Moduł 1. Krótka historia Pythona. PEP8.

1. Wstęp

1.1 Historia języka Python

Twórcą Pythona jest holender **Guido van Rossum**, a sama nazwa, co niektórych zapewne nie zdziwi, pochodzi od popularnego serialu BBC „Latający Cyrk Monty Pythona". Prace nad pierwszym interpreterem Pythona rozpoczęły się w 1989 roku jako następcy języka ABC. Wszystkie wersje aż do wersji 1.2 powstawały w CWI (Centrum Matematyki i Informatyki) w Amsterdamie gdzie Guido wówczas pracował. Od wersji 2.1 Python był udostępniany jako projekt **Open Source** przez niedochodową organizację **Python Software Foundation (PSF)**. Obecnie nad rozwojem Pythona pracuje wiele osób, ale Guido wciąż jest zaangażowany w ten proces. **Update 2021**: Po sporach ze społecznością co do dalszego kierunku rozwoju języka Python Guido angażuje się już znacznie mniej w ten proces.

Sam twórca w 1995 roku wyemigrował do USA gdzie w latach 2005 – 2013 pracował dla Google a następnie dla Dropbox'a (do października 2019 roku). We wspólnocie Pythona van Rossum pełnił do lipca 2018 roku funkcję Benevolent Dictator for Life (BDFL), co oznacza, że nadzorował rozwój języka, podejmując w razie konieczności ostateczne decyzje. W 2019 roku był jeszcze członkiem Python Steering Council, ale wycofał swoją kandydaturę na rok 2020. Po odejściu z Dropboxa ogłosił przejście na emeryturę, ale już w listopadzie 2020 ogłosił, że dołącza do firmy Microsoft, aby dalej rozwijać Pythona.

Ważnym momentem w historii Pythona było utworzenie drugiej głównej gałęzi – Pythona 3 w roku 2008. Od tego momentu wersja 2 oraz 3 były rozwijane oddzielnie, ale czas wersji 2 właściwie minął, o czym świadczy termin zakończenia wsparcia z dniem 12 kwietnia 2020 roku. Rozwój języka jest prowadzony przy wykorzystaniu **PEP (Python Enhancement Proposal)**. Dokumenty te to propozycje rozszerzeń lub zmian w języku w postaci artykułu, który jest poddawany pod dyskusję wśród programistów Pythona. Każdy dokument zawiera opis proponowanego rozwiązania, uzasadnienie oraz aktualny status. Po osiągnięciu konsensusu propozycje są przyjmowane lub odrzucane.

W dokumencie https://peps.python.org/pep-0020/ został przedstawiony **The Zen of Python**, który nakreśla idee organizacji składni języka, którymi posługiwali się jego twórcy, ale też jest zbiorem zaleceń dla innych programistów tego języka.

1.2. Wybrane cechy języka Python

Python jest językiem **ogólnego przeznaczenia**, którego ideą przewodnią jest czytelność i klarowność kodu źródłowego. Standardową implementacją języka jest CPython (napisany w C), ale istnieją też inne, np. Jython (napisany w Javie), CLPython napisany w Common Lisp, IronPython (na platformę .NET) i PyPy napisany w Pythonie.

Python nie wymusza jednego stylu programowania, dając możliwość programowania obiektowego, programowania strukturalnego oraz programowania funkcyjnego.

Inne cechy języka Python:

- **Typy** sprawdzane są **dynamicznie** (w przeciwieństwie np. do Javy, C#),
- Do zarządzania pamięcią używany jest garbage collector,

- Wszystkie wartości przekazywane są przez referencję,
- Jest czasem kwalifikowany jako język skryptowy,
- Nie ma enkapsulacji, jak to ma miejsce w C++ czy Javie, ale istnieją mechanizmy, które pozwalają na osiągnięcie podobnego efektu,
- Możliwe jest tworzenie funkcji ze zmienną liczbą argumentów,
- Możliwe jest tworzenie funkcji z argumentami o wartościach domyślnych.

1.3. Python i data science

Python nie jest jedynym ani też jednoznacznie najlepszym językiem dla data science. Jego największym konkurentem w tej dziedzinie jest **R**, który od samego początku był tworzony z myślą o statystyce, która jak wiadomo w dziedzinie data science ma szerokie zastosowanie. Trwają niezliczone spory i porównania próbujące udowodnić wyższość jednego rozwiązania nad drugim. Skoro jednak będziemy zajmować się Pythonem przytoczę kilka jego zalet pod kątem data science:

- **Python jest językiem ogólnego przeznaczenia** co powoduje, że oprócz możliwości wykorzystania specjalistycznych bibliotek np. do budowy sztucznych sieci neuronowych, można bez konieczności integracji z innymi rozwiązaniami zbudować kompletną aplikację desktopową lub webową,

- **Python może być traktowany jako język skryptowy** co dodatkowo w połączeniu z np. Jupyter Notebook (dawniej IPython), pozwala na bardzo szybkie testowanie i prototypowanie poprzez pisanie kodu „na bieżąco", co powoduje brak konieczności zapisywania kodu w pliku i jego późniejszego uruchamiania co znacznie przyspiesza proces stworzenia działającego prototypu,

- **Bogata paleta bardzo dobrej jakości bibliotek dla AI (Artificial Intelligence) i Data Science**. Za przykład mogą tutaj posłużyć **NumPy, Pandas, SciPy, matplotlib czy scikit-learn**, które jednak wykraczają poza zakres niniejszego przedmiotu.

