Programowanie
po 8 wykładzie

Andrzej Giniewicz

26.04.2024

W tym tygodniu zajmujemy się testowaniem oprogramowania z wykorzystaniem dwóch bibliotek Pythona przeznaczonych do tego celu — **pytest** oraz **hypothesis**.

1 Na początek ćwiczenie

Zacznijmy od stworzenia katalogu na projekt. Po utworzeniu wchodzimy do pustego folderu i uruchamiamy Visual Studio Code wewnątrz. Można również skorzystać z linii komend i wydać komendę

```
code ścieżka_do_katalogu
```

Po uruchomieniu, Visual Studio Code zapyta, czy ufamy twórcom tego katalogu[1]. Zaznaczamy, że tak. Dzięki temu mamy stworzony projekt.

Otwieramy terminal wewnątrz Visual Studio Code i wydajemy polecenie, które stworzy wirtualne środowisko Pythona. W tym wirtualnym środowisku możemy doinstalowywać pakiety, bez ryzyka, że pojawią się jakieś konflikty.

```
python -m venv .venv
```

W projekcie pojawi się katalog o nazwie `.venv`. Nie powinniśmy z niego korzystać ręcznie. Jeśli korzystamy z Power Shell na Windowsie (nie z BASH, ZSH lub CMD.exe), powinniśmy wydać zgodę na wykonywanie skryptów na tym komputerze komendą

```
Set-ExecutionPolicy Unrestricted -Scope CurrentUser
```

Zamykamy terminal (ikonka kosza, nie wystarczy zamknąć widoku krzyżykiem, ponieważ terminal wciąż będzie otwarty). Otwieramy paletę poleceń (z menu „Wyświetl” lub odpowiednim dla naszego systemu skrótem klawiaturowym) i zaczynamy wpisywać

```
Python: Wybierz wersję interpretera
```

[1] Więcej możesz przeczytać w dokumentacji `https://code.visualstudio.com/docs/editor/workspace-trust`.

Na liście odnajdujemy wersję Pythona zawierającą w nazwie katalog `.venv`, aby powiedzieć Visual Studio Code, że będziemy korzystać ze wskazanego wirtualnego środowiska. Od teraz, gdy otwieramy nowy terminal, przed ścieżką w każdej linii powinno pojawiać się `(.venv)` na znak tego, że prawidłowo skonfigurowaliśmy środowisko. Inną metodą sprawdzenia, jest wykonanie w terminalu wewnątrz Visual Studio Code komendy

```
python -c "import sys; print(sys.prefix)"
```

W odpowiedzi powinniśmy zobaczyć ścieżkę do katalogu `.venv`, który powstał podczas tworzenia wirtualnego środowiska. Jeśli do tego momentu coś poszło nie tak, nie przechodź dalej, tylko zapytaj prowadzącego lub umów się na konsultacje.

Teraz wewnątrz środowiska zainstalujemy najnowszą wersję biblioteki **hypothesis**, która będzie nam potrzebna do przeprowadzenia niektórych testów.

```
pip install hypothesis
```

Jeśli `pip` zasugeruje aktualizację, lepiej jej nie wykonywać z poziomu Visual Studio Code, tylko poczekać na aktualizację pochodzącą z Anacondy. Poprawność instalacji `hypothesis` możemy sprawdzić wykonując komendę

```
python -c "import hypothesis; print(hypothesis.__version__)"
```

Jeśli zobaczymy numer wersji, wszystko poszło dobrze. Jeśli nie, to należy prześledzić komunikaty o błędzie i zobaczyć, czy znajduje się w nich podpowiedź, co poszło nie tak. Jeśli wszystko poszło dobrze, możemy zamknąć terminal w Visual Studio Code.

W eksploratorze plików w Visual Studio Code klikamy ikonkę dodawania nowego pliku. Nazywamy go `operacje.py`. Wpisujemy kod

```
def dodawanie(x, y):
    return x+y
```

który będzie służył jako minimalny kod, który możemy testować. Podczas tworzenia, zwróć uwagę, że na pasku na dole oprócz wersji Pythona powinien pojawić się katalog `(.venv)`.

