Helion

Python ninja

70 sekretnych receptur i taktyk programistycznych

Packt›

Cody Jackson

Tytuł oryginału: Secret Recipes of the Python Ninja: Over 70 recipes that uncover powerful programming tactics in Python

Tłumaczenie: Agnieszka Górczyńska

ISBN: 978-83-283-5317-6

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/pytnin.zip

Printed in Poland.

Spis treści

ROZDZIAŁ 2

Zastosowanie interpretera Pythona

W tym rozdziale dowiesz się nieco więcej na temat interpretera Pythona jako zarówno narzędzia interaktywnego, jak i przeznaczonego do uruchamiania programów Pythona. W szczególności omówię następujące zagadnienia:

- uruchamianie środowiska Pythona;
- wykorzystanie opcji polecenia Pythona;
- praca ze zmiennymi środowiskowymi;
- definiowanie skryptu jako wykonywalnego;
- modyfikacja procesu uruchamiania interpretera interaktywnego;
- alternatywne implementacje Pythona;
- instalowanie Pythona w Windowsie;
- osadzanie Pythona w innych aplikacjach;
- zastosowanie alternatywnej powłoki Pythona — IPython;
- zastosowanie alternatywnej powłoki Pythona — bpython;
- zastosowanie alternatywnej powłoki Pythona — DreamPie.

Wprowadzenie

Jedną z zalet Pythona jest to, że zalicza się on do języków interpretowanych, a nie kompilowanych. Dlatego też kod Pythona jest przetwarzany w momencie jego wywołania i nie musi być wcześniej skompilowany przed użyciem. Dzięki temu języki interpretowane zwykle mają powłokę interaktywną pozwalającą użytkownikom na przetestowanie kodu i otrzymanie natychmiastowego wyniku bez konieczności tworzenia oddzielnego pliku kodu źródłowego.

Oczywiście aby w jak największym stopniu wykorzystać funkcjonalność języka programowania, konieczne jest posiadanie trwałych plików kodu źródłowego. Podczas pracy z powłoką interaktywną cały kod znajduje się w pamięci RAM i zakończenie sesji oznacza definitywną utratę tego kodu. Dlatego powłoka interaktywna to doskonały sposób na szybkie przetestowanie pomysłów, ale zdecydowanie nie chcesz za jej pomocą tworzyć pełnych programów.

W tym rozdziale pokażę, jak uruchamiać programy Pythona, a także jak wykorzystać funkcjonalność języka za pomocą jego powłoki interaktywnej. Poruszę również temat funkcji specjalnych w systemie operacyjnym Windows. Na końcu przedstawię wybrane alternatywne powłoki Pythona, które być może zechcesz wypróbować.

Uruchamianie środowiska Pythona

Domyślnie przy instalacji Pythona do systemowej ścieżki dostępu dodawany jest katalog zawierający interpreter Pythona. Dzięki temu wpisanie w wierszu polecenia `python` wywoływać będzie interpreter tego języka.

To polecenie jest najczęściej używane do uruchamiania skryptów. Przy czym niekiedy wymagane jest uruchomienie określonej wersji Pythona dla konkretnego programu.

Jak to zrobić?

1. Najprostsza postać polecenia przeznaczonego do uruchomienia programu Pythona przedstawia się następująco:

```
$ python <nazwa_skryptu>.py
```

2. W kolejnym fragmencie kodu przedstawiłem przykłady uruchamiania konkretnych wersji Pythona, według potrzeb:

```
$ python2 nazwa_skryptu.py # Użycie najnowszego dostępnego wydania Python 2
$ python2.7 ... # Użycie wydania Python 2.7
$ python3 ... # Użycie najnowszego dostępnego wydania Python 3
$ python3.5.2 ... # Użycie wydania Python 3.5.2
```

Jak to działa?

Wydanie polecenia `python2` lub `python3` powoduje uruchomienie najnowszej zainstalowanej wersji podanego wydania Pythona. Z kolei pozostałe przykłady pokazały uruchomienie konkretnej wersji interpretera Pythona. Niezależnie od wersji dostępnych w witrynie internetowej Pythona możesz użyć tylko tych, które masz zainstalowane w systemie.

To ma sens, ponieważ programista być może chce zapewnić obsługę starszego oprogramowania, którego wybrane funkcje mogą być niezgodne z najnowszymi wydaniami języka. Dlatego wywołanie konkretnej wersji gwarantuje programiście możliwość użycia prawidłowego środowiska.

Opcje polecenia python

W przypadku użycia nieinteraktywnego interpreter Pythona monitoruje powłokę i przetwarza wszystkie dane wejściowe, zanim polecenie będzie faktycznie wykonane. Oto wszystkie opcje dostępne podczas uruchamiania Pythona z poziomu powłoki:

```
python [-bBdEhiIOqsSuvVWx?] [-c polecenie | -m nazwa-modułu | skrypt | - ]
[argumenty]
```

Podczas pracy z **powłoką** (systemy w rodziny UNIX/Linux) lub **wierszem polecenia** (Windows) przykłady poleceń często zawierają nawias kwadratowy oznaczający parametry opcjonalne. W omawianym przykładzie mamy trzy grupy opcjonalne możliwe do użycia w trakcie wywołania polecenia `python`: opcje ogólne, opcje interfejsu i argumenty.

Jak to zrobić?

1. W trakcie wywołania Pythona z poziomu powłoki masz do dyspozycji kilka opcji. Aby przejść do trybu interaktywnego, wystarczy wydać polecenie `python` bez żadnych opcji:

```
$ python
Python 3.6.3 |Anaconda, Inc.| (default, Oct 13 2017, 12:02:49)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

2. Aby uruchomić zwykły program Pythona bez żadnych opcji specjalnych, wystarczy podać jego nazwę po poleceniu `python`:

```
$ python <nazwa_skryptu>.py
```

3. Wykonanie serii poleceń Pythona bez przejścia do trybu interaktywnego lub podania nazwy pliku jest możliwe dzięki użyciu opcji `–c`:

```
$ python -c "print('Witaj, świecie!')"
```

4. Aby wywołać moduł Pythona jako samodzielny program, należy użyć opcji `–m`:

