Python - krótkie wprowadzenie

Na niniejszej stronie zebrałem kilka najważniejszych informacji oraz drogowskazów, ułatwiających, w założeniu, rozpoczęcie obcowania z popularnym językiem programowania, jakim jest **Python**. Idąc z duchem czasu, jedynym słusznym dla nas standardem jest najnowsza stabilna wersja z gałęzi wersji 3 tego języka. W momencie pisania tego tekstu, chodzi o wersję **3.6.5**. W praktyce jednak, w zakresie obowiązującym nas na zajęciach laboratoryjnych, wystarczająca będzie dowolna wersja z gałęzi 3.6 lub 3.5.

W porównaniu do znanego nam Basha, różnica polega przede wszystkim na tym, iż o ile ten pierwszy moglibyśmy śmiało nazwać po prostu interfejsem do uruchamiania innych programów i przetwarzania ich wyników, tak Python jest już bardzo zaawansowanym językiem programowania, pozwalającym definiować bardzo złożone struktury i abstrakcje programistyczne. Jego główną zaletą jest przejrzysta składnia, nastawiona na czytelność i brak zbędnych udziwnień. Chociaż na pozór Python może się wydawać językiem czysto obiektowym, tak naprawdę w pewnym zakresie wspiera także inne paradygmaty (między innymi programowanie funkcyjne). Nie bez znaczenia jest także dostępność ogromnej liczby modułów, pozwalających błyskawicznie skupić się bezpośrednio na realizowanym zadaniu. Wszystko to sprawia, iż w języku Python bardzo szybko pisze się programy, chociaż jednocześnie sam język jest bardzo dynamiczny i należy mieć na uwadze jego nieustanną ewolucję.

Informacje wstępne

Sam Python jako język definiuje po prostu składnię i semantykę języka programowania; opisuje paradygmaty i możliwe konstrukcje w ramach pewnej abstrakcji - w skrócie to, że przykładowy zapis

```
for i in [1, 2, 3]:
    print(i)
```

ma jakiś sens i zachowa się w określony sposób. Kod taki może zinterpretować chociażby człowiek, znajdując reguły składni tego języka, jednak w praktyce znacznie lepiej jest, żeby zrobił to sam komputer ;-)

Oczywiście, jak można podejrzewać, komputer wprost nie będzie w stanie poradzić sobie z zadaniem zrozumienia i wykonania przedstawionego powyżej kodu - musielibyśmy go przekształcić do odpowiedniego kodu maszynowego, zrozumiałego dla konkretnego procesora. Tak to się dzieje chociażby w przypadku programów napisanych w języku C - i, co prawda, da się to zrobić również w przypadku programów napisanych w języku Python, jednak wykonuje się to zwykle wyłącznie w bardzo szczególnych przypadkach, gdzie wydajność wykonania samego kodu ma kluczową rolę ponad jego przenaszalność i elastyczność. Pozostawmy tę ostatnią kwestię wyłącznie jako ciekawostkę, z hasłem `cython` jako ciekawostką do wyszukania dla bardziej ambitnych osób.

Narzędzie, które będzie dynamicznie potrafiło zinterpretować kod programu (skryptu) napisanego w języku Python, a następnie zrealizować odpowiednie, spodziewane akcje, będziemy nazywać interpreterem (szok, zaskoczenie). Jest to narzędzie podobne do kompilatora, a różnica pomiędzy tymi dwoma określeniami polega przede wszystkim na tym, że interpreter analizuje i wykonuje na bieżąco kolejne instrukcje napisanego programu, zaś kompilator najpierw przetwarza cały kod. Skutkiem jest to, że przygotowany program w pierwszym przypadku uruchamia się szybciej oraz jest bardziej elastyczny (często można go nawet modyfikować w locie), zaś jednak to program skompilowany będzie wykonywał się (z reguły) znacznie wydajniej, kiedy już zostanie skompilowany (często ten proces wiąże się z szeregiem optymalizacji). Z drugiej strony często nie bez znaczenia jest fakt, iż przygotowane skrypty z reguły zajmują niewiele miejsca, a jedynym warunkiem ich uruchomienia na dowolnej platformie, czy w systemie operacyjnym, jest dostępność na niej/nim odpowiedniego interpretera, kiedy kompilowalne programy wymagają zapewnienia odpowiednich dynamicznie ładowalnych bibliotek w systemie (także już skompilowanych), albo wykonania tak zwanego linkowania statycznego, które owocuje ogromnymi (pod względem rozmiaru) plikami wykonywalnymi.

