Instrukcja do zadań

Przed Tobą kilka(naście) prostych, krótkich, ale wymagających w niektórych przypadkach solidnego pogłówkowania, zadań. Poziomem trudności są one zbliżone do tych, które robiliśmy na zajęciach. Wprowadzają one jednak element, z którego nie korzystaliśmy w poprzedni weekend: testy. W katalogu zadania_domowe, znajduje się kilkanaście plików. Pierwsza połowa z nich to zadania, druga to wspomniane testy – pliki weryfikujące działanie Twojego kodu możesz poznać po nazwie: rozpoczynają się one od „test_" (zgodnie z zaleceniami).

Każde pojedyncze zadanie to jedna funkcja do napisania (nie należy zmieniać jej parametrów ani nazwy, ponieważ nie uda się wtedy zaliczyć testów). To, co ma ona robić, jest opisane w każdym pliku na samej górze poprzez użycie pythonowej dokumentacji kodu znanej jako docstring (polecam zapoznać się z tym zagadnieniem, nie ma tu wiele filozofii, a na „rozgrzewkę" czasami rekruterzy pytają, co to takiego). Przykładowy docstring z pliku ukrywanie_hasla.py został zaprezentowany na poniższym obrazku.

```
"""
    Napisz prosta funkcje "szyfrujaca". Jej zadaniem jest zamiana
    co trzeciego znaku w hasle na znak gwiazdki (*).
    Przyklad:
    >> x = hide_password("moje_super_tajne_haslo123")
    >> print(x)
    "mo*e_*up*r_*aj*e_*as*o1*3"
    Pamietaj, ze dlugosc napisu nie musi byc podzielna przez 3.
"""
```

Dzięki temu, że cały kod opakowany zostanie w funkcje, będziemy mogli importować go (poprzez import samych funkcji) do innego pliku i tam przetestować. W programistycznym świecie staramy się nigdy nie mieszać kodu produkcyjnego z testami – stawiamy między nimi grubą krechę i nierzadko są one nie tyle co odseparowane plikami, co mogą znajdować się nawet kilka katalogów (pakietów) dalej. My skorzystamy z najprostszej opcji: testy to inny plik, ale ten sam katalog, bez udziwnień.

Jeżeli zajrzysz do obojętnie jakiego pliku testowego, zobaczysz że napisany on został z wykorzystaniem modułu unittest – wbudowanej biblioteki pythonowej (polecam przypomnieć sobie materiał z bloku TDD). Oto jak wygląda przykładowy kod z testami. Poniżej postaram się opisać pokrótce, co jest do czego (w praktycznym podejściu, rzecz jasna).

1. importy – najpierw import z biblioteki standardowej Pythona, potem import naszej funkcji z danego pliku, którą chcemy przetestować.
2. definicja klasy testowej – klasa TestCase, zdefiniowana w bibliotece unittest, posiada wiele funkcji i potrafi automatycznie „zarządzać" testami. Chcemy, żeby nasza klasa też to potrafiła, z tego też powodu dziedziczymy po klasie TestCase z moduły unittest. Od tego momentu nasza klasa będzie miała dostęp do asercji (np. assertEqual, assertTrue, assertFalse). Jeżeli nie pamiętasz, na co są asercje, polecam ponownie powtórzyć sekcję testowania. Odsyłam również do oficjalnej dokumentacji modułu unittest: https://docs.python.org/3.7/library/unittest.html.
3. metoda testowa – każda metoda testowa powinna się zaczynać od „test_". Jest to narzucone odgórnie przez zasady biblioteki unittest, z której tu korzystamy. Jeżeli poprosimy ją o uruchomienie naszych testów, to poszuka ona metod rozpoczynających się od tej właśnie nazwy i je uruchomi. Dzięki temu możemy zdefiniować sobie funkcje pomocnicze nierozpoczynające się od „test_": nie zostaną one przez bibliotekę potraktowane jako testy i nie zostaną uruchomione. Każda metoda powinna sprawdzać tylko jeden przypadek. Jeżeli będziesz mieć problem z zadaniem, zawsze możesz potraktować testy jako dodatkową dokumentację i zobaczyć, czego oczekują od naszego kodu. Analizując test można się wiele dowiedzieć o działającej aplikacji (lub o wymaganiach do niej, tak jak jest w przypadku naszych zadanek).
4. odpalenie testów – jeżeli plik testowy zostanie odpalony jako program (czyli komendą python <ścieżka_do_pliku>), instrukcja unittest.main() sprawi, że biblioteka unittest rozpocznie procedurę testową i uruchomi wszystkie metody w naszej klasie, których nazwy rozpoczynają się od „test_". Z tego korzystamy chcąc odpalić nasze testy.

Procedura działania jest prosta: działamy prawie że w technice TDD (prawie, ponieważ masz dostępne wszystkie potrzebne testy, normalnie w TDD pisze się jeden test, odpala się go, pisze kod produkcyjny na podstawie testu i tak w kółko). Każda funkcja będąca obiektem zadania ma w swoim ciele jedynie wyrażenie pass – musisz je zastąpić swoją implementacją. Po napisaniu kodu (albo kilka razy w trakcie pisania, zostawiam Ci to do wyboru – druga opcja lepsza ☺) należy uruchomić testy sprawdzające poprawność rozwiązania. Jeżeli wszystkie zakończą się sukcesem (wszystkie będą na PASS), masz 95% pewności, że Twój kod jest poprawny. Gdyby coś notorycznie nie chciało przechodzić, a miał(a)byś pewność, że Twój kod jest poprawny, śmiało pisz na ap.lukaszpaluch@gmail.com. Nie należy oszukiwać i zmieniać testów ☺ Gdybyś poczuł(a) potrzebę dodania paru nowych, nie ma żadnego problemu, a nawet zachęcam do realizacji takiego pomysłu.

Nic nie stoi na przeszkodzie, by dodatkowo testować kod poprzez dopisanie w pliku z zadaniem linijki if __name__ == '__main__': i uruchomienie w tym bloku Twojej funkcji. Testy z danego pliku uruchomisz poprzez komendę „python <sciezka_do_pliku>", natomiast wszystkie testy z każdego pliku da się odpalić wklepując w terminalu „python -m unittest discover" - należy być w katalogu, w którym znajdują się zadania i testy. Przy pierwszym uruchomieniu testów, żaden nie powinien zostać zaliczony. Wraz z uzupełnianiem przez Ciebie funkcji, stopniowo powinny one zacząć przechodzić.

Przykładowe rozwiązania zostaną umieszczone na tym samym repozytorium nie wcześniej niż pod koniec następnego tygodnia. Naprawdę zachęcam do samodzielnego rozwiązywania zadań. Powodzenia!