```
$ python -m random
```

5. Inne wybrane opcje zostaną przedstawione w dalszej części rozdziału.

Jak to działa?

Powłoka Pythona akceptuje opcje interfejsu, ogólne, różne i argumenty. Każda z tych grup jest opcjonalna i większość programistów nie musi z nich korzystać. Jednak dobrze jest wiedzieć, czym dysponujesz, na wypadek, gdy zechcesz wyjść poza podstawy.

Opcje interfejsu

W przypadku wywołania bez żadnych opcji interpreter Pythona zostaje uruchomiony w trybie interaktywnym. W tym trybie monitoruje powłokę pod kątem poleceń Pythona i wykonuje je po ich wpisaniu.

Aby opuścić tryb interaktywny, należy wprowadzić znak **końca pliku** (ang. *end-of-file* — EOF) — w systemach z rodziny UNIX/Linux trzeba nacisnąć klawisze *Ctrl+D*, natomiast w Windowsie *Ctrl+Z*. (Normalnie znak EOF jest dostarczany automatycznie podczas odczytu programu z pliku. Tak się nie dzieje w przypadku trybu interaktywnego i użytkownik musi samodzielnie wprowadzić ten znak).

Opcje omówione w tym punkcie mogą być łączone z opcjami różnymi, które omówiłem w jednym z kolejnych punktów.

- `-c <"polecenie">` — użycie tej opcji spowoduje, że Python wykona podane polecenie. Wprowadzone polecenie może składać się z jednego lub więcej słów kluczowych rozdzielonych znakami nowego wiersza. Podobnie jak w przypadku zwykłego kodu Pythona, także tutaj należy pamiętać o białych znakach, które w kodzie źródłowym Pythona mają znaczenie. Polecenie przeznaczone do wykonania musi zostać ujęte w znaki cytowania (cudzysłów lub apostrof).
- `-m <moduł>` — ta opcja powoduje przeszukanie przez interpreter systemowej ścieżki dostępu, znalezienie podanego modułu i wykonanie zawartości zdefiniowanej w module `__main__`. Moduły wykonywane za pomocą tej metody nie wymagają rozszerzenia *.py*. Istnieje możliwość podania pakietów modułów i wówczas jako moduł `__main__` Python wykona `<pkg>.__main__`.

  Tej opcji nie można używać wraz ze skompilowanymi modułami C, co dotyczy również modułów wbudowanych, ponieważ nie są one kodem Pythona. Natomiast można ją stosować z prekompilowanymi plikami Pythona, *.pyc*, nawet jeśli niedostępny jest oryginalny plik kodu źródłowego — to jest czysty kod Pythona.

  W przypadku użycia tej opcji wykonany zostanie wszelki kod umieszczony pod wierszem `if __name__ == "__main__"`. Jest to dobre miejsce na umieszczenie kodu konfiguracyjnego lub przeprowadzającego testy.
- `<skrypt>` — ta opcja spowoduje wykonanie kodu Pythona znajdującego się w podanym skrypcie. Skrypt ten musi znajdować się w ścieżce dostępu systemu plików (bezwzględnej lub względnej) wskazującej zwykły plik Pythona, katalog zawierający plik *__main__.py* lub archiwum wraz z plikiem *__main__.py*.
- `-` — sam myślnik nakazuje interpreterowi odczyt danych pochodzących ze standardowego wejścia (`sys.stdin`). Jeżeli standardowe wejście jest połączone z terminalem, wówczas zostanie uruchomiony zwykły tryb interaktywny. Wprawdzie klawiatura to domyślne urządzenie wejściowe, ale `sys.stdin` może zostać przekierowane na plik dowolnego typu, np. zwykły plik tekstowy lub plik w formacie CSV.

Opcje ogólne

Podobnie jak w przypadku większości programów, opcje ogólne Pythona są znane z produktów komercyjnych, jak również opracowywanych samodzielnie.

- `-?`. `-h`, `--help` — każda z tych opcji spowoduje wyświetlenie krótkiego opisu polecenia i wszystkich dostępnych opcji.
- `-V`, `-VV`, `--version` — opcje `-V` i `--version` powodują wyświetlenie numeru wersji interpretera Pythona, z kolei opcja `-VV` oznacza przejście do trybu bardziej szczegółowego (tylko dla Pythona 3), który dostarcza więcej informacji, np. o środowisku Pythona lub użytej wersji kompilatora GCC.

Opcje różne

Dla polecenia `python` dostępnych jest wiele opcji różnych. Wprawdzie większość z nich jest dostępna dla wydań Pythona 2 i 3, ale między nimi mogą występować pewne różnice. Jeżeli pojawią się jakiekolwiek wątpliwości, wówczas najlepiej zajrzeć do dokumentacji znajdującej się na stronie *https://docs.python.org/2.7/using/cmdline.html*. (Upewnij się, że przeglądasz dokumentację do używanej przez siebie wersji Pythona).

Oto krótkie omówienie wybranych opcji:

- `-b`, `-bb` — powoduje wyświetlenie ostrzeżenia podczas porównywania typów `bytes` i `bytesarray` z `str` lub porównywania typu `bytes` z `int`, natomiast podwójne `b` powoduje wygenerowanie błędu zamiast ostrzeżenia.
- `-B` — powoduje nietworzenie plików kodu bajtowego *.pyc* podczas importowania modułów kodu źródłowego. Ta opcja jest powiązana z `PYTHONDONTWRITEBYTECODE`.
- `-d` — powoduje włączenie analizatora danych wyjściowych debugowania. Ta opcja jest powiązana z `PYTHONDEBUG`.
- `-E` — powoduje zignorowanie wszystkich zdefiniowanych zmiennych środowiskowych `PYTHON*`, takich jak `PYTHONDEBUG`.
- `-i` — gdy skrypt jest pierwszym argumentem polecenia `python` lub została użyta opcja `-c`, omawiana tutaj opcja powoduje przejście interpretera do trybu interaktywnego po wykonaniu skryptu bądź polecenia. Zmiana trybu następuje nawet wtedy, gdy `sys.stdin` nie jest terminalem. Jest to użyteczna możliwość w przypadku zgłoszenia wyjątku, ponieważ powala programiście na interaktywne prześledzenie stosu wywołań.
- `-I` — powoduje uruchomienie interpretera w trybie izolowanym (automatycznie oznacza także użycie opcji `-E` i `-s`). Tryb izolowany oznacza, że `sys.path` nie przechwytuje katalogu skryptu lub katalogu pakietów użytkownika. Ponadto zostają zignorowane wszystkie zmienne `PYTHON*`. Mogą zostać nałożone dodatkowe ograniczenia, aby uniemożliwić użytkownikowi wstrzyknięcie do programu Pythona kodu o złośliwym działaniu.
- `-J` — ta opcja jest zarezerwowana do użycia przez implementację Jython.
- `-O`, `-OO` — powodują włączenie podstawowych optymalizacji. Jak już wspomniałem w rozdziale 1., te opcje powodują usunięcie z kodu Pythona wywołań `assert()`.

Są powiązane z PYTHONOPTIMIZE. Użycie -OO skutkuje usunięciem z kodu również komentarzy typu docstring.

- -q — tryb cichy zawiesza wyświetlanie przez interpreter Pythona informacji o prawach autorskich i wersji, nawet w trybie interaktywnym. Ta opcja okazuje się użyteczna podczas wykonywania programów odczytujących dane ze zdalnych systemów i nie wymagających przedstawiania informacji.
- -R — ta opcja nie ma znaczenia w wydaniu Python 3.3 i nowszych. Powoduje włączenie losowości wartości hash przez użycie __hash__() dla obiektów typu str, bytes i datetime. Pozostają one stałe w poszczególnych procesach Pythona, ale inne w różnych wywołaniach Pythona. Ta opcja jest powiązana z PYTHONHASHSEED.
- -s — powoduje niedodawanie katalogu *site-packages* użytkownika do ścieżki w sys.path. Skutkuje to wymaganiem od użytkownika, aby podał ścieżkę dostępu do żądanych pakietów znajdujących się w wymienionym katalogu.
- -S — wyłącza importowanie modułu site i związane z tym modyfikacje systemowej ścieżki dostępu sys.path. Nawet jeśli moduł site zostanie później wyraźnie zaimportowany, modyfikacje te nadal będą wyłączone. Wywołanie site.main() jest wymagane, aby zezwolić na użycie modyfikacji.
- -u — wymusza niebuforowanie danych binarnych dla strumieni wyjściowych stdout i stderr. Nie ma wpływu na tekstowe operacji wejścia – wyjścia w trybie interaktywnym lub buforowanie bloków w trybie nieinteraktywnym. Ta opcja jest powiązana z PYTHONUNBUFFERED.
- -v, -vv — powoduje wyświetlenie komunikatu za każdym razem, gdy moduł jest inicjalizowany. Komunikat ów podaje położenie (plik lub wbudowany moduł) komponentu wczytującego ten moduł. Ponadto będą wyświetlane informacje o operacjach porządkowych przeprowadzanych po zakończeniu pracy z modułem. Użycie opcji -vv spowoduje wyświetlenie komunikatu za każdym razem, gdy sprawdzany jest plik w poszukiwaniu modułu. Ta opcja jest powiązana z PYTHONVERBOSE.
- -W <arg> — kontroluje wyświetlanie ostrzeżeń. Domyślnie każde ostrzeżenie jest wyświetlane tylko jednokrotnie dla wiersza kodu, który je wygenerował. Istnieje możliwość użycia wielu opcji -W, każdej z innym argumentem. W przypadku dopasowania więcej niż jednej opcji zostanie zwrócona ostatnia dopasowana. Ta opcja jest powiązana z PYTHONWARNINGS.

  Dostępne są następujące argumenty:

  - ignore — zignorowanie wszystkich ostrzeżeń.
  - default — wyraźne żądanie zachowania domyślnego, czyli jednokrotnego wyświetlenia każdego ostrzeżenia dla każdego wiersza kodu źródłowego, który spowodował wygenerowanie ostrzeżenia. Nie ma tutaj znaczenia, ile razy wiersz będzie przetwarzany.
  - all — wyświetlenie ostrzeżenia za każdym razem, gdy wystąpi. Ostrzeżenie może zostać wyświetlone wielokrotnie, jeśli powodujący je wiersz kodu będzie przetwarzany więcej niż raz, na przykład w pętli.

- `module` — wyświetlenie ostrzeżenia po jego pierwszym wystąpieniu w każdym module.
- `once` — wyświetlenie ostrzeżenia po jego pierwszym wystąpieniu w programie.
- `error` — zamiast wyświetlenia ostrzeżenia nastąpi zgłoszenie wyjątku.

Moduł `warnings` można zaimportować w programie Pythona, aby zachować kontrolę nad ostrzeżeniami generowanymi przez program.

- `-x` — pominięcie pierwszego wiersza kodu źródłowego. W systemach z rodziny UNIX/Linux pierwszy wiersz skryptu to zwykle tak zwany *shebang* i zawiera kod taki jak `#!/usr/bin/python`. W wymienionym przypadku shebang wskazuje, gdzie skrypt ma szukać środowiska Pythona. Omawiana opcja powoduje pominięcie tego wiersza. Dzięki temu można stosować inne formaty shebang niż powszechnie używane w systemach UNIX i Linux.
- `-X <wartość>` — ta opcja jest zarezerwowana dla opcji specyficznych dla implementacji, a także do przekazywania dowolnych wartości i ich pobierania za pomocą słownika `sys._xoptions`.

  Aktualnie są zdefiniowane następujące wartości:

  - `faulthandler` — zezwala na użycie modułu `faulthandler`, który zapisuje stos wywołań Pythona w przypadku wystąpienia błędów w programie.
  - `showrefcount` — działa jedynie podczas debugowania. Wyświetla całkowitą wartość licznika odwołań i liczbę użytych bloków pamięci, gdy program zakończy działanie lub po każdym poleceniu w sesji interaktywnej.
  - `tracemalloc` — za pomocą modułu `tracemalloc` rozpoczyna monitorowanie alokacji pamięci Pythona. Domyślnie na stosie wywołań znajduje się ostatnia ramka.
  - `showalloccount` — gdy program zakończy działanie, następuje wyświetlenie całkowitej liczby zaalokowanych obiektów dla poszczególnych typów. Działa jedynie po zdefiniowaniu `COUNT_ALLOCS` w trakcie kompilacji Pythona.

Zobacz również

Więcej informacji na ten temat znajdziesz także w rozdziale 1., poświęconym pracy z modułami Pythona.

Praca ze zmiennymi środowiskowymi

Zmienne środowiskowe to część systemu operacyjnego i mają wpływ na jego działanie. Python ma charakterystyczne dla siebie zmienne wpływające na sposób jego działania, a dokładniej na zachowanie interpretera. Są one przetwarzane przed opcjami powłoki, które nadpisują zmienne środowiskowe w przypadku wystąpienia konfliktu.

Jak to zrobić?

1. Zmienne środowiskowe są dostępne za pomocą obiektu Pythona `os.environ`.
2. Ponieważ obiekt `environ` jest słownikiem, można wybrać konkretną zmienną do wyświetlenia:

```
>>> import os
>>> print(os.environ["PATH"])
/home/cody/anaconda3/bin:/home/cody/bin:/home/cody/
.local/bin:/usr/local/sbin:/usr/local/bin:/usr
/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

3. Dodanie nowej zmiennej jest proste i sprowadza się do wydania następującego polecenia:

```
>>> os.environ["PYTHONOPTIMIZE"] = "1"
```

Jak to działa?

Dostępna jest ogromna liczba zmiennych środowiskowych charakterystycznych dla Pythona. W tym miejscu wymieniłem tylko wybrane.

- `PYTHONHOME` — używana do zmiany położenia bibliotek standardowych Pythona. Domyślnie te biblioteki znajdują się w */usr/local/lib/<wersja_pythona>*.
- `PYTHONPATH` — modyfikuje domyślną ścieżkę dostępu sprawdzaną podczas wyszukiwania modułów. Format jest taki sam jak w przypadku zmiennej `PATH` powłoki.