Podobnie jak istnieje wiele kompilatorów języka C, dostępne są różne interpretery języka Python. Różnią się one przede wszystkim językiem implementacji samego interpretera. Z najważniejszych implementacji, warto kojarzyć takie narzędzia, jak: CPython (napisany w C; najpopularniejszy); Jython (dawniej JPython; oparty na Javie) IronPython (napisany w C#) oraz PyPy (interpreter języka Python, napisany w języku Python!). Różne implementacje stosowane są najczęściej w specyficznych zastosowaniach, kiedy na przykład posiadamy już jakieś duże narzędzie opracowane w innym języku i chcemy umożliwić jego rozszerzanie przez wtyczki w języku Python. Co do zasady, kod w języku Python powinien zachowywać się tak samo niezależnie od wybranego interpretera, aczkolwiek należy mieć na uwadze, iż mogą występować pewne bardzo, bardzo subtelne różnice, na przykład w zachowaniu wątków czy arytmetyki zmiennoprzecinkowej.

Jak zacząć

Zgodnie z tym, co zostało powiedziane wyżej, najważniejszą rzeczą, potrzebną do uruchomienia skryptu, napisanego w języku Python, jest posiadanie odpowiedniego interpretera tego języka. Najpopularniejszy **CPython** dostępny jest w większości systemowych menadżerów pakietów pod nazwą **python3** lub po prostu **python** (ten drugi wariant szczególnie w przypadku bardziej postępowych dystrybucji Linuksa).

Instalacja w systemach z rodziny Ubuntu Linux - jako administrator należy wykonać w powłoce komendę:

```
apt-get install python3
```

Instalacja w ~~najlepszym istniejącym~~ systemie operacyjnym Arch Linux - jako administrator proszę skorzystać z komendy:

```
pacman -Syu python
```

Instalacja w systemach z rodziny macOS - najlepiej jest użyć menadżera pakietów **brew**:

```
brew install python3
```

W systemach z rodziny Microsoft Windows... Po przemyśleniu, co robi się ze swoim życiem, posiłkowałbym się oficjalną instrukcją. Podobno działa i podobno da się tego rozsądnie używać (to znaczy - z poziomu konsoli), ale moje upośledzenie w tym temacie nie pozwala mi tego potwierdzić. Przepraszam.

UWAGA! Chociaż dostępne są wersje interpretera języka Python na różne platformy sprzętowe i systemy operacyjne, to sam Python powinien się zachowywać identycznie niezależnie od konfiguracji środowiska, w którym jest uruchomiony. Trzeba jednak pamiętać, że ostatecznie wszystkie możliwości interpretera będą ograniczone przez to, co jest dostępne w danym środowisku. Przykładowo, jeśli system plików w naszym miejscu pracy nie będzie obsługiwał dowiązań tudzież mechanizmów kontroli uprawnień dostępów (na przykład w systemach plików `FAT`/`NTFS`, typowych dla systemów operacyjnych z rodziny Microsoft Windows) to wykonanie odpowiednich instrukcji w języku Python nie przyniesie żadnego efektu, albo wręcz zakończy się błędem. W przypadku funkcji i modułów biblioteki standardowej, opisanych w oficjalnej dokumentacji, zwykle można znaleźć odpowiednie adnotacje o takich ograniczeniach.

Niezależne od wszystkiego, w końcu powinno odnieść się sukces w instalacji interpretera języka Python. Poprawność instalacji można potwierdzić, uruchamiając w konsoli komendę

```
python3 --version
```

Uzyskanie w wyniku ciągu z numerem wersji, który będzie rozpoczynał się od cyfry **3**, oznacza, iż jesteśmy gotowi do pracy.

Osoby bardziej zainteresowane mogą pokusić się o zaopaotrzenie swojego środowiska pracy w porządny edytor/narzędzie wspierające tworzenie programów w języku Python. Spośród cieszących się największym uznaniem użytkowników, warto wymienić tutaj program PyCharm. Na potrzeby naszych zajęć, wystarczającym będzie jednak **vim** lub chociażby nawet zwykły notatnik.