Teraz utworzymy moduł o nazwie `testy`. W tym celu tworzymy katalog o nazwie `testy` i w środku pusty plik o nazwie `__init__.py`. Po lewej klikamy ikonę kolby stożkowej[2], jednego z rodzajów szkła laboratoryjnego. Pojawią się dwa duże niebieskie przyciski. Klikamy „Configure Python Tests", następnie wybieramy „pytest" oraz „. Root directory". Jeśli czegoś nam brakuje, Visual Studio Code doinstaluje odpowiednie programy w środowisku wirtualnym. Przyciski zostaną na miejscu, jednak sprawa zmieni się, gdy dodamy pierwszy test. W katalogu `testy` stwórzmy plik o nazwie `operacje_test.py` i wewnątrz funkcję

```
from operacje import dodawanie

def test_dodawanie_przemienne():
    assert dodawanie(1, 2) == dodawanie(2, 1)
```

[2] Patrz `https://pl.wikipedia.org/wiki/Kolba_stożkowa`.

Zaraz po zapisaniu pliku, w zakładce z kolbą stożkową pojawi się rozwijana lista, w której na końcu znajdować się będzie właśnie dodana nazwa funkcji. W samym pliku z testem, na lewo od definicji funkcji będzie zielony przycisk „play". Jeśli go klikniemy, test zostanie uruchomiony. Klikając dwa przyciski „play" nad rozwijaną listą testów, uruchomimy wszystkie testy. Po kliknięciu, obok nazwy testu pojawi się zielony znaczek potwierdzenia, że nasz test zadziałał[3].

Sprawdzimy jeszcze, czy działają testy z **hypothesis**. Zmodyfikujmy plik z testem tak, aby wyglądał następująco.

```
from operacje import dodawanie
from hypothesis import given, strategies as st


@given(st.integers(), st.integers())
def test_dodawanie_przemienne(x, y):
    assert dodawanie(x, y) == dodawanie(y, x)
```

Uruchommy testy ponownie i upewnijmy się, że obok testu pojawi się zielony znaczek z potwierdzeniem, że wszystko poszło dobrze.

Zwróćmy uwagę, że tym razem nie musieliśmy wymyślać przykładu, tylko zapisaliśmy regułę, że funkcja jest przemienna dla dowolnych liczb całkowitych. Zepsujmy teraz kod. Wróćmy do pliku `operacje.py` i zmieńmy definicję dodawania na

```
def dodawanie(x, y):
    return x-y
```

Po uruchomieniu testu, zobaczymy czerwony znaczek z nieprawidłowym wynikiem. Gdy przejdziemy do testu i klikniemy czerwoną linię, zobaczymy informacje, dla jakiego przypadku coś poszło nie tak oraz jaki problem pojawił się w kodzie. Tutaj zobaczymy informację

```
./testy/operacje_test.py::test_dodawanie_przemienne Failed:
[undefined]assert -1 == 1
 +  where -1 = dodawanie(0, 1)
 +  and   1 = dodawanie(1, 0)
```

co oznacza, że `assert` sprawdzał, czy wartości są sobie równe, ale otrzymał `-1` i `1`, przy czym `-1` było wynikiem działania komendy `dodawanie(1, 0)`, natomiast wynik `1` był wynikiem kodu `dodawanie(0, 1)`. Dalej w informacji zobaczymy, w którym miejscu i przy jakich argumentach to się stało

```
testy\operacje_test.py:6:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

[3]W razie problemów sprawdź dokumentację na stronie `https://code.visualstudio.com/docs/python/testing`.

```
x = 0, y = 1

    @given(st.integers(), st.integers())
    def test_dodawanie_przemienne(x, y):
>       assert dodawanie(x, y) == dodawanie(y, x)
```

Oznacza to, że **hypothesis** znalazło problem w teście w pliku `testy\operacje_test.py` w linii 6, w szczególności — dwie liczby całkowite, dla których warunek