  Wprawdzie katalogi są zwykle umieszczane w lokalizacji wskazywanej przez zmienną `PYTHONPATH`, poszczególne elementy mogą prowadzić do archiwów ZIP zawierających czyste moduły Pythona. Moduły te mogą mieć postać kodu źródłowego lub skompilowanych plików Pythona.
- `PYTHONSTARTUP` — powoduje wykonanie poleceń Pythona we wskazanym pliku startowym, zanim pojawi się znak zachęty trybu interaktywnego. Ten plik jest wykonywany w tej samej przestrzeni nazw co znak zachęty trybu interaktywnego, więc obiekty definiowane lub importowane w pliku startowym mogą być używane natywnie, czyli bez konieczności stosowania składni z użyciem kropki.

  Za pomocą tego pliku można zmienić znaki zachęty trybu interaktywnego. W szczególności dotyczy to używanych w trybie interaktywnym znaków zachęty `sys.ps1` (`>>>`) i `sys.ps2` (`...`), które można zastąpić innymi.

  Ten plik pozwala również zmodyfikować zaczep `sys.__interactivehook__`, którego zadaniem jest konfiguracja modułu `rlcompleter` definiującego sposób, w jaki Python będzie uzupełniał prawidłowe identyfikatory i słowa kluczowe modułu GNU `readline`. Innymi słowy ten zaczep jest odpowiedzialny za skonfigurowanie uzupełniania poleceń w Pythonie po naciśnięciu klawisza *Tab* i za zdefiniowanie domyślnego pliku historii poleceń (*~/.python_history*).

- PYTHONOPTIMIZE — jeżeli ma wartość inną niż pusty ciąg tekstowy, działa dokładnie tak samo jak opcja –O, natomiast wartość w postaci ciągu tekstowego przedstawiającego liczbę, np. "2", ma takie samo znaczenie jak kilkukrotne użycie opcji -O.
- PYTHONDEBUG — jeżeli ma wartość inną niż pusty ciąg tekstowy, działa dokładnie tak samo jak opcja –d, natomiast wartość w postaci ciągu tekstowego przedstawiającego liczbę, np. "2", ma takie samo znaczenie jak kilkukrotne użycie opcji -d.
- PYTHONINSPECT — jeżeli ma wartość inną niż pusty ciąg tekstowy, działa dokładnie tak samo jak opcja -i. Ta zmienna środowiskowa może być również modyfikowana za pomocą kodu Pythona przez użycie polecenia os.environ do wymuszenia trybu analizy po zakończeniu działania programu.
- PYTHONUNBUFFERED — jeżeli ma wartość inną niż pusty ciąg tekstowy, działa dokładnie tak samo jak opcja -u.
- PYTHONVERBOSE — jeżeli ma wartość inną niż pusty ciąg tekstowy, działa dokładnie tak samo jak opcja –v, natomiast wartość w postaci liczbowej ma takie samo znaczenie jak kilkukrotne użycie opcji -v.
- PYTHONCASEOK — po zdefiniowaniu tej zmiennej Python zignoruje wielkość liter w poleceniach import. Ta opcja jest dostępna jedynie w systemach Windows i macOS.
- PYTHONDONTWRITEBYTECODE — jeżeli ma wartość inną niż pusty ciąg tekstowy, interpreter nie będzie zapisywał kodu bajtowego (pliki z rozszerzeniem *.pyc*) podczas importowania plików kodu źródłowego. Działanie tej zmiennej jest dokładnie takie samo jak opcji -B.
- PYTHONHASHSEED — gdy ma przypisaną wartość random lub w ogóle nie jest zdefiniowana, losowo wybrana wartość zostanie użyta jako zalążek podczas generowania wartości hash dla obiektów typu str, bytes i datetime. Natomiast wartość w postaci liczby całkowitej spowoduje, że ta liczba zostanie użyta jako zalążek podczas generowania wartości hash. Dzięki temu można zapewnić powtarzalność wyników.
- PYTHONIOENCODING — jeżeli zostanie zdefiniowana przed uruchomieniem interpretera, podane kodowanie zostanie nadpisane dla stdin, stdout i stderr. Składnia ma postać encodingname:errorhandler. Oba elementy składni są opcjonalne i mają takie samo znaczenie jak w przypadku funkcji str.encode().

  Począwszy od wydania Python 3.6, kodowanie wskazywane przez tę zmienną jest ignorowane w systemie Windows podczas użycia konsoli interaktywnej, o ile nie została zdefiniowana zmienna PYTHONLEGACYWINDOWSSTDIO.
- PYTHONNOUSERSITE — po zdefiniowaniu tej zmiennej Python nie dołączy katalogu *site-packages* użytkownika do ścieżki dostępu w sys.path.
- PYTHONUSERBASE — powoduje zdefiniowanie katalogu *base* użytkownika. Katalog ten jest używany jako ścieżka dostępu dla *site-packaes* i instalacji modułu Distutils po wydaniu polecenia python setup.py install -user.
- PYTHONEXECUTABLE — po zdefiniowaniu tej zmiennej element sys.argv[0] będzie miał jej wartość, a nie wartość uzyskaną ze środowiska uruchomieniowego C. Ta zmienna działa jedynie w systemie macOS.

- `PYTHONWARNINGS` — po zdefiniowaniu działa tak samo jak opcja `-W`. Przypisanie jej rozdzielonego przecinkami ciągu tekstowego jest odpowiednikiem użycia wielu opcji `-W`.
- `PYTHONFAULTHANDLER` — jeżeli ma wartość inną niż pusty ciąg tekstowy, w trakcie uruchamiania Pythona zostanie wywołana funkcja `faulthandler.enable()`. Działanie tej zmiennej jest dokładnie takie samo jak użycie opcji `-X faulthandler`.
- `PYTHONTRACEMALLOC` — jeżeli ma wartość inną niż pusty ciąg tekstowy, moduł `tracemalloc` rozpoczyna monitorowanie alokacji pamięci przez Pythona. Podana wartość zmiennej określa liczbę ramek przechowywanych na stosie wywołań.
- `PYTHONASYNCIODEBUG` — jeżeli ma wartość inną niż pusty ciąg tekstowy, włączony zostanie tryb `debug` modułu `asyncio`.
- `PYTHONMALLOC` — definiuje alokację pamięci Pythona, a także instaluje zaczepy przydatne podczas procesu debugowania.

  Oto dostępne komponenty związane z alokacją pamięci:

  - `malloc` — użycie znanej z języka C funkcji `malloc()` we wszystkich domenach.
  - `pymalloc` — użycie `pymalloc` dla domen `PYMEM_DOMAIN_MEM` i `PYMEM_DOMAIN_OBJ`, natomiast znanej z C `malloc()` dla domeny `PYMEM_DOMAIN_RAW`.

  Dostępne zaczepy debugowania są między innymi następujące:

  - `debug` — instaluje zaczepy debugowania na bazie domyślnego sposobu alokacji pamięci.
  - `malloc_debug` — działa tak samo jak przedstawiony wcześniej `malloc`, a dodatkowo instaluje zaczepy debugowania.
  - `pymalloc_debug` — działa tak samo jak przedstawiony wcześniej `pymalloc`, a dodatkowo instaluje zaczepy debugowania.

  Gdy Python jest skompilowany w trybie debugowania, następuje zdefiniowanie `pymalloc_debug` i automatyczne stosowanie zaczepów debugowania. Natomiast kompilacja w trybie wydania powoduje zdefiniowanie zwykłego trybu `pymalloc`. Jeżeli żaden z trybów `pymalloc` nie będzie dostępny, zostaną użyte zwykłe tryby `malloc`.
- `PYTHONMALLOCSTATS` — jeżeli ma wartość inną niż pusty ciąg tekstowy, Python wyświetla dane statystyczne dla alokatora `pymalloc` za każdym razem, gdy będzie tworzony nowy obiekt `pymalloc` i podczas zamykania programu. Natomiast jeśli alokator `pymalloc` jest niedostępny, ta zmienna zostanie zignorowana.
- `PYTHONLEGACYWINDOWSENCODING` — po zdefiniowaniu tej zmiennej domyślne kodowanie systemu plików i tryb błędów otrzymają wartości sprzed wydania Pythona 3.6. Jeżeli używasz wydania 3.6 lub nowszego, kodowanie to `utf-8`, a tryb błędów to `surrogatepass`. Ta możliwość jest dostępna jedynie w systemie Windows.
- `PYTHONLEGACYWINDOWSTDIO` — po zdefiniowaniu tej zmiennej nie będą używane obiekty odczytu i zapisu nowej konsoli, co spowoduje kodowanie znaków Unicode na podstawie strony kodowej aktywnej konsoli, a nie UTF-8. Ta zmienna jest dostępna jedynie w systemie Windows.

- PYTHONTHREADDEBUG — po zdefiniowaniu tej zmiennej Python będzie wyświetlać informacje debugowania dla wątków (definiowana tylko podczas kompilacji Pythona w trybie debugowania).
- PYTHONDUMPREFS — po zdefiniowaniu tej zmiennej Python będzie zapisywać obiekty i liczniki odwołań, które nadal istnieją po zakończeniu działania interpretera (definiowana tylko podczas kompilacji Pythona w trybie debugowania).

Definiowanie skryptu jako wykonywalnego

Normalnie uruchomienie programu Pythona wymaga wydania polecenia `python <nazwa-programu>.py`. Jednak istnieje możliwość zdefiniowania programu Pythona jako samowykonywalnego, co nie będzie wymagało wydania polecenia `python`.

Jak to zrobić?