Pierwszy skrypt

Pod względem mechanicznym, przygotowanie i uruchomienie dowolnego skryptu, przygotowanego w języku Python, nie różni się zbytnio od wcześniej opracowywanych przez nas skryptów powłoki Bash. Tradycyjnie skrypty języka Python to pliki tekstowe o rozszerzeniu `*.py`, na przykład `skrypt.py`. Uruchomić taki skrpyt możemy na dwa sposoby.

Pierwszy wariant to uruchomienie z poziomu konsoli samego interpretera języka Python, przekazując mu jako argument ścieżkę do pliku, zawierającego interesujący nas kod do wykonania; przykładowo:

```
python3 skrypt.py
```

Alternatywnie można uczynić wykonywalnym sam taki plik z kodem (`chmod +x skrypt.py`) i opatrzyć go odpowiednim nagłówkiem ze ścieżką do interpretera (tak zwany *shebang*, zwykle `#!/bin/bash` w naszych dotychczasowych skryptach). Aby odnaleźć scieżkę do odpowiedniego interpretera, możemy posłużyć się programem `whereis`. Wywołanie `whereis python3` zaskutkuje zwróceniem nam ścieżki, na przykład `/usr/bin/python3`, którą można wykorzystać. W praktyce jednak znacznie częściej obecnie stosuje się program `env` do dynamicznego odnalezienia programu o interesującej nas nazwie wśród katalogów podanych w zmiennej środowiskowej `$PATH`. Odpowiedni zapis nagłówka wygląda wtedy następująco:

```
#!/usr/bin/env python3
```

wtedy możemy zwyczajnie wywołać sam skrypt z poziomu konsoli:

```
./skrypt.py
```

Zastosowanie programu `env` jest bardzo popularne szczególnie w odniesieniu do języka Python, gdyż często może nam zależeć na zastosowaniu interpretera w jakiejś konkretnej wersji - tutaj kolejne hasło dla osób bardziej zainteresowanych, moduł `venv` (środowiska wirtualne - virtual environments). W innych wypadkach zaś (nie tylko języka Python) program `env` pozwala uczynić nasz skrypt bardziej przenaszalnym - chociaż w większości systemów programy są typowo instalowane w katalogu `/usr/bin/`, to zdarzają się wyjątki...

Podstawy składni

W sieci można znaleźć wiele bardzo dobrych, istniejących już kursów języka Python, dlatego nie warto tworzyć naprędce kolejnego, tylko lepiej skorzystać, na przykład z tego. Pierwsza jego część, "Python 3 Basic Tutorial", w bardzo jasny sposób omawia właściwie wszystkie najistotniejsze mechanizmy i konstrukcje, które będą potrzebne w ramach naszych zajęć. W szczególności proszę też, ze względu na tematykę kursu, zwrócić uwagę na rozdział "Python 3 - Files I/O". Tutaj zaś skupimy się na dwóch innych rzeczach.

Pierwszą istotną rzeczą jest sposób wczytywania parametrów, przekazanych do skryptu podczas jego uruchomienia. W powłoce Bash odpowiednie argumenty były dostępne po prostu jako zmienne `$1`, `$2`, itd. W języku Python takie argumenty nie są jednak dostępne wprost, a jedynie za pośrednictwem modułu **sys**. Konkretnie to należy najpierw załadować taki moduł, a następnie przekazane argumenty można odnaleźć w specjalnej liście **sys.argv**:

```
#!/usr/bin/env python3
import sys

for (i, arg) in enumerate(sys.argv):
    print('Argument {} to {}'.format(i, arg))
```

W podanym przykładzie najpierw wczytujemy moduł **sys**. Następnie w pętli iterujemy po każdym elemencie tablicy `sys.argv`. Dodatkowo, funkcja `enumerate()` działa w ten sposób, że dla każdego kolejnego elementu z iterowalnego zbioru, zwraca ona krotkę, składającą się z kolejnego numeru (poczynając od zera `0`) oraz samego danego elementu. Na koniec wyświetlamy oba elementy każdej krotki, podstawiając je za pomocą metody `format()` z obiektu klasy `str` (ciągu znaków). Przykładowe uruchomienie powyższego skryptu da następujący efekt:

```
szymon@Tardis:~$ ./skrypt.py abc def ghi
Argument 0 to ./skrypt.py
Argument 1 to abc
Argument 2 to def
Argument 3 to ghi
```

Więcej o module **sys** można dowiedzieć się z oficjalnej dokumentacji.