```
dodawanie(x, y) == dodawanie(y, x)
```

w linii zaznaczonej symbolem > nie był spełniony i są to $x = 0$ oraz $y = 1$.

Dysponując taką informacją, o wiele łatwiej zidentyfikować problem i szukać dalszych błędów, ponieważ możemy prześledzić dokładnie ten konkretny przypadek, w którym $x = 0$ i $y = 1$, i zobaczyć, czy wszystko dobrze zaimplementowaliśmy. W naszym przypadku przekonamy się, że pomyliliśmy znak + z -. Po usunięciu błędu w kodzie testy znów przechodzą bez błędów.

2 Testy jednostkowe i biblioteka pytest

Testy jednostkowe (ang. *unit test*) są jednym z podstawowych narzędzi zapewnienia jakości kodu. Testy jednostkowe to testy niepodzielnej funkcjonalności, czyli niewielkich fragmentów kodu, w których sprawdzamy, czy kod działa zgodnie z oczekiwaniami. Testy są odseparowane od właściwego programu, aby nie obciążać każdego wykonania dodatkowymi komendami. Zazwyczaj testy powstają w czterech momentach:

1. zanim powstanie dana funkcjonalność,
2. w trakcie powstawania danej funkcjonalności,
3. po powstaniu danej funkcjonalności, ale przed jej udostępnieniem klientowi,
4. po zgłoszeniu błędu do danej funkcjonalności.

Tworzenie testów przed funkcjonalnością nazywamy TDD (ang. *Test-Driven Development*). TDD jest podejściem, w którym testy pełnią funkcję specyfikacji. Zanim zabierzemy się za zaprogramowanie funkcji, piszemy testy. Dzięki temu musimy na tym etapie wymyślić, jak funkcja będzie działała, bez myślenia o tym, w jaki sposób to działanie zostanie osiągnięte. Implementujemy dodawanie? Powinno być przemienne, powinno być łączne, zero powinno być elementem neutralnym tej operacji, powinno dać się dodać dwie dowolne liczby, i tak dalej. Implementujemy maksimum elementów na liście? Dzięki temu, już podczas pisania testu, zastanowimy się, co się powinno stać, gdy otrzymamy listę pustą. Czy zwracamy `None`, czy może wyjątek? To pytania, na które musimy odpowiedzieć bardzo

wcześnie, gdy stosujemy TDD. Dzięki temu, gdy siadamy do implementowania funkcji, mamy już wszystko przemyślane i zwykle kod będzie o wiele czystszy. Osoby chętne zgłębić technikę TDD zachęcam do zapoznania się z drugim wydaniem książki Harry'ego Percivala „Test-Driven Development with Python", wyd. O'Reilly[4].

Testy mogą również powstawać podczas pisania kodu. Gdy eksperymentujemy, uruchamiamy kod i okazuje się, że nie działa, możemy wpaść na pomysł, dla jakiego przypadku coś dziwnego się wydarzyło. W ten sposób generujemy nowe testy. Testy pisane przed i w trakcie pisania kodu, zwykle implementowane są przez tę samą osobę, która pisze kod funkcji. Po napisaniu kodu wciąż mogą powstawać nowe testy, choć wtedy (przynajmniej w firmach), często robione są przez inny dział lub praktykantów, którzy pisząc testy, zapoznają się z kodem. Jeśli autor kodu napisał testy przed lub w trakcie powstawania funkcji, najlepiej, jeśli dalsze testy odda komuś innemu, kto świeżym okiem spojrzy na problem.

Ostatnim momentem, gdy powstają testy, jest czas po zgłoszeniu błędu. Jeśli użytkownik zgłasza błąd, w którym pisze, że coś nie zadziałało w konkretnej sytuacji, pierwszym zadaniem programisty jest zwykle przygotowanie testu, który obrazuje problematyczny przypadek. Następnie pracujemy nad poprawką błędu tak długo, aż testy będą przechodzić. Po usunięciu błędu test pozostaje w kodzie, dzięki czemu, jeśli z jakiegoś powodu błąd wróci, będziemy o tym od razu wiedzieli. W ten sposób rozwija się na przykład pakiet matematyczny Sage[5], w którym przed wydaniem nowej wersji, wszystkie testy muszą przejść bez błędów — w tym przynajmniej po jednym teście na każdy zgłoszony w historii istnienia pakietu błąd. Takie rozwiązanie jest sporym zapewnieniem jakości oprogramowania.

Testować warto nie tylko w dużych projektach. Nawet w małym projekcie, w którym pracujemy sami lub w kilka osób, o rozmiarze takim, jak projekt zaliczeniowy na kursie z zaawansowanych metod programowania, jest to korzystne. Niekiedy pozornie niewinna zmiana w jednym miejscu kodu powoduje, że w zupełnie innym miejscu, coś przestaje działać. Jeśli mamy napisane testy, zaraz po wprowadzeniu zmiany, jesteśmy w stanie uruchomić testy i upewnić się, że jednak nic nie namieszaliśmy.

Patrząc na kody z przykładu, do pisania testów używamy instrukcji `assert`. Zgłasza ona błąd, gdy warunek występujący po komendzie nie jest spełniony. Instrukcja ta jest częścią języka Python, po co nam zatem biblioteka **pytest**? Możemy wyszczególnić kilka powodów:

- **pytest** zajmuje się zbieraniem oraz uruchamianiem testów — domyślnie znajduje wszystkie funkcje zaczynające się od `test_` oraz metod zaczynających się od `test_` w klasach zaczynających się od `Test` w plikach zaczynających się od `test_` lub kończących na `_test`;

- **pytest** uruchamia wszystkie testy, niezależnie od tego, który zgłosi błąd. Pojawienie się `assert` w normalnych warunkach przerywa działanie i powoduje zgłoszenie wyjątku, więc jeden błędny test, wyłączałby wszystkie dalsze testy bez odpowiedniego narzędzia do testowania;

[4] Patrz `https://learning.oreilly.com/library/view/test-driven-development-with/9781491958698/`.

[5] Darmowy pakiet matematyczny, porównywalny z pakietami takimi jak Matlab, Maple i Mathematica, wykorzystujący język Python jako podstawę do działania, patrz `https://www.sagemath.org/`.

- **pytest** pozwala na napisanie testu, który upewnia się, że zostaje zgłoszony wyjątek. Na przykład