1. W systemach z rodziny UNIX i Linux umieszczenie w pierwszym wierszu programu polecenia `#!/usr/bin/env python` pozwala na wykonanie programu przez odwołanie się do Pythona za pomocą zmiennej `PATH` użytkownika. Oczywiście przy założeniu, że Python został zainstalowany w lokalizacji wymienionej w ścieżce dostępu zdefiniowanej przez zmienną `PATH`. W przeciwnym razie program trzeba będzie uruchamiać w zwykły sposób.
2. Po dodaniu wymienionego wiersza do programu również plikowi trzeba nadać uprawnienia wykonywania, co wymaga wydania takiego polecenia:

   ```
   $ chmod +x <nazwa-programu>.py
   ```

3. Jeżeli używasz programu terminala wyświetlającego pliki i katalogi w kolorach zależnych od trybu działania, wówczas wydanie polecenia `ls` w katalogu zawierającym plik zmodyfikowany przez polecenie w poprzednim punkcie powinno spowodować, że ten plik będzie w innym kolorze niż pliki niewykonywalne.
4. Aby uruchomić program, wystarczy wydać polecenie `./<nazwa-programu>.py`. To spowoduje uruchomienie programu bez konieczności wcześniejszego podania słowa kluczowego `python`.

Co dalej?

Windows nie ma trybu wykonywania, więc omówione tutaj kroki są niezbędne jedynie w systemach z rodziny UNIX i Linux. Windows automatycznie powiązuje pliki *.py* z *python.exe*, co oznacza, że zostanę uruchomione przez interpreter Pythona. Natomiast pliki z rozszerzeniem *.pyw* zapobiegają wyświetlaniu okna konsoli po uruchomieniu programu Pythona w Windowsie.

Zmiana sposobu uruchamiania interpretera interaktywnego

Jak wspomniałem w recepturze dotyczącej pracy ze zmiennymi środowiskowymi, zmienna `PYTHONSTARTUP` może wskazywać plik zawierający polecenia do wykonania przed uruchomieniem interpretera Pythona. Jest to funkcjonalność, którą można porównać do pliku *.profile* w systemach z rodziny UNIX i Linux.

Ponieważ ten plik startowy jest analizowany tylko podczas użycia trybu interaktywnego, nie trzeba się przejmować próbami przygotowania konfiguracji dla uruchamianych skryptów (z dalszej części rozdziału dowiesz się, jak w skrypcie dodać plik startowy). Polecenia zdefiniowane w tym pliku są wykonywane w tej samej przestrzeni nazw co interpreter interaktywny, więc nie ma konieczności podawania w pełni kwalifikowanych nazw funkcji lub importowania z wykorzystaniem poleceń z kropką. Plik startowy jest odpowiedzialny również za wprowadzanie zmian w znakach zachęty powłoki interaktywnej: `>>>` (`sys.ps1`) i `...` (`sys.ps2`).

Jak to zrobić?

1. Aby odczytać dodatkowy plik startowy znajdujący się w katalogu bieżącym, należy użyć poniższego polecenia, które pokazuje przykład kodu umieszczonego w globalnym pliku startowym (tutaj *read_startup.py*):

```
if os.path.isfile('.pythonrc.py'):
    exec(open('.pythonrc.py').read()
```

2. Wprawdzie plik startowy jest szukany jedynie podczas uruchamiania trybu interaktywnego, ale można się do niego odwołać również z poziomu skryptu. Przykład takiego rozwiązania pokazałem w pliku *startup_script.py*.

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

Zobacz również

Zapoznaj się tez z podrozdziałem „Praca ze zmiennymi środowiskowymi" we wcześniejszej części rozdziału.

Alternatywne implementacje Pythona

Python został przeniesiony do wielu różnych środowisk, takich jak Java i .NET. To oznacza możliwość użycia Pythona w zwykły sposób w tych środowiskach z dostępem do API i kodu związanego z działaniem tych środowisk.

Jython jest używany do integracji z Javą, IronPython do integracji z .NET Framework, Stackless Python charakteryzuje się większą wydajnością w działaniu wątków, natomiast MicroPython jest przeznaczony do użycia wraz z mikrokontrolerami.

Jak to zrobić?

1. Aby użyć Jythona, plik Javy z rozszerzeniem *.jar* dostarcza wykonywalny plik instalacyjny. Podczas instalacji dostępne są dwie opcje.
2. Zwykła instalacja z użyciem graficznego interfejsu użytkownika jest możliwa po zastosowaniu następującego polecenia:

```
$ java -jar jython_installer-2.7.1.jar
```

3. Natomiast w przypadku systemów działających z poziomu powłoki, na przykład serwera, należy skorzystać z następującego polecenia:

```
$ java -jar jython_installer-2.7.1.jar –console
```

4. IronPython można zainstalować za pomocą pliku instalacyjnego Windowsa z rozszerzeniem *.msi*, za pomocą archiwum ZIP lub przez pobranie kodu źródłowego. Instalacja z użyciem *.msi* odbywa się w taki sam sposób jak każdego innego programu w Windowsie. Pliku *.zip* i kodu źródłowego można użyć na platformach innych niż Windows.
5. NuGet to menedżer pakietów dla .NET Framework. IronPythona można zainstalować za pomocą NuGet podobnie jak ma to miejsce w przypadku pakietów narzędzia `pip`. Wymagane są dwa pakiety, ponieważ biblioteka standardowa Pythona znajduje się w oddzielnym pakiecie. W omawianym przykładzie trzeba wydać następujące polecenia NuGet:

```
Install-Package IronPython
Install-Package IronPython.StdLib
```

6. Aby zainstalować Stacklessa, konkretna metoda zależy od używanego systemu operacyjnego. W przypadku systemu z rodziny UNIX i Linux należy zastosować standardowy proces `configure`, `make`, `install`:

```
$ ./configure
$ make
$ make test
$ sudo make install
```

7. W systemie macOS procedura jest nieco bardziej skomplikowana. Python powinien być skonfigurowany wraz z opcją `--enable-framework`, a następnie trzeba wydać polecenie `make frameworkinstall`, aby dokończyć instalację Stacklessa.

8. W systemie Windows jest to jeszcze bardziej skomplikowane. Trzeba zainstalować oprogramowanie Microsoft Visual Studio 2015 wraz z systemem kontroli wersji *Subversion*. Polecenie `build.bat -e` jest używane do skompilowania Stackless. Więcej informacji szczegółowych na ten temat znajdziesz w dokumentacji, do której lektury jeszcze przed rozpoczęciem procesu instalacji gorąco Cię zachęcam.
9. MicroPython jest dostępny w postaci plików *.zip* i *.tar.gz*, a także w repozytorium GitHub. Podczas instalacji wymagana jest pewna liczba opcji i zależności, choć ogólna procedura przedstawia się następująco:

```
$ git submodule update --init
$ cd ports/unix
$ make axtls
$ make
```

Co dalej?

W tym miejscu wspomnę o dostępnych implementacjach Pythona dla różnych platform i frameworków.

- **Jython**. Jest to implementacja Pythona dla wirtualnej maszyny Javy (ang. *Java Virtual Machine* — JVM). Jython pobiera zwykły interpreter Pythona i modyfikuje go w taki sposób, aby móc się z nim komunikować i uruchomić go na platformie Javy. Dlatego obie wymienione platformy zostają zintegrowane i możliwe staje się użycie w programach Pythona bibliotek i aplikacji Javy.

  Wprawdzie członkowie projektu Jython dołożyli wszelkich starań, aby wszystkie moduły Pythona działały w wirtualnej maszynie Javy, mimo wszystko można dostrzec pewne różnice. Największa polega na tym, że rozszerzenia stworzone w języku C nie będą działały w Jythonie; jednak większość modułów Pythona będzie działać prawidłowo w Jythonie bez modyfikacji. Rozszerzenia napisane w języku C należy ponownie przepisać w języku Java, aby zapewnić ich prawidłowe działanie.

  Kod Jythona działa doskonale w środowisku Javy, natomiast użycie standardowego kodu CPython (domyślne środowisko Pythona) może powodować problemy. Mimo wszystko kod Jythona będzie normalnie działał bez żadnych przeszkód w środowisku CPython, o ile nie wykorzystuje jakiegokolwiek sposobu integracji z Javą.
- **IronPython**. Jest to Python dla opracowanej przez Microsoft platformy .NET Framework. Program IronPython może wykorzystywać możliwości oferowane przez .NET Framework, a także zwykłe biblioteki Pythona. Ponadto inne języki .NET, na przykład C#, mogą implementować kod IronPythona.

  Z powodu wspomnianej funkcjonalności .NET IronPython to doskonałe narzędzie dla programistów Windowsa lub programistów systemów UNIX i Linux używających środowiska Mono. Zwykłe projekty Pythona mogą być opracowywane w IronPythonie. Programiści zyskują również możliwość użycia Pythona zamiast innych języków skryptowych, takich jak VBScript i PowerShell. Opracowane przez Microsoft

środowisko programistyczne, Visual Studio, ma wtyczkę o nazwie Python Tools, która pozwala wykorzystać pełną funkcjonalność Visual Studio podczas tworzenia kodu Pythona.

> IronPython jest dostępny jedynie dla wydania Python 2.7. Nie został jeszcze przeniesiony dla wydania Python 3. Zastosowanie narzędzi takich jak `3to2` nie gwarantuje poprawności działania ze względu na różnice w naturze wydań Pythona 2 i 3.

- **Stackless Python**. Jest to usprawniona wersja Pythona koncentrująca się na poprawieniu programowania opartego na wątkach bez wprowadzania zwykłych komplikacji związanych z wątkami Pythona. Wykorzystując mikrowątki, Stackless ma na celu udoskonalenie struktury programu, znaczne poprawienie czytelności kodu wielowątkowego i zwiększenie produktywności programisty.

  Te usprawnienia zostały osiągnięte przez uniknięcie zwykłego stosu wywołań C i wykorzystanie własnego stosu zarządzanego przez interpreter. Mikrowątki obsługują wykonywanie zadań programu w tym samym procesorze. W ten sposób otrzymujemy alternatywę dla tradycyjnych metod programowania asynchronicznego. To eliminuje obciążenie związane z wielowątkowością programów działających w jednym procesorze, ponieważ nie ma opóźnienia związanego z przełączaniem między trybem użytkownika i jądra.