- **Społeczność** – jako język ogólnego zastosowania społeczność Pythona jest bardzo duża co przekłada się na łatwość uzyskania odpowiedzi na pytania, sporą ilość dobrej dokumentacji oraz rozbudowaną listę bibliotek i dodatków.

2. Organizacja kodu według PEP8

Jak zostało już wspomniane w punkcie 1 zmiany w specyfikacji języka odbywają się poprzez system PEP. Dokument o numerze PEP8 jest jedną (ale nie jedyną) propozycją organizacji kodu języka Python. Oryginalna treść dokumentu dostępna jest pod adresem https://www.python.org/dev/peps/pep-0008/.

Pod adresem https://realpython.com/python-pep8/ można znaleźć więcej przykładów z wyjaśnieniami jak stosować zalecenia PEP8 w praktyce oraz jakie narzędzia mogą nam w tym pomóc.

2.1. Wcięcia

W kodzie języka Python nie znajdziemy znanych z PHP, Javy czy C# klamerek do separacji bloków kodu, określania ram ciała metody czy klasy lub zakresu operacji w pętli. Tutaj do tego celu służą odpowiednio umieszczone **wcięcia i puste linie** między w/w elementami. Dla osób, które nigdy wcześniej nie miały do czynienia z taką organizacją kodu może to być pewną nowością, ale dość szybko staje się zrozumiałe i intuicyjne.

Listing 1

```
if score >= 100:
    print('Zwycięstwo !')
```

Każdy kolejny poziom zagnieżdżenia w bloku kodu poprzedza odstęp w postaci wielokrotności **4 spacji (pojedyncza wartość wcięcia)**. Dopuszczalne jest również stosowanie tabulatorów jako wcięć, ale zalecane są spacje a dodatkowo w wersji Python 3 użycie jednocześnie spacji i tabulatorów jako wcięć nie jest dozwolone. Zazwyczaj nie musimy się jednak martwić ręcznym wstawianiem wcięć, chyba że korzystamy z narzędzia do edycji kodu, które nie posiada wsparcia dla danego języka programowania.

Wcięcia używamy również w sytuacjach, w których linia kodu jest zbyt długa i powinna być złamana na większą ilość wierszy. Zalecana długość linii według PEP8 to 79 znaków.

Listing 2

```
wyslane = wyslij_wiadomosc(e_mail_odbiorcy, temat,
                           wiadomosc)
```

W takim przypadku wcięcie sięga znaku otwarcia deklaracji listy atrybutów wywoływanej metody lub funkcji. Dopuszczalne są jednak odstępstwa od tej reguły pozwalające na zastosowanie mniejszego wcięcia dla kolejnych linii.

Listing 3

```
wyslane = wyslij_wiadomosc(e_mail_odbiorcy,
    temat, wiadomosc)
```

Jednak sama dokumentacja mówi o tym, że jest to opcjonalne formatowanie, więc należy go używać tylko z konieczności a nie jako regułę.

Deklaracja zmiennych takich jak lista, tablica, krotka czy słownik dzięki wcięciom często poprawia ich czytelność co jest główną przyczyną, którą kierowano się określając reguły formatowania kodu w Pythonie.

Listing 4

```
lista = [
    1, 2, 3,
    4, 5, 6,
]
```

W przypadku łamania linii i operatorów (np. arytmetycznych) obowiązuje zasada przenoszenia operatora do nowej linii.

Listing 5

```
zysk = (przychod
        - koszty
        - podatki)
```

2.2. Puste linie

Funkcje najwyższego rzędu oraz definicje klas oddzielamy od pozostałych bloków kodu dwiema pustymi liniami.

Listing 6

```
def zrob_cos():
    return 'zrobione'


def tez_cos_zrob():
    return 'też zrobione'
```

Metody klas oraz funkcje lokalne oddzielone są natomiast pojedynczą pustą linią.

Listing 7

```
class Osoba:

    def __init__(self, imie, nazwisko, plec):
        self.imie = imie
        self.nazwisko = nazwisko
        self.plec = plec

    def przedstaw_sie(self):
        print('Nazywam się {0} {1}'.format(self.imie, self.nazwisko))

    def moj_wiek(self):
        print('Moja płeć to: {0}.'.format(self.plec))


os = Osoba('Jan', 'Kowalski', 'mężczyzna')
os.przedstaw_sie()
os.moj_wiek()
```

Listing 8

```
def funkcja_top_level():

    def funkcja_lokalna():
        pass
```

```
    def kolejna_funkcja_lokalna():
        pass
```

Pojedyncze puste linie mogą być również stosowane wewnątrz funkcji aby odseparować od siebie logiczne sekcje funkcji.

