Podstawy składni języka Python

Alloha World!

W lekcjach programowania utarło się, że zawsze zaczynamy od już przysłowiowego "Hello World". Skrypty czy programy tego typu nie mają na celu pokazania jak minimalną ilością znaków da się wyświetlić coś na ekranie, a sposób interakcji i przepływu programista-komputer.

W Pythonie mamy możliwość wykorzystania interpretera REPL, przykład poniżej oraz stworzenia skryptu, który wykonamy z linii poleceń.

```
$ python

Python 3.6.0 (default, Dec 24 2016, 08:01:42)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> print('Hello World!')
Hello World!
```

Zwróć uwagę na wersję Pythona. Jeżeli po wpisaniu polecenia `python` uruchomi się wersja 2.x, możesz spróbować polecenia `python3` Powyższy przykład ilustruje moment wpisania polecenia `python`, standardowy tekst informujący o wersji i kompilacji języka oraz znak zachęty >;>;>; (ang. prompt). Polecenia wpisujemy po tym znaku a ich wynik wyświetla się poniżej (i nie zawiera wcięcia). Dalej w materiałach będziemy posługiwali się już samym znakiem zachęty.

Drugim sposobem jest stworzenie skryptu posiadającego następujące linie. Ta metoda przydaje nam się gdy nasze programy zaczną rosnąć na więcej niż jedną dwie linijki. Warto zwrócić uwagę na pierwszą linię, na tzw. shebang `#!` i następujące po nim polecenie. To jest deklaracja programu, którego kod źródłowy znajduje się poniżej. Linijka ta jest opcjonalna, ale dla zachowania poprawności i warto w naszych skryptach coś takiego zadeklarować. Już po pierwszej linii widzimy, że skrypt będzie zinterpretowany jako kod źródłowy trzeciej wersji Pythona.

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

print('Hello World!')
```

Wynik uruchomienia powyższego skryptu będzie identyczny z efektem uzyskanym w REPL, tzn, na naszym ekranie ukaże się napis `Hello World`.

Dla wszystkich, którzy potrzebują wiedzieć jak wygląda najmniejszy kod, który wyświetli nam te słowa polecam poniższy kod.

```
print('Hello World!')
```

Deklaracje na początku pliku

Deklaracja interpretera

Jest to specjalny rodzaj komentarza który opisaliśmy pokrótce powyżej. Ten typ komentarza występuje tylko w pierwszej linii programu i definiuje interpreter kodu źródłowego dla kodu poniżej.

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

Deklaracja kodowania znaków w pliku

Jest to kolejny rodzaj specjalnego komentarza, który instruuje interpreter i jawnie wskazuje na sposób kodowania znaków w pliku z kodem źródłowym. W skryptach Pythona w wersji drugiej był obowiązkowy, jeżeli w kodzie lub komentarzach w pliku znajdowały się znaki z poza zakresu ASCII, np. polskie znaki diakrytyczne ą, ę, ś, ć itp.

```
# -*- coding: utf-8 -*-
```

Informacja na temat modułu

Pierwszy wielolinijkowy komentarz w pliku jest traktowany jako opis modułu. Może się w nim znajdować np. licencja użytkowania programu, instrukcja jego obsługi itp. Bardzo ciekawym pomysłem jest również napisanie komentarza opisującego parametry programów wykorzystującego standard *unix takiego opisu. Dzięki temu poza samym jednoznacznym opisem działania programu zgodnym z ogólnie przyjętą konwencją dostajemy możliwość wykorzystania modułu docopt do jego sparsowania i obsługi parametrów przekazywanych z linii poleceń.

Docopt bierze opis z komentarza i parsuje zmienne zgodnie z instrukcją czyniąc niektóre elementy obligatoryjnymi, możliwymi do podania jedno - lub wielokrotnie itp. Samym opisem działania tego modułu zajmiemy się w sekcji jemu poświęconej.

Wcięcia zamiast nawiasów klamrowych

Jest to chyba najbardziej ciekawa rzecz w samym języku. Autorzy specyfikacji zdecydowali się na zastąpienie nawiasów klamrowych wcięciami, czyli tzw. białymi spacjami (ang. whitespace). Jest to dość nietypowe rozwiązanie, które okazało się bardzo rewolucyjne i niesamowicie podniosło czytelność kodu źródłowego.

Sama idea spowodowała dużą polaryzację programistów. Jedni bardzo sobie chwalą to rozwiązanie, a inni przyzwyczajeni do języków przypominających składnią C są jej zaciekłymi wrogami. Osobiście jestem wielkim zwolennikiem takiego rozwiązania!