  Mikrowątki wykorzystują tak zwane tasklety do przedstawiania małych zadań w wątku Pythona i mogą być stosowane zamiast tradycyjnych wątków lub procesów. Dwukierunkowa komunikacja między mikrowątkami jest obsługiwana przez kanały, a harmonogram zadań jest definiowany w konfiguracji rotacyjnej. Dlatego harmonogram zadań taskletów może współpracować z innymi komponentami systemu lub rezerwować zasoby na wyłączność. Wreszcie serializacja dostępna za pomocą modułu `pickle` Pythona pozwala na opóźnione wznowienie mikrowątku.

  Jedną z wad Stacklessa jest to, że choć mikrowątki stanowią usprawnienie względem zwykłych wątków Pythona, to jednak nie eliminują globalnej blokady interpretera. Ponadto tasklety znajdują się w pojedynczym wątku, wielowątkowość i wieloprocesowość nie jest obsługiwana.

  Innymi słowy nie mamy do czynienia z prawdziwym przetwarzaniem równoległym, lecz jedynie z wielozadaniowością w ramach pojedynczego procesora współdzielonego między tasklety. To jest taka sama funkcjonalność jak wielowątkowość oferowana przez Pythona. Aby wykorzystać współbieżność między wieloma procesorami, komunikacja międzyprocesowa musi być skonfigurowana ponad procesami Stacklessa.

  Wreszcie z powodu zmian wprowadzonych w kodzie źródłowym Pythona, aby zaimplementować mikrowątki, Stacklessa nie można zainstalować na bazie istniejącej instalacji Pythona. Dlatego pełna instalacja Stacklessa musi pozostać oddzielna od wszelkich innych dystrybucji Pythona.

- **MicroPython**. Jest to okrojona wersja Pythona 3.4 zaprojektowana do użycia z mikrokontrolerami i systemami osadzonymi. Wprawdzie MicroPython oferuje większość funkcji standardowego Pythona, ale zawiera drobne zmiany w języku,

aby go lepiej przystosować do mikrokontrolerów. Kluczową cechą MicroPythona jest to, że można go uruchomić, mając do dyspozycji jedynie 16 KB pamięci RAM, a jego kod źródłowy zajmuje tylko 256 KB miejsca na dysku.

Dostępny jest unikatowy mikrokontroler, pyboard, zaprojektowany do użycia wraz z MicroPythonem. Ten mikrokontroler jest podobny do Raspberry Pi, przy czym jest jeszcze mniejszy. Mimo wszystko wciąż ma 30 pinów GPIO, cztery wbudowane diody LED, przyśpieszeniomierz i wiele innych komponentów. Ponieważ został zaprojektowany do użycia z MicroPythonem, w zasadzie otrzymujesz Python OS przeznaczony do działania w mikrokontrolerze.

Instalowanie Pythona w Windowsie

W porównaniu do systemów z rodziny UNIX i Linux dostarczanych standardowo wraz z zainstalowanym Pythonem system operacyjny Windows nie zawiera domyślnie Pythona. Na szczęście można pobrać pakiet instalacyjny w formacie MSI przygotowany dla różnych wersji Windowsa. Tak zainstalowany Python jest przeznaczony do użycia przez jednego użytkownika, a nie wszystkich użytkowników danego komputera. Jednak w trakcie instalacji można wybrać opcję pozwalającą wszystkim użytkownikom komputera uzyskać dostęp do Pythona.

Zaczynamy

Ponieważ Python zawiera przygotowany dla różnych systemów operacyjnych kod charakterystyczny dla platformy (aby zminimalizować ilość zbędnego kodu), obsługuje systemy operacyjne Windows dopóty, dopóki są obsługiwane przez firmę Microsoft. To obejmuje również wydłużoną obsługę i dlatego produkty uznawane przez Microsoft za nieobsługiwane nie będą także obsługiwane przez Pythona.

Użytkownik systemu Windows XP lub starszego nie może zainstalować żadnej nowszej wersji Pythona niż 3.4. Zgodnie z informacjami zamieszczonymi w dokumentacji Pythona w systemie Windows Vista można zainstalować Pythona w wersji 3.6 lub nowszej. Ponieważ oficjalna obsługa systemu Windows Vista przez Microsoft zakończyła się w roku 2017, także kolejne wydania Pythona nie będą działały w tym systemie operacyjnym. Ponadto trzeba znać architekturę używanego procesora: 32- lub 64-bitowy. Wprawdzie oprogramowanie 32-bitowe działa w systemie 64-bitowym, ale na odwrót już nie.

Poza tym do dyspozycji są dwa rodzaje plików instalacyjnych — do instalacji offline i online. W przypadku tej pierwszej pakiet instalacyjny zawiera wszystkie komponenty niezbędne do przeprowadzenia instalacji standardowej, a dostęp do internetu jest potrzebny tylko do pobrania funkcji opcjonalnych. Z kolei pakiet instalacji internetowej jest mniejszy od wersji offline i umożliwia użytkownikowi wybór funkcji do zainstalowania, które następnie zostaną pobrane z internetu.

Jak to zrobić?

1. Pierwsze uruchomienie pakietu instalacyjnego w Windowsie pozwala na wybór jednej z dwóch opcji: instalacji domyślnej lub niestandardowej. Wybierz domyślną, gdy występują następujące czynniki:
   - Instalację przeprowadzasz tylko dla siebie i inni użytkownicy komputera nie muszą mieć dostępu do Pythona.
   - Chcesz zainstalować jedynie bibliotekę standardową Pythona, zestaw testowy, narzędzie `pip` i program uruchamiający Pythona w systemie Windows.
   - Związane z Pythonem skróty są dostępne jedynie dla bieżącego użytkownika.
2. Zdecyduj się na instalację niestandardową, jeżeli chcesz zachować większą kontrolę nad tym procesem. W szczególności dotyczy to sytuacji, gdy występują następujące czynniki:
   - Chcesz zainstalować konkretne składniki Pythona.
   - Chcesz zmienić miejsce instalacji Pythona.
   - Chcesz zainstalować symbole debugowania lub pliki binarne.
   - Chcesz przeprowadzić instalację dla wszystkich użytkowników systemu.
   - Chcesz wstępnie skompilować bibliotekę standardową na kod bajtowy.
3. Instalacja niestandardowa wymaga uprawnień użytkownika administracyjnego. Graficzny interfejs użytkownika to standardowy sposób instalacji Pythona za pomocą kreatora pozwalającego przejść przez cały proces. Ewentualnie można użyć skryptów wiersza polecenia do zautomatyzowania instalacji w wielu komputerach bez jakiejkolwiek reakcji ze strony użytkownika. W przypadku instalacji z poziomu wiersza polecenia podczas uruchamiania pliku *.exe* dostępnych jest kilka opcji:

```
python-3.6.0.exe /quiet # GUI nie będzie wyświetlone, nastąpi instalacja podstawowa Pythona
... /passive # Pominięcie użytkownika podczas instalacji, ale wyświetlenie
             #  informacji o postępie i błędach
... /uninstall # Natychmiastowe rozpoczęcie usuwania Pythona, wyświetlenie pytania
               # potwierdzającego operację
```

Stosowanie programu uruchamiającego Pythona w Windowsie

Począwszy od wydania Pythona 3.3, wraz z językiem zostaje domyślnie zainstalowany program umożliwiający uruchomienie Pythona. Ten program pozwala skryptom Pythona lub wiersza polecenia Windowsa na określenie konkretnej wersji Pythona oraz na jej odszukanie i uruchomienie.

Gdy masz zainstalowane wydanie Pythona 3.3 lub nowsze, program uruchamiający jest zgodny ze wszystkimi wersjami Pythona. Wybierze najbardziej odpowiednią wersję języka dla skryptu i zastosuj instalację Pythona przeprowadzaną dla danego użytkownika, a nie dla wszystkich.

Jak to zrobić?

1. Aby sprawdzić, czy program uruchamiający Pythona został zainstalowany, w oknie wiersza polecenia Windowsa wpisz `py`. Jeżeli ów program jest zainstalowany, nastąpi uruchomienie najnowszej dostępnej wersji Pythona.
2. Natomiast w przeciwnym razie otrzymasz następujący komunikat błędu:

   ```
   Nazwa 'py' nie jest rozpoznawana jako polecenie wewnętrzne lub zewnętrzne,
   program wykonywalny lub plik wsadowy.
   ```

3. Przyjmując założenie, że zainstalowana jest inna wersja Pythona, możesz ją wskazać za pomocą opcji `-`:

   ```
   py -2.6 # Uruchomienie Pythona w wersji 2.6
   py -2 # Uruchomienie najnowszej dostępnej wersji Pythona 2
   ```

4. Jeżeli używasz wirtualnego środowiska Pythona, a program uruchamiający Pythona nie otrzymał żadnej opcji wskazującej konkretną wersję języka, wówczas zostanie użyty interpreter środowiska wirtualnego, a nie systemowy. Aby móc użyć interpretera systemowego, środowisko wirtualne musi zostać najpierw dezaktywowane lub należy wyraźnie podać systemową wersję Pythona.
5. Program uruchamiający pozwala na użycie w Windowsie wiersza shebang (`#!`), który jest dość powszechnie stosowany w systemach z rodziny UNIX i Linux. Wprawdzie dostępnych jest wiele wariantów ścieżki środowiskowej Pythona, ale warto w tym miejscu wspomnieć, że jedna z najczęściej stosowanych (`/usr/bin/env python`) działa w Windowsie w dokładnie taki sam sposób jak w systemach UNIX i Linux. To oznacza sprawdzenie przez Windows zmiennej systemowej `PATH` w poszukiwaniu pliku wykonywalnego Pythona, a dopiero później zainstalowanie interpreterów. Dokładnie tak samo ten proces przebiega w systemach z rodziny UNIX i Linux.
6. Wiersz shebang może zawierać opcje interpretera Pythona, jakby zostały podane w wierszu polecenia. Dlatego też `#! /usr/bin/python -v` spowoduje wyświetlenie informacji o wersji używanego interpretera Pythona. Takie samo zachowanie otrzymujemy po wydaniu polecenia `python -v` w wierszu polecenia.