2.3. Organizacja importów

Poszczególne instrukcje importu powinny być rozdzielone na oddzielne linie. **Listing 9**

```
# Tak
import os
import sys

# Nie
import sys, os
# Poprawny jest natomiast taki sposób definiowania importu:
from subprocess import Popen, PIPE
```

Inne zasady dotyczące organizacji importów.

Importy powinny być umieszczane na początku pliku tuż za ewentualnymi komentarzami dla modułu i elementami docstring. Kolejnośc importów ma również znaczenie.

Oto zalecany porządek:

- import bibliotek standardowych
- import powiązanych bibliotek zewnętrznych (ang. third party imports)
- import lokalnych aplikacji/bibliotek

Zalecane jest również dodawanie pustej linii po każdej z w/w grup importów. Jako, że Python umożliwia zarówno import całej biblioteki lub tylko wybranych jej modułów często trzeba dobrać odpowiedni sposób do sytuacji, ale z reguły zaleca się wykonywanie importu i dodanie aliasu lub import modułu zamiast konkretnej klasy z tego modułu co zmniejsza ryzyko wystąpienia konfliktów w przestrzeni nazw. Więcej informacji oraz przykłady znajdują się w rozdziale poświęconym zarządzaniu i importowi pakietów.

Dobrym pomysłem na zdobycie wiedzy na temat dobrej organizacji kodu w naszych modułach jest zaglądanie do kodu źródłowego publicznie dostępnych, dobrej jakości modułów i bibliotek Pythona.

2.4. Konwencje nazewnicze

Jest to moim zdaniem jedna z ważniejszych zasad do przyswojenia i pilnowania aż do wyrobienia nawyku. Warto zatem mieć pod ręką krótkie zestawienie:

- https://peps.python.org/pep-0008/#naming-conventions
- https://realpython.com/python-pep8/#naming-styles

Zasady co do zalecanego stylu są dość precyzyjne, ale dobranie odpowiedniej nazwy wymaga już nieco więcej wprawy i przemyślenia. Wykorzystanie jednoliterowych nazw dla zmiennych opisujących niewiadome w równaniu ma sens, ale użycie takiej samej zasady dla przechowania imienia, listy nazw produktów będzie znacznie utrudniało interpretację kodu i jego szybkie zrozumienie. Również autorowi i to po dość krótkiej przerwie w obcowaniu z tym kodem. Używanie zbyt długich nazw również ma swoje wady, które zamiast poprawienia czytelności zadziałają wręcz przeciwnie, lub zbyt często będą nas zmuszały do łamania pojedynczej linii kodu. Tu zalecany jest umiar. Warto również, mimo tego, że Python jest językiem typowanym dynamicznie, nawiązywać w jakiś stopniu do typu wartości, które zmienna przechowuje, np. `names_list lub list_of_names (typ list), has_permission lub is_valid (typ bool), number_of_digits lub word_count (typ numeryczny) itp.`. To jest proces i dlatego warto wracać do swojego kodu po pewnym czasie i wykonywać **refaktoryzację** samodzielnie lub wykorzystać inne osoby w zespole do przeprowadzenia procesu **code review**.

2.5. Inne zalecenia

Zmienne typu string można umieszczać zarówno w cudzysłowie lub w apostrofach, gdyż w przypadku Pythona nie ma to znaczenia. Natomiast PEP8 nie zaleca żonglowania tym zapisem i trzymania się jednej z opcji. Sytuacją, w której dozwolone jest użycie obu jednocześnie, jest ciąg tekstowy, który sam już zawiera cudzysłów lub apostrof – wtedy należy użyć innego niż w samym tekście, lub zastosować zapis, gdzie każdy specjalny znak będzie poprzedzony znakiem ucieczki (ang. escape sign) ****.

Listing 10

```
artykul = 'Recenzja "Władcy Pierścieni".'
artykul = "Recenzja 'Władcy Pierścieni'."

# Ale można również tak:
artykul = "Recenzja \"Władcy Pierścieni\"."

# lub tak
artykul = """Recenzja "Władcy Pierścieni"."""
```

> Spacje w wyrażeniach i definicjach są pożądane, ale nie należy ich nadużywać.

Listing 11

```
dobrze: zakupy(szynka[1], {jajka: 2})
    x = 1
    lista[index]
    lista[1:4]

źle:    zakupy( szynka[ 1 ], { jajka: 2 } )
    x=1
    lista [index]
    lista[1: 4]
```

Wszystkie operatory binarne powinny być otoczone pojedynczą spacją.