiuje import *względny*: nazwa `name` jest poszukiwana w odniesieniu do pakietu, w którym występuje ta instrukcja.
6. *Pakiet przestrzeni nazw* jest rozszerzeniem modelu importu, który reprezentuje pakiet składający się z jednego lub większej liczby katalogów nie zawierających plików *__init__.py*. Gdy Python znajdzie takie katalogi podczas wyszukiwania importu i nie znajdzie wcześniej prostego modułu lub zwykłego pakietu o takiej nazwie, tworzy pakiet przestrzeni nazw, który jest wirtualnym połączeniem wszystkich znalezionych katalogów o nazwie odpowiadającej nazwie żądanego modułu. Dalsze zagnieżdżone komponenty są wyszukiwane we wszystkich katalogach pakietu przestrzeni nazw. W rezultacie powstaje pakiet bardzo podobny do zwykłego pakietu, ale jego zawartość może być podzielona na wiele katalogów.

Skorowidz

A

B

C

D

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

Ź

Nic dziwnego, że programiści kochają Pythona: jest wszechstronny, czytelny, darmowy i działa na każdej platformie. Można się go stosunkowo szybko nauczyć — ale jest jeden warunek: aby w pełni wykorzystać jego możliwości, trzeba zdobyć solidne podstawy, zrozumieć kilka trudniejszych koncepcji i... dużo ćwiczyć, pisząc własny kod.

Ta książka stanowi kompleksowe i obszerne wprowadzenie do języka Python. Pomoże Ci opanować jego podstawy i przygotuje do praktycznego zastosowania nabytej wiedzy. To wydanie zostało zaktualizowane i rozszerzone, aby odzwierciedlić zmiany zachodzące w świecie Pythona. Pominięto omówienie nieaktualnej wersji 2.X, opisano nowe narzędzia, dodane do Pythona w wersji 3.12, a także innych jego edycji, które są dziś szeroko używane. Naukę ułatwią Ci liczne quizy, ćwiczenia, pomocne ilustracje i przykładowe fragmenty kodu. To idealne kompendium dla każdego, kto chce szybko zacząć programować w Pythonie i tworzyć wydajny kod wysokiej jakości.

Zaufaj Pythonowi i odkryj świat programowania!

W książce:

- ogólny model składni Pythona
- wbudowane typy obiektów i ich przetwarzanie
- stosowanie funkcji i organizowanie kodu w modułach i pakietach
- obsługa wyjątków i inne narzędzia programistyczne
- zaawansowane narzędzia Pythona, między innymi dekoratory, deskryptory i metaklasy
- kod idiomatyczny, który działa na różnych platformach

Mark Lutz od niemal trzydziestu lat zajmuje się nauczaniem Pythona. Jest autorem niezwykle popularnych i wielokrotnie wznawianych książek poświęconych temu językowi programowania, przeprowadził też setki sesji treningowych w tym zakresie. Obecne tworzy w Pythonie aplikacje, które działają zarówno na komputerach osobistych, jak i telefonach.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-2942-5
9 788328 929425
Cena: 199,00 zł