Osadzanie Pythona w innych aplikacjach

Osadzona dystrybucja Pythona to plik w formacie ZIP zawierający minimalną wersję interpretera Pythona. Jego przeznaczeniem jest dostarczanie środowiska Pythona innym programom, a nie bezpośrednie wykorzystywanie przez użytkowników końcowych.

Po wypakowaniu z archiwum środowisko to jest w zasadzie odizolowane od systemu operacyjnego, czyli zawiera wszystkie niezbędne składniki. Biblioteka standardowa jest wstępnie skompilowana na postać kodu bajtowego i dołączone są wszystkie powiązane z Pythonem pliki w formatach *.exe* i *.dll*. Natomiast nie zostały dołączone narzędzie `pip`, dokumentacja i środowi-

sko Tcl/Tk. Z powodu braku środowiska Tcl/Tk niedostępne do użycia jest środowisko programistyczne IDLE i powiązane z nim pliki Tkinter.

Wraz z osadzoną dystrybucją Pythona nie jest dostarczane środowisko uruchomieniowe Microsoft C. Wprawdzie jest ono często instalowane w systemie przez inne oprogramowanie lub usługę Windows Update, ostatecznie do programu instalacyjnego należy zapewnienie, że to środowisko uruchomieniowe będzie dostępne do użycia przez Pythona.

Niezbędne pakiety Pythona opracowane przez firmy trzecie muszą być dostarczone przez program instalacyjny, poza samym osadzonym środowiskiem Pythona. Ponieważ narzędzie `pip` pozostaje niedostępne, odpowiednie pakiety powinny być dostarczane wraz z całymi aplikacjami, aby były uaktualniane razem z tymi programami.

Jak to zrobić?

1. Utwórz aplikację Pythona w zwykły sposób.
2. Jeżeli użycie Pythona nie powinno być oczywiste dla użytkownika końcowego, wówczas należy dołączyć dostosowany do własnych potrzeb program uruchamiający Pythona. Jego działanie polega po prostu na wywołaniu modułu `__main__` Pythona za pomocą na stałe zdefiniowanego polecenia.

   W przypadku korzystania z własnego programu uruchamiającego pakiety Pythona mogą zostać umieszczone w dowolnym położeniu systemu plików, ponieważ program uruchomieniowy można przygotować w taki sposób, aby wskazywał konkretną ścieżkę wyszukiwania podczas uruchamiania programu.
3. Jeżeli użycie Pythona nie musi być ukryte, wystarczy przygotowanie zwykłego pliku wsadowego lub skrótu pozwalającego na bezpośrednie wywołanie `python.exe` wraz z niezbędnymi argumentami. Po zastosowaniu podanego podejścia użycie Pythona będzie oczywiste, ponieważ nie zostanie wykorzystana prawdziwa nazwa programu, lecz zamiast niej nazwa interpretera Pythona. Dlatego też użytkownikowi końcowemu trudno będzie zidentyfikować określony program wśród innych uruchomionych procesów Pythona.

   W przypadku stosowania tej metody zaleca się instalowanie pakietów Pythona w podkatalogach znajdujących się w tym samym katalogu, w którym jest program wykonywalny interpretera Pythona. Dzięki temu pakiety zostaną uwzględnione w ścieżce dostępu wskazywanej przez zmienną systemową `PATH`, ponieważ są podkatalogami programu głównego.
4. Alternatywne zastosowanie osadzonego Pythona polega na jego użyciu w charakterze języka dostarczającego możliwości skryptowe dla kodu natywnego, na przykład programów C++. W takim przypadku większość oprogramowania jest utworzona w języku innym niż Python, a wywołanie kodu Pythona odbywa się za pomocą *python.exe* lub *python3.dll*. Niezależnie od tego Python zostaje wypakowany z osadzonej dystrybucji do podkatalogu, co pozwala wywołać interpreter Pythona.

   Pakiety mogą być instalowane w katalogu systemu plików, ponieważ ich ścieżki dostępu można podać w kodzie przed skonfigurowaniem interpretera Pythona.

5. Na rysunku 2.1 pokazałem przykład osadzenia Pythona na bardzo głębokim poziomie. Ten przykład pochodzi ze strony *https://docs.python.org/3/extending/embedding.html*.

```
#include <Python.h>

int
main(int argc, char *argv[])
{
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
        fprintf(stderr, "Fatal error: cannot decode argv[0]\n");
        exit(1);
    }
    Py_SetProgramName(program);  /* optional but recommended */
    Py_Initialize();
    PyRun_SimpleString("from time import time,ctime\n"
                       "print('Today is', ctime(time()))\n");
    if (Py_FinalizeEx() < 0) {
        exit(120);
    }
    PyMem_RawFree(program);
    return 0;
}
```

Rysunek 2.1. Przykład osadzenia Pythona w kodzie C

Jak to działa?

W przedstawionym na rysunku 2.1 kodzie język C został użyty w celu uzyskania dostępu do Pythona. Ponieważ to nie jest książka poświęcona programowaniu w języku C, nie będę dokładnie omawiał tego kodu. Oto krótkie wyjaśnienie sposobu jego działania:

1. Python zostaje zaimportowany do kodu jako plik nagłówkowy.
2. Kod w języku C otrzymuje ścieżki dostępu do bibliotek Pythona.
3. Inicjalizacja interpretera Pythona.
4. Skrypt Pythona na stałe umieszczony w kodzie C zostaje przetworzony.
5. Zakończenie działania interpretera Pythona.
6. Zakończenie działania programu w języku C.

W rzeczywistej praktyce przeznaczony do uruchomienia program Pythona będzie raczej pobierany z pliku, a nie osadzony w kodzie języka C. To zdejmuje z programisty konieczność alokowania pamięci i wczytania zawartości pliku.

Zastosowanie alternatywnej powłoki Pythona — IPython

Wprawdzie jest użyteczna, ale domyślna powłoka interpretera Pythona ma poważne ograniczenia w porównaniu do możliwości obecnych komputerów. W przypadku początkujących

programistów wadą interaktywnego interpretera Pythona jest brak między innymi obsługi podświetlania składni i automatycznego stosowania wcięć.

IPython to jeden z najpopularniejszych zamienników powłoki interaktywnej Pythona. Oto wybrane funkcje oferowane przez IPythona, a niedostępne w standardowej powłoce Pythona:

- rozbudowana introspekcja obiektu pozwalająca przeglądać komentarze typu docstring, kod źródłowy i inne obiekty dostępne dla interpretera;
- trwała historia danych wejściowych;
- buforowanie wyników danych wyjściowych;
- rozszerzalne uzupełnianie klawiszem *Tab* wraz z obsługą zmiennych, słów kluczowych, funkcji i nazw plików;
- poprzedzone znakiem `%` polecenia *magiczne* umożliwiające kontrolowanie środowiska i interakcję z systemem operacyjnym;
- rozbudowana konfiguracja systemu;
- rejestrowanie danych sesji i jej ponowne wczytanie;
- osadzanie danych w graficznym interfejsie użytkownika i programach Pythona;
- zintegrowany dostęp do debugera i profilera;
- edycja w wielu wierszach;
- podświetlanie składni.

Wraz z IPythonem otrzymujemy komponent Jupyter, który pozwala tworzyć notatniki. Początkowo notatnik był integralną częścią IPythona, ale z czasem stał się oddzielnym projektem, co pozwoliło zaoferować potężne możliwości notatników także innym językom programowania. Dlatego też IPython i Jupyter można stosować oddzielnie, zawierają one różne frontendy i backendy, dostarczając niezbędną funkcjonalność.

Notatnik Jupytera to oparta na przeglądarce WWW aplikacja, której można używać podczas pracy nad programem i dokumentacją oraz wykonywać w niej kod, wyświetlać wygenerowane dane wyjściowe w postaci tekstu, obrazów i innego rodzaju danych.

Notatnik Jupytera jako aplikacja internetowa oferuje następujące funkcje:

- edycja bezpośrednio w przeglądarce WWW wraz z obsługą podświetlania składni, automatycznego stosowania wcięć, introspekcji i uzupełniania klawiszem *Tab*;
- wykonywanie kodu bezpośrednio w przeglądarce WWW i dołączanie wyników do kodu źródłowego;
- możliwość wyświetlania bogatych mediów, między innymi w formatach HTML, LaTeX, PNG i SVG;
- możliwość edycji tekstu w formacie Markdown;
- stosowanie notacji matematycznych za pomocą oprogramowania LaTeX.

Innym pakietem zaliczającym się do rodziny IPython jest IPython Paraller, znany również jako `ipyparaller`. Ten pakiet obsługuje następujące modele programowania równoległego:

- SPMD (ang. *Single Program, Multiple Data*);
- MPMD (ang. *Multiple Programs, Multiple Data*);
- przekazywanie wiadomości za pomocą MPI;
- zespół zadań;
- dane równoległe,
- połączenie wcześniej wymienionych modeli;
- zastosowanie niestandardowych rozwiązań.

Największą zaletą użycia `ipyparallel` jest umożliwienie opracowania aplikacji równolegle przetwarzających dane, a także przetestowania tych programów i ich interaktywnego użycia. Standardowo równoległość odbywa się przez utworzenie kodu i jego uruchomienie, aby poznać wyniki. Sesje interaktywnego kodu mogą znacznie zwiększyć szybkość tworzenia projektu, ponieważ pokazują, czy warto dalej poświęcać czas na rozwój danego algorytmu.

Zaczynamy

Powłokę IPython można zainstalować za pomocą narzędzia `pip` , choć wcześniej może okazać się konieczne zainstalowanie jeszcze pakietu `setuptools`.

```
$ pip install ipython
```

Powłoka IPython jest dostępna również jako część pakietu Anaconda, czyli dystrybucji Pythona powstałej pod kątem nauk o danych i uczenia maszynowego. Poza powłoką IPython dystrybucja Anaconda oferuje ogromną liczbę pakietów dla nauk ścisłych, nauk o danych i sztucznej inteligencji.