```
from pytest import raises

def test_dzielenie_przez_zero():
    with raises(ZeroDivisionError):
        1 / 0
```

  nie zgłasza błędu, tylko kończy się prawidłowo, ponieważ spodziewamy się zgłoszenia wyjątku `ZeroDivisionError`. Wyjątek ten zostaje przechwycony, obsłużony i program kontynuuje działanie. Dla odmiany, gdyby `1 / 0` nie zgłosiło wyjątku lub zgłosiło inny wyjątek, test zakończyłby się niepowodzeniem.

Fakt, że **pytest** nie przerywa działania, gdy natrafi na wyjątek, nie powoduje, że wszystkie warunki możemy wrzucić do jednej funkcji, ponieważ jedna funkcja testu powinna sprawdzać jedną rzecz. Jeśli dana funkcja zakończy się błędem, wciąż pominiemy sprawdzanie dalszej jej części i przejdziemy do kolejnej funkcji testu. Jest to spójne z ideą testów jednostkowych, które powinny testować jedną, niewielką funkcjonalność.

3 Biblioteka hypothesis

Testy nie zawsze łatwo wymyślić. Biblioteka **hypothesis** implementuje tak zwane sprawdzanie własności. Opisujemy w niej za pomocą dekoratorów, jakie możliwe dane chcemy testować. W przykładzie z początku wykładu, sprawdzaliśmy test dla dwóch dowolnych liczb całkowitych. Następnie **hypothesis** generuje dużo losowych oraz brzegowych przypadków. Przez przypadki brzegowe, rozumiemy niestandardowe, rzadko spotykane wartości lub bardzo często spotykane wartości. Pozornie może się wydawać, że funkcja `dodawanie` działa prawidłowo i że test również jest napisany prawidłowo. Przypomnijmy, funkcja to

```
def dodawanie(x, y):
    return x+y
```

natomiast jej test to

```
@given(st.integers(), st.integers())
def test_dodawanie_przemienne(x, y):
    assert dodawanie(x, y) == dodawanie(y, x)
```

Dlaczego sprawdzamy dla liczb całkowitych? Dla zmiennoprzecinkowych też powinno być OK. Zmieńmy więc strategię.

