Lista 5: Struktury danych. Pakiety i moduły

Witold Dyrka

Maj 2021

Rozwiązania zadań, o ile nie wskazano inaczej, proszę przedstawić w formie kodu źródłowego w Pythonie (plik tekstowy z rozszerzeniem `.py`). Pliki z kodem źródłowym należy umieszczać w kontenerze odbiorczym bezpośrednio (pliki spakowane nie będą akceptowane).

Każdy plik z kodem źródłowym proszę opatrzyć komentarzem informującym o autorstwie. Powyższy zapis jest traktowany jako równoważny oświadczeniu, że kod został napisany samodzielnie. W przeciwnym przypadku komentarz powinien określać rodzaj i zakres udziału zewnętrznego oraz precyzyjnie wskazywać jego źródła. Umiejętne *oraz* dobrze udokumentowane korzystanie ze źródeł zewnętrznych nie obniża wartości samego rozwiązania, jednak nadmierne lub, co gorsze, bezrefleksyjne użycie materiałów zewnętrznych może negatywnie wpłynąć na proces nauki programowania.

Kod powinien być zredagowany zgodnie z zasadami przedstawionymi na wykładzie. Dobre praktyki dotyczące redagowania kodu w Pythonie opisuje dokument *PEP8 – Style Guide for Python Code*[1].

Ponadto dla każdej z funkcji implementujących algorytm napisz funkcję testującą, która na kilku przykładach zademonstruje poprawne działanie testowanej funkcji. Funkcja testująca powinna wyświetlać czytelne komunikaty. Do automatyzacji weryfikacji poprawności warto wykorzystać asercje.

Przesyłany do oceny kod źródłowy powinien być opatrzony komentarzami. Ogólne zasady tworzenia komentarzy dokumentacyjnych w Pythonie zawiera dokument *PEP257 – Docstring Conventions*[2]. Polecamy trzymać się stylu komentarzy dokumentacyjnych Google[3] albo NumPy[4]. Porównanie obu stylów znajdziemy w dokumentacji generatora dokumentacji Sphinx[5].

[1] https://www.python.org/dev/peps/pep-0008/

[2] https://www.python.org/dev/peps/pep-0257/

[3] https://google.github.io/styleguide/pyguide.html

[4] https://numpydoc.readthedocs.io/en/latest/format.html#docstring-standard

[5] https://www.sphinx-doc.org/en/master/usage/extensions/napoleon.html

Zad. 1

Wykład 6 przedstawia implementację stosu (klasa `Stos`).

a) Zapisz ją w postaci modułu `stos`.

b) Uzupełnij moduł `stos` o komentarze dokumentacyjne.

c) Uzupełnij moduł `stos` o testy — dla każdej funkcji co najmniej trzy. Testy powinny stanowić część modułu i być uruchamiane tylko w przypadku wykorzystania modułu jako głównego skryptu programu, co osiągniemy wywołując je pod warunkiem: `if __name__ == "__main__"`.

d) Wykorzystaj moduł `stos` do napisania funkcji sprawdzającej poprawne rozmieszczenie nawiasów „okrągłych" w zadanym jako parametr funkcji napisie. Funkcja powinna zwracać wartość logiczną `True` jeśli każdemu nawiasowi otwierającemu (`(`) odpowiada nawias zamykający (`)`), albo zwracać `False` — w przeciwnym przypadku. Znaki niebędące nawiasami „okrągłymi" powinny zostać pominięte podczas analizy.

   Na przykład zapis: `max(a, max(b, c))` jest poprawny, a zapisy: `max(a, max(b,c)` lub `max(a, b, c))` — nie.

   Funkcję należy opatrzyć komentarzami dokumentacyjnymi oraz co najmniej trzema testami poprawności.

e) Wykorzystaj moduł `stos` do napisania funkcji sprawdzającej poprawne rozmieszczenie potencjalnie pomieszanych nawiasów trzech typów: „okrągłych", „kwadratowych" i „klamrowych" w zadanym jako parametr funkcji napisie. Funkcja powinna zwracać wartość logiczną `True` jeśli każdemu nawiasowi otwierającemu danego typu odpowiada nawias zamykający tego samego typu oraz nawiasy nie krzyżują się, albo zwracać `False` — w przeciwnym przypadku. Znaki niebędące nawiasami wymienionych typów powinny zostać pominięte podczas analizy.

   Na przykład zapis: `max{a*[b+(1/c)], e/[1+f]}` jest poprawny, a zapisy: `max{a*[b+(1/c], e/[1+f]}` lub `max{a*[b+(1/c], e/[1+f])}` — nie.

   Funkcję należy opatrzyć komentarzami dokumentacyjnymi oraz co najmniej trzema testami poprawności.

   Poprawne rozwiązanie niniejszego podpunktu zawiera w sobie rozwiązanie podpunktu poprzedniego.

Uwaga! Przedmiotem zadania jest wykorzystanie implementacji stosu przedstawionej na wykładzie, a nie dowolnej innej implementacji.

Zad. 2

Wykład 6 przedstawia implementację listy dynamicznej (klasa `Lista`). Wykonaj następujące modyfikacje tej implementacji — tworząc moduł o nazwie `lista2`:

a) Przechowywanie odnośnika do ostatniego elementu listy ma swoje zalety, ale nie jest konieczne dla zapewnienia wszystkich istotnych funkcjonalności tej struktury. Zrezygnuj z atrybutu `koniec` listy i dostosuj odpowiednio wszystkie funkcje obsługujące tę strukturę.

b) Rekurencyjna funkcja `nastepny` zwraca wskazany (względem bieżącego) element listy. Zaimplementuj wersję iteracyjną tej funkcji.

c) Napisz funkcję `dlugosc(lista)`, zwracającą liczbę elementów na liście.

d) Napisz funkcję `znajdz(lista, wartosc)`, zwracającą indeks elementu o zadanej wartości albo `None`, gdy takiego nie ma.

e) Napisz komentarze dokumentacyjne do wszystkich funkcji.

f) Dla każdej funkcji napisz przynajmniej trzy testy weryfikujące ich poprawność. Testy powinny stanowić część modułu i być uruchamiane tylko w przypadku wykorzystania modułu jako głównego skryptu programu, co osiągniemy wywołując je pod warunkiem: `if __name__ == "__main__"`.

Uwaga! Przedmiotem zadania jest modyfikacja implementacji listy przedstawionej na wykładzie, a nie tworzenie alternatywnej implementacji.

Zad. 3

Utwórz pakiet `struktury_dynamiczne` zawierający moduły `stos` i `lista2`. Utwórz skrypt demonstrujący użycie wymienionych wyżej modułów jako części utworzonego pakietu.

Ważne

Kompletne rozwiązanie listy powinno zawierać:

a) moduł `stos` w pliku `stos.py`,

b) osobny plik z funkcjami rozwiązującymi podpunkty d) i e) zadania 1,

c) moduł `lista2` w pliku `lista2.py`,

d) skrypt demonstrujący użycie pakietu `struktury_dynamiczne`,

e) a także: spakowany w formacie `zip`, `tar` lub `tar.gz` pakiet `struktury_dynamiczne`. Uwaga! Przesłanie spakowanego pakietu nie zwalnia z obowiązku przesłania tych samych modułów — jego składowych — w postaci niespakowanej (podpunkty a) i c)).