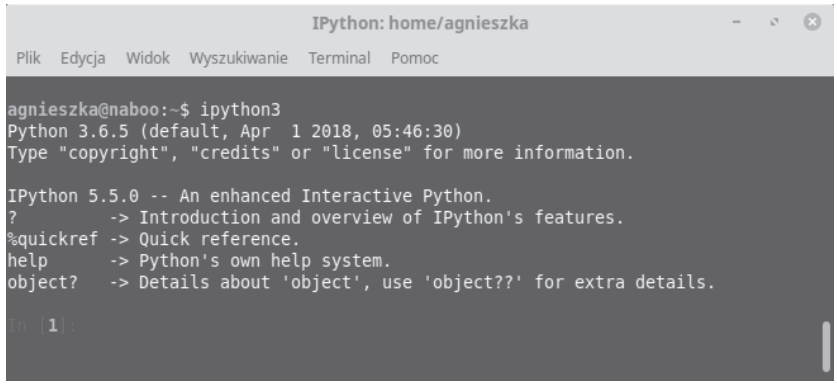
Jeżeli nie używasz przygotowanego środowiska takiego jak Anaconda, do połączenia funkcjonalności Jupyter i IPythona wydaj następujące polecenia:

```
$ python -m pip install ipykernel
$ python -m ipykernel install [--user] [--name <machine-readable-name>] [--
display-name <"Nazwa przyjazna dla użytkownika">]
```

- `user` określa instalację dla bieżącego użytkownika, a nie globalną dla całego systemu.
- `name` nadaje nazwę jądru IPythona. Jest to niezbędne tylko wtedy, gdy jednocześnie będzie działało wiele jąder IPythona.
- `display-name` to nazwa danego jądra IPythona. Możliwość nadawania nazw jest najbardziej użyteczna po zainstalowaniu wielu jąder IPythona.

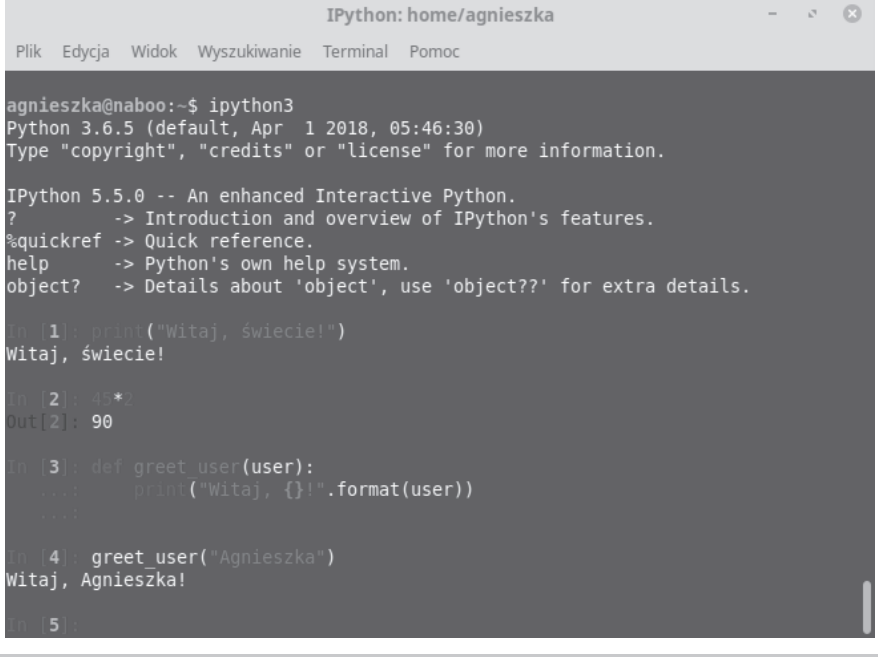
Jak to zrobić?

1. Aby rozpocząć sesję interaktywną w powłoce IPython, należy wydać polecenie `ipython` (zobacz rysunek 2.2). Jeżeli masz zainstalowane różne wersje Pythona, musisz wydać polecenie `ipython3`.



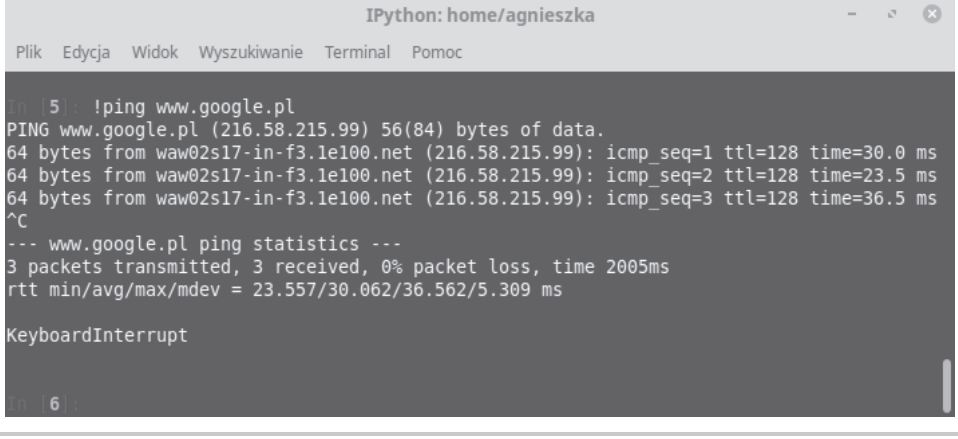
Rysunek 2.2. Uruchomienie sesji interaktywnej IPythona

2. Zwróć uwagę na znak zachęty, którym jest `In [N]:` zamiast `>>>`. Liczba `N` odwołuje się do numeru polecenia w historii IPythona, można jej użyć do ponownego wykonania polecenia, podobnie jak ma to miejsce w powłoce systemowej bash.
3. Interpreter IPython działa podobnie jak standardowy, choć oferuje znacznie większą funkcjonalność. Statyczny tekst w przykładach nie oddaje pełni możliwości powłoki, ponieważ podświetlanie składni, automatyczne stosowanie wcięć i uzupełnianie klawiszem *Tab* działają w czasie rzeczywistym. Na rysunku 2.3 pokazałem przykład kilku prostych poleceń wydanych w powłoce interpretera IPython.



Rysunek 2.3. Przykłady poleceń wydanych w powłoce interpretera IPython

4. Zwróć uwagę, że wynik wykonania polecenia jest wyświetlany w wierszu rozpoczynającym się od `Out [N]:`. Podobnie jak w przypadku `In [N]:`, także numer wiersza danych wyjściowych można wywołać w dowolnym miejscu kodu.
5. Aby dowiedzieć się więcej na temat dowolnego obiektu, należy użyć znaku zapytania, na przykład `<nazwa_obiektu>?`. Więcej informacji otrzymasz po użyciu dwóch znaków zapytania: `<nazwa_obiektu>??`.
6. Funkcje magiczne są unikatową cechą IPythona. To w zasadzie są wbudowane skróty umożliwiające kontrolowanie sposobu działania IPythona, jak również dostarczanie funkcji systemowych, podobnie jak w przypadku dostępu do poleceń powłoki bash.
   - Wiersze magiczne rozpoczynają się od znaku `%` i działają podobnie jak polecenia powłoki bash: argument jest przekazywany funkcji magicznej. Wszystko w wierszu poza samym wywołaniem funkcji jest uznawane za argument.
   - Egzemplarz polecenia magicznego zwraca wynik, podobnie jak zwykła funkcja. Dlatego wynik może być przypisany zmiennej.
   - Egzemplarz bloku magicznego jest poprzedzony znakami `%%`. Działa podobnie jak wiersz magiczny z wyjątkiem tego, że argument może się składać z kilku wierszy, a nie tylko jednego.
   - Funkcje magiczne wpływają na działanie powłoki IPythona, pracę z kodem źródłowym i zapewniają funkcje narzędziowe ogólnego przeznaczenia.
7. IPython oferuje wbudowany dziennik historii poleceń zawierający zarówno same polecenia, jak i wyniki ich wykonania. Funkcja magiczna `%history` wyświetla historię poleceń. Do pracy z tą historią można wykorzystać jeszcze inne funkcje magiczne, które pozwalają między innymi na ponowne wykonywanie wcześniejszych poleceń lub ich kopiowanie do bieżącej sesji.
8. Współpraca z systemem operacyjnym jest możliwa dzięki poprzedzeniu polecenia prefiksem `!`. Dlatego aby wykorzystać powłokę bash w sesji IPythona bez jej opuszczania lub otwierania nowego okna powłoki, należy wydać polecenie `!<polecenie-powłoki-bash>`, na przykład `!ping`, jak pokazałem na rysunku 2.4.



Rysunek 2.4. Wykonanie polecenia powłoki bash w powłoce IPythona

9. IPython pozwala na obsługę danych wyjściowych w postaci bogatych mediów, o ile powłoka IPython jest używana jako jądro dla innego frontendu oprogramowania. Możliwe jest na przykład wyświetlanie wykresów za pomocą biblioteki matplotlib. To okazuje się szczególnie użyteczne podczas pracy z notatnikami Jupytera, ponieważ w oknie przeglądarki WWW mamy wówczas kod źródłowy i wykres wynikowy.
10. Obsługiwane jest również interaktywne tworzenie graficznego interfejsu użytkownika. W takim przypadku IPython czeka na dane wejściowe z pętli zdarzeń pakietu narzędziowego graficznego interfejsu użytkownika. Aby skorzystać z tej możliwości, należy użyć funkcji magicznej `%gui <nazwa_pakietu>`. Obsługiwane frameworki GUI to wxPython, PyQT, PyGTK i Tk.
11. IPython ma możliwość interaktywnego uruchamiania skryptów, na przykład jak w przypadku prezentacji. Dodanie kilku znaczników do komentarzy osadzonych w kodzie źródłowym dzieli go na osobne bloki, z których każdy działa oddzielnie. IPython wyświetli blok przed wykonaniem kodu, a następnie powróci do powłoki interaktywnej, pozwalając na wykorzystanie wygenerowanego wyniku.
12. Możliwe jest dodanie w innych programach obsługi osadzonego IPythona, podobnie jak ma to miejsce w przypadku osadzonej dystrybucji Pythona.

Co dalej?