```
>>> from __future__ import braces
  File "<stdin>", line 1
SyntaxError: not a chance
```

Końce linii

Pierwszą rzeczą (poza znaczącymi wcięciami), która może zaskoczyć programistów przyzwyczajonych do składni C jest brak konieczności, a nawet zalecenie do niestawiania znaku średnika ; na końcu linii. Programy interpretowane są linia po linii. Linia kończy się tam, gdzie ostatni znak polecenia.

Python pozwala na stosowanie znaków końca linii zarówno znanych z systemów Windows (rn) jak i środowiska *nix (n).

Duck typing

W językach programowania można doszukać się wielu systemów typowania. System typowania informuje kompilator o obiekcie oraz o jego zachowaniach. Ponadto niesie za sobą informację na temat ilości pamięci, którą trzeba dla takiego obiektu zarezerwować. Istnieje nawet cała gałąź zajmująca się systemami typów. Obecnie najczęściej wykorzystywane języki programowania dzielą się na statycznie - silnie typowane (JAVA, C, C++ i pochodne) oraz dynamicznie - słabo typowane (Python, Ruby, PHP itp.). Oczywiście mogą znaleźć się rozwiązania hybrydowe oraz z tzw. inrefencją typów itp.

W naszym przypadku skupmy się na samym mechanizmie dynamicznego typowania. Określenie to oznacza, że język nie posiada typów zmiennych i obiektów, które jawnie trzeba deklarować. Inicjując zmienną nie musimy powiedzieć, że jest to `int`. Co więcej po chwili do tej zmiennej możemy przypisać dowolny obiekt, np. łańcuch znaków i kompilator nie powie nam złego słowa. Kompilator podczas działania oprogramowania niejawnie może zmienić typ obiektu i dokonać na nim konwersji.

Wśród programistów popularne jest powiedzenie "jeżeli chodzi jak kaczka i kwacze jak kaczka, to musi być to kaczka". Od tego powiedzenia wzięła się nazwa Duck typing. Określenie to jest wykorzystywane w stosunku do języków, których typy obiektów rozpoznawane są po metodach, które można na nich wykonać. Nie zawsze takie zgadywanie jest celne i jednoznacznie i precyzyjnie określa typ. Może się okazać, że obiekt np. `Samochód` posiada metody `uruchom_silnik()` i `jedz_prosto()` podobnie jak `Motor`. Jeden i drugi obiekt będzie zachowywał się podobnie. Języki wykorzystujące ten mechanizm wykorzystują specjalne metody porównawcze, które jednoznacznie dają informację kompilatorowi czy dwa obiekty są równe.

Sam mechanizm dynamicznego typowania jest dość kontrowersyjny, ze względu na możliwość bycia nieścisłym. W praktyce okazuje się, że rozwój oprogramowania wykorzystującego ten sposób jest dużo szybszy. Za to zwolennicy statycznego typowania, twierdzą, że projekty wykorzystujące duck typing są trudne w utrzymaniu po latach. Celem tego dokumentu nie jest udowadnianie wyższości jednego rozwiązania nad drugim. Zachęcam jednak do zapoznania się z wykładem "The Unreasonable Effectiveness of Dynamic Typing for Practical Programs", którego autorem jest "Robert Smallshire". Wykład zamieszczonym został w serwisie InfoQ (http://www.infoq.com/presentations/dynamic-static-typing). Wykład w ciekawy sposób dotyka problematyki porównania tych dwóch metod systemu typów. Wykład jest o tyle ciekawy, że bazuje na statystycznej analizie projektów umieszczonych na https://github.com a nie tylko bazuje na domysłach i flamewar jakie programiści lubią prowadzić.

Wszystko jest obiektem

W Pythonie wszystkie rzeczy są obiektem. Każdy element posiada swoje metody, które możemy na nim uruchomić. W dalszej części tych materiałów będziemy korzystali z polecenia `help()` aby zobaczyć jakiego z jakiego typu obiektem mamy okazję pracować oraz co możemy z nim zrobić.

Komentarze

Komentarze są wykorzystywane by podpowiedzieć programiście, który będzie czytał kod źródłowy w przyszłości co dana funkcja, metoda lub po prostu kolejna linijka kodu robi. Jestem wielkim fanem pisania tak swoich programów, aby komentarze w kodzie były zbędne. Dobrego dzielenia aplikacji na mniejsze części, właściwego stosowania whitespace'ów, precyzyjnego i opisowego ich nazywania. Komentarze mogą być bardzo przydatne, ale w większości sytuacji jeżeli potrzebujemy z nich skorzystać to znaczy, że logicznie źle rozplanowaliśmy układ naszego kodu. Ponadto komentarze mają brzydką właściwość szybkiego starzenia się, tzn. kod ewoluuje, a komentarz opisuje zachowanie starej funkcji. Może to powodować dezinformację.

Zakomentowany kod

Bardzo często spotykam się z problemem zakomntowanego kodu. O ile komentarze opisujące działanie poszczególnych elementów są użyteczne to zakomentowany kod jest nieakceptowalny. Często stosujemy tą technikę by chwilowo wyłączyć działanie jakiejś funkcjonalności. Jednakże niedopuszczalne jest commitowanie zmian zawierających zakomentowany kod. Kod taki bardzo często jest już niedziałający i taki pozostanie na zawsze. Bardzo często słyszę argument, że może kiedyś będziemy chcieli powrócić do tego kodu i bez sensu będzie go wymyślać i pisać na nowo. W dobie systemów kontroli wersji sytuacja ta nie będzie stwarzała jakiegokolwiek problemu. Wystarczy przeglądnąć diffa (podgląd różnicowy) pliku albo wykonać `git blame` i mamy dostęp do starego sposobu.

Nieuruchamiający się i niewywoływany kod nie powinien znaleźć się w repozytorium. Kropka!

Komentowanie linii

W Pythonie mamy kilka sposobów komentowania. Najprostszym z nich jest komentowanie całej linii poprzez wykorzystanie znaku zwanego "pound" lub "hash" #. Ciąg znaków znajdujących się za # zostanie zignorowany przez kompilator.