Druga uwaga wiąże się ze stylem pisanego kodu. Chociaż ze względu na specyfikę języka, napisanie naprawdę brzydkiego kodu jest raczej niełatwe, to jednak wykonalne. Z tego powodu powstał oficjalny dokument, definiujący sposób zapisu różnych elementów składni w języku Python, tak aby zwiększyć czytelność kodu. Dokument ten można znaleźć w sieci, wpisując frazę PEP-8 - dokument numer 8 z listy *Python Enhancement Proposals*. Warto się z nim zapoznać i stosować do wymienionych w nim zasad, bowiem tworzenie kodu zgodnie z jego regułami sprawi, iż nasz kod będzie z reguły lepiej oceniany przez społeczność i ewentualnych współpracowników (oglądanie rzeczy sformatowanych według wspólnego schematu pozwala bardziej skupić się na samej logice kodu). Istnieje narzędzie, podobne do znanego nam programu **shellcheck**, automatycznie sprawdzające

zgodność naszego kodu z dobrymi praktykami, wypracowanymi przez społeczność. Nazywa się ono flake8 i można go znaleźć jako pakiet w standardowych systemowych repozytoriach pakietów, albo używając menadżera modułów o nazwie **pip**. Ten temat również pozostawiam bardziej zainteresowanym osobom.

Moduł `glob`

Moduł `glob` pozwala w szybki i prosty sposób uzyskać listę ścieżek do istniejących elementów systemu plików, których położenia pasują do określonego wzorca. Nazwa modułu pochodzi od mechanizmu o tej samej nazwie, który wykorzystywany jest, między innymi w powłoce Bash, do tworzenia pasujących ścieżek - innymi słowy do tego, żeby komenda `ls dir/file*.txt` została uruchomiona tak naprawdę jako `ls dir/file1.txt dir/file2.txt dir/file7.txt` (wzorzec `dir/file*.txt` pozwolił odnaleźć 3 pasujące ścieżki do plików).

Oficjalna dokumentacja modułu glob jest bardzo zwięzła, a jednocześnie świetnie ilustruje na przykładach najważniejsze warianty użycia.

Moduł `os`

Moduł `os` to potężny zestaw wielu użytecznych interfejsów, pozwalających wykonywać różne akcje systemu operacyjnego, głównie te związane z systemem plików, chociaż nie tylko. Można tutaj znaleźć między innymi funkcje do tworzenia dowiązań, testowania atrybutów plików, czy zmiany nazw lub usuwania elementów systemu plików. Należy mieć na uwadze, iż moduł `os` dostarcza także funkcje `os.open()` i podobne, które pozwalają na niskopoziomowe operacje na plikach (czytanie w trybie binarnym, przesunięcia w celu odczytania konkretnych wartości, itd.). Jest to coś zupełnie innego, niż wbudowana w język funkcja `open()`.

Podmoduł `os.path` skupia się zaś na wszelkiego rodzaju manipulacjach ścieżkami. Pozwala przede wszystkim budować ścieżki w sposób uniwersalny i niezależny od systemu operacyjnego - na przykład w stosunku do stosowanych separatorów katalogów / (Unix/Linux) albo \ (Windows). Znajdziemy tutaj także funkcje do testowania, czy element systemu plików wskazywany przez ścieżkę istnieje, a przy tym jest on określonego typu elementem (plikiem, katalogiem, dowiązaniem).

Oficjalna dokumentacja modułu os oraz podmodułu os.path jest bardzo obszerna - proszę przynajmniej pobieżnie zapoznać się z dostępnymi metodami i funkcjami, aby w razie p.otrzeby szybko odnaleźć potrzebne rzeczy i ich opcje.

Moduł `subprocess`

TBA

Moduł `re`

TBA