Począwszy od wydania IPython 6.0, nie są obsługiwane wersje interpretera Pythona wcześniejsze niż 3.3. Jeżeli chcesz skorzystać ze starszych wersji Pythona, musisz użyć IPython 5 LTS.

Zastosowanie alternatywnej powłoki Pythona — bpython

Powłoka bpython powstała dla programistów, którzy chcą otrzymać większą funkcjonalność w środowisku Pythona bez obciążenia związanego z IPythona lub koniecznością jego poznania. Dlatego bpython oferuje wiele funkcji znanych ze środowiska IDE, ale w znacznie lżejszym pakiecie. Oto kilka wybranych funkcji bpythona:

- podświetlanie składni;
- sugestie automatycznego uzupełniania podczas tworzenia kodu;
- sugerowane parametry podczas uzupełniania funkcji;
- funkcja przewijania kodu pozbywająca się ostatniego wiersza i ponownie wykonująca cały kod źródłowy;
- integracja z serwisem pastebin, co pozwala na umieszczenie w tym serwisie widocznego kodu.

Zaczynamy

Aby móc używać powłoki bpython, trzeba oprócz samego pakietu powłoki zainstalować jeszcze kilka innych pakietów:

- Pygments,
- requests,
- Sphinx (opcjonalny, tylko dla dokumentacji),
- mock (opcjonalny, tylko dla zestawu testów),
- babel (opcjonalny, tylko na potrzeby internacjonalizacji),
- curtsies,
- greenlet,
- urwid (opcjonalny, tylko dla bpython-urwind),
- requests[security] dla wersji Pythona wcześniejszych niż 2.7.7.

Jak to zrobić?

1. Utwórz środowisko wirtualne dla projektu, na przykład wydając następujące polecenia:

```
$ virtualenv bpython-dev # Ustalenie użytej wersji Pythona
$ source bpython-dev/bin/activate
# Niezbędne za każdym razem, gdy chcesz pracować z powłoką bpython
```

2. W swoim systemie sklonuj repozytorium `bpython` z serwisu GitHub:

```
$ git clone git@github.com:<nazwa_użytkownika_github>/bpython/bpython.git
```

3. Zainstaluj bypythona wraz z zależnościami:

```
$ cd bpython
$ pip install -e . # Instalacja bpythona i niezbędnych zależności
$ pip install watchdog urwid # Instalacja zależności opcjonalnych
$ pip install sphinx mock nose # Instalacja zależności programistycznych
$ bpython # Uruchomienie powłoki bpython
```

4. Jako alternatywa dla instalacji za pomocą narzędzia `pip` w dystrybucji systemu z rodziny UNIX i Linux można użyć systemowego menedżera pakietów. System prawdopodobnie ma już niezbędne pliki. Operacja wyszukania z użyciem `apt` i nazwy `python-<pakiet>` pokaże, czy dany pakiet jest dostępny. Aby zainstalować dany pakiet, należy wydać polecenie podobne do następującego:

```
$ sudo apt install python[3]-<pakiet>
```

   Liczba 3 jest opcjonalna, jeśli instalujesz pakiet dla wydania Python 2, ale niezbędna, gdy chcesz otrzymać pakiet dla wydania Python 3.

   Powłokę bpython można zainstalować również za pomocą `easyinstall`, `pip`, a także zwykłego `apt install`.

5. Dokumentacja powłoki bpython jest dostarczana wraz z repozytorium. Utworzenie lokalnej kopii dokumentacji wymaga zainstalowanego narzędzia `sphinx` i wydania następującego polecenia:

```
$ make -C doc/sphinx html
```

   Po wygenerowaniu dokumentacji masz do niej dostęp, otwierając w przeglądarce WWW plik *doc/sphinx/build/html/index.html*.

6. W pliku konfiguracyjnym powłoki bpython mamy dużą liczbę opcji. (Domyślnie ten plik znajduje się w katalogu *~/.config/bpython/config*). Dostępne są opcje pozwalające zdefiniować automatyczne uzupełnianie, motyw kolorów, mapowanie klawiatury itd.

7. Dostępna jest również konfiguracja motywu, co odbywa się za pomocą opcji `color_scheme`. Motyw jest używany do kontrolowania podświetlenia składni, a także w samej powłoce Pythona.

Co dalej?

W chwili powstawania tej książki aktualna wersja powłoki bpython to 0.17. Wprawdzie ta powłoka jest sklasyfikowana jako betaware, ale przekonałem się, że działa doskonale w codziennej pracy. Pomoc techniczna jest dostępna za pomocą kanału IRC, grupy dyskusyjnej Google Groups oraz poprzez różne media społecznościowe. Więcej informacji na temat tej powłoki, w tym między innymi zrzuty ekranu, znajdziesz w witrynie internetowej projektu.

Zastosowanie alternatywnej powłoki Pythona — DreamPie

Kontynuując nutę usprawnienia zwykłego Pythona, DreamPie oferuje kilka nowych pomysłów dla alternatywnej powłoki. Funkcjonalność oferowana przez DreamPie to między innymi:

- podział powłoki interaktywnej na okno historii i kodu (podobnie jak w przypadku IPythona okno historii zawiera listę wcześniej wykonanych poleceń i wygenerowanych przez nie danych wyjściowych, natomiast okno kodu zawiera aktualnie edytowany kod; różnica w przypadku okna kodu polega na tym, że działa jak zwykły edytor tekstu, pozwalając tworzyć dowolną ilość kodu przed jego wykonaniem);
- polecenie umożliwiające skopiowanie żądanego kodu i wklejenie go w pliku z zachowaniem wcięć;
- automatyczne uzupełnianie atrybutów i nazw plików;
- introspekcja kodu, wyświetlanie argumentów funkcji i dokumentacji;
- historia sesji może być zapisana w pliku HTML, który później można wczytać do powłoki DreamPie i ponownie wykorzystać;
- automatyczne dodawanie nawiasów i znaków cytowania po funkcjach i metodach;
- integracja z biblioteką matplotlib pozwalająca tworzyć interaktywne wykresy;

- obsługa niemal wszystkich implementacji Pythona, między innymi Jythona, IronPythona i PyPy;
- obsługa niezależnie od platformy.

Zaczynamy

Przed instalacją DreamPy konieczne jest zainstalowanie Pythona 2.7, PyGTK i `pyqtsourceview` (trzeba użyć Pythona 2.7, ponieważ PyGTK nie zapewnia obsługi Pythona 3).

Jak to zrobić?

1. Zalecanym sposobem pobrania powłoki DreamPy jest sklonowanie repozytorium GitHub:

```
$ git clone https://github.com/noamraph/dreampie.git
```

2. Ewentualnie można skorzystać z plików binarnych dostępnych dla systemów Windows, macOS i Linux (odpowiednie łącza znajdziesz na stronie DreamPie: *http://www.dreampie.org/download.html*). Te pliki są uaktualniane wolniej niż repozytorium GitHub i dlatego oferowane przez nie wersje są mniej stabilne.

Co dalej?

Nie udało mi się zmusić powłoki DreamPie do pracy w systemie Xubuntu 16.04 z Pythonem 2.7.11. Komunikat błędu wskazywał na niezaimportowany moduł Glib Object System (`gobject`). Nawet próba ręcznej instalacji tego pakietu nie pozwoliła na zakończoną sukcesem instalację powłoki DreamPie.

Ostatnia aktualizacja witryny internetowej DreamPie miała miejsce w 2012 roku. Nie istnieje dokumentacja przedstawiająca sposób użycia tej powłoki. Według informacji zamieszczonych w serwisie GitHub ostatnia aktualizacja powłoki nastąpiła w listopadzie 2017 roku. Wydaje się więc, że GitHub to teraz strona domowa projektu DreamPie.

Skorowidz

G

I

J

K

L

M

N

O

P

R

S

Ś

T

U

W

Z

Python — łatwiejszy, niż sądzisz, potężniejszy, niż myślisz!

Python jest językiem, którego można się nauczyć stosunkowo łatwo — a potem dość szybko przejść do praktyki. To duża zaleta: nic tak nie motywuje do dalszej pracy, jak pierwsze sukcesy na wczesnym etapie. Niemniej wielu nawet dość doświadczonych programistów Pythona nie wykorzystuje najlepszych cech tego języka. Ich aplikacje mogłyby być bardziej niezawodne, a kod — czystszy. Co gorsza, wiele ze znakomitych narzędzi i technologii powiązanych z Pythonem nie przebiło się do ogólnej świadomości społeczności skupionej wokół języka, przez co nie wykorzystuje się w pełni ich możliwości.

Celem tej książki jest rozwiązanie tego problemu. To rzecz przeznaczona dla programistów Pythona, którzy chcą znacząco poprawić jakość swoich aplikacji. Wyjaśniono tu mało znane lub błędnie rozumiane aspekty implementacji modułów standardowej biblioteki Pythona. Starannie opisano dekoratory, menedżery kontekstu, współprogramy i generatory oraz szczegóły wewnętrznego działania metod specjalnych. Pokazano alternatywne powłoki interaktywne, które mogą okazać się dużym ułatwieniem podczas kodowania. Ciekawym elementem książki jest prezentacja projektu PyPy, dzięki któremu można zapewnić współbieżność kodu. Nie zabrakło przydatnych informacji o tworzeniu dokumentacji kodu Pythona.

Dzięki tej książce między innymi:

- zrozumiesz różnice między plikami *.py* i *.pyc*
- wykorzystasz współprogramy do symulowania wielowątkowości
- zastosujesz moduł decimal do lepszego prowadzenia działań na liczbach zmiennoprzecinkowych
- zgłębisz tajniki podinterpreterów poprawiających współbieżność w Pythonie
- poprawisz funkcjonalność programu za pomocą dekoratorów

Cody Jackson — jest weteranem wojskowym, służył w marynarce. Założył firmę Socius Consulting zajmującą się IT i doradztwem w zakresie zarządzania biznesowego. Informatyką i programowaniem pasjonuje się od 1994 roku, jest samoukiem — sam zaczął pisać kod w Pythonie.

ISBN 978-83-283-5317-6
9 788328 353176
Cena: 57,00 zł

Packt>