```
>>> # na ekranie otrzymamy: Hello World!
... print('Hello World!')
Hello Wold!
```

Tu możemy zaobserwować zachowanie, o którym wspominaliśmy trochę wcześniej, tzn. kontynuacja jest oznaczana przez znak zachęty trzech kropek `...`.

Komentarze inline

Kolejnym sposobem jest komentowanie inline tzn. w linijce. Tego typu komentarze stosuje się aby wytłumaczyć zachowanie poszczególnych linii kodu. Choć kompilator dopuszcza ich stosowanie, to w ramach dobrych praktyk lepiej zastąpić je komentarzami w linijce poprzedzającej wywołanie.

```
>>> print('Hello Wold!') # na ekranie otrzymamy: Hello World!
Hello Wold!
```

Komentarze wieloliniowe

Komentarze wieloliniowe w Pythonie można robić na dwa sposoby poprzez wykorzystanie trzech znaków cudzysłowia:

- pojedynczego ';';',
- podwójnego ";";";.

W jednym i drugim przypadku cudzysłowie podwójne lub pojedyncze będzie oznaczało początek jak i koniec komentarza. Rodzaj cudzysłowiów nie ma znaczenia, ale utarło się aby stosować podwójne ";.

```
"""
Tu jest treść komentarza, który obejmuje wiele linii
W ramach dobrych praktyk, powinniśmy takim komentarzem opisać każdą z funkcji,
aby narzędzia takie jak np. ``help()`` wyświetlały ładne podpowiadanie działania.
"""
```

Są dwie szkoły tworzenia takich komentarzy. Jedna mówi, aby tekst pisać bezpośrednio po znaku cudzysłowia, a druga od nowej linijki. Jest to kwestia estetyki i czytelności komentarza.