```
LICZBA = st.one_of([st.integers(), st.floats()])

@given(LICZBA, LICZBA)
def test_dodawanie_przemienne(x, y):
    assert dodawanie(x, y) == dodawanie(y, x)
```

Okazuje się, że dla $x = 0$ oraz $y = NaN$, czyli wartości zmiennoprzecinkowej oznaczającej nieprawidłową liczbę (ang. *Not a Number*), nasze sprawdzenie warunku nie działa, w szczególności problemem jest porównanie w teście wartości NaN z NaN, ponieważ

```
float("NaN") != float("NaN")
```

zwraca wartość prawda, a

```
float("NaN") == float("NaN")
```

zwraca wartość fałsz (zgodnie ze specyfikacją liczb zmiennoprzecinkowych IEEE-754).

Ten przykład pokazuje, że nie zawsze możemy być w stu procentach pewni, że przewidzieliśmy wszystkie możliwości. Biblioteka **hypothesis** jest zatem bardzo pomocna.

4 Raport pokrycia

Raport pokrycia kodu testem pozwala na sprawdzenie, które warunki już przetestowaliśmy, a które nie. Aby móc wygodnie korzystać z raportów w Visual Studio Code, doinstalujemy dodatek „Coverage Gutters"[6]. Po zainstalowaniu dodatku, z terminala uruchamiamy również komendę

```
pip install pytest-cov
```

dzięki czemu, w wirtualnym środowisku będzie dostępny plugin do generowania raportów. Dodajmy nową funkcję do naszych operacji.

```
def ogranicz(x, a, b):
    if x<a:
        return a
    elif x>b:
        return b
    else:
        return c
```

(celowo umieściliśmy błąd w ostatniej linii). Dodajmy teraz test. Cały plik z testem będzie wyglądał następująco.

```
from operacje import dodawanie, ogranicz
from hypothesis import given, strategies as st


@given(st.integers(), st.integers())
def test_dodawanie_przemienne(x, y):
    assert dodawanie(x, y) == dodawanie(y, x)
```

[6] Patrz https://marketplace.visualstudio.com/items?itemName=ryanluker.vscode-coverage-gutters.

```
def test_ogranicz_lewa():
    assert ogranicz(-7, 0, 1) == 0


def test_ogranicz_prawa():
    assert ogranicz(7, 3, 4) == 4
```

Upewnijmy się, że wszystkie testy przechodzą, pomimo błędu w kodzie. Wejdźmy teraz do terminala i użyjmy komendy

```
pytest --cov=operacje --cov-report=xml .
```

Spowoduje to wygenerowanie raportu dla modułu `operacje.py` i zapisanie w formacie XML kompatybilnym z dodatkiem Coverage Gutters. Po wykonaniu komendy przechodzimy do pliku `operacje.py` i z palety komend wybieramy „Coverage Gutters: Display Coverage" lub używamy skrótu klawiaturowego podanego obok nazwy na liście komend. Zwróćmy uwagę, że na lewo od numerów linii kodu pojawią się kolory oznaczające poziom sprawdzenia kodu przez testy. W szczególności na lewo od linijki `return c` będzie czerwony kolor, co oznacza, że linia ta nie była ani razu uruchomiona w żadnym teście. Dzięki temu wiemy, że napisaliśmy niewystarczająco wnikliwe testy, aby wyłapać błąd dotyczący tej ścieżki uruchomienia. Z testem przygotowanym przez **hypothesis** będzie łatwiej. Jeśli ustawimy test w następujący sposób

```
from operacje import dodawanie, ogranicz
from hypothesis import assume, given, strategies as st


@given(st.integers(), st.integers())
def test_dodawanie_przemienne(x, y):
    assert dodawanie(x, y) == dodawanie(y, x)


@given(st.floats(allow_nan=False),
       st.floats(allow_nan=False),
       st.floats(allow_nan=False))
def test_ogranicz(x, a, b):
    assume(a <= b)
    y = ogranicz(x, a, b)
    assert a <= y <= b
```

uwzględniając, aby x, a oraz b były liczbami zmiennoprzecinkowymi innymi niż NaN oraz aby $a \leqslant b$, uzyskamy test pokazujący nasz błąd. Po naprawieniu błędu w ostatniej linii na

```
def ogranicz(x, a, b):
    if x<a:
```

```
        return a
    elif x>b:
        return b
    else:
        return x
```

uzyskamy przechodzący test oraz stuprocentowe pokrycie.