**3 PyTest - Mocki**

**Wstęp**

Po zapoznaniu się z podstawami testowania i zasadami pracy z PyTest-em, nadeszła pora na zagłębienie się w nieco bardziej niecodzienne i skomplikowane zagadnienia - mocki oraz mockowanie funkcjonalności aplikacji testowanej. Zdaję sobie sprawę, że mogą to być dla Ciebie nowe pojęcia, dlatego zatrzymajmy się na chwilę i w pierwszej kolejności odpowiedzmy sobie na pytanie, czym są wyżej wymienione pojęcia.

**Czym jest mock?**

Wyobraź sobie aplikację, która bada wahania kryptowalut na rynku. Na bieżąco śledzi ona, jak zmienia się w czasie cena tych wirtualnych aktyw, a na żądanie użytkownika, łączy się z API i zwraca przykłady takich pozycji, których wartość wzrosła najbardziej w ciągu ostatnich 3 miesięcy.

Przyznaj, że aplikacja jest ciekawa oraz może znacznie usprawniać efektywność handlowania wykupionymi zasobami. Jest jednak pewna bolączka dla programistów tworzących takie rozwiązanie. Załóżmy, że proces łączenia się z API i zwracania wyników jest dość czasochłonny, bo wyliczenie odpowiednich wartości, zajmuje około 6 sekund. Dla użytkownika takiej aplikacji nie jest to duży problem - naturalna kolej rzeczy przy wykonywaniu obliczeń na dużych zasobach danych. Programista jest jednak postawiony w nieco mniej korzystnej sytuacji - problemem jest bowiem uruchamianie testów takiej aplikacji. Otóż za każdym razem, chcąc sprawdzić pojedynczą funkcjonalność związaną z działaniem API, tester będzie zmuszony każdorazowo czekać przez 6 sekund, zanim aplikacja zwróci odpowiednią odpowiedź.   
Dobrze wiesz, że przy testowaniu powinniśmy bardzo skrupulatnie podchodzić do ilości tworzonych testów jednostkowych - im więcej tym lepiej - dlatego bardzo możliwe, że będziemy musieli kilkanaście razy uruchomić różne testy wykorzystujące API - a co za tym idzie, tester będzie musiał bardzo długo czekać na wykonanie wszystkich sprawdzeń… I tutaj właśnie problem rozwiązuje mockowanie obiektów!

Mock możesz bowiem wyobrazić sobie jako makietę danej funkcjonalności. Mockujemy po to, aby nie było konieczne wykonywanie całej operacji (i w rezultacie czasochłonne oczekiwanie na wynik), tylko ustalenie z góry, co ma się zadziać, gdy wywołamy, np. daną funkcję, czy skorzystamy z obiektu. Za pomocą mocka możesz definiować dowolne zachowanie programu.

Myślę, że najlepiej zrozumiesz to na przykładzie.

**Prosty przykład mockowania**

Wyobraź sobie, że chcesz otestować zachowanie funkcji do\_calculation(), która uruchamia pobieranie danych przez API. Problem stanowi właśnie długość operacji wykorzystujących API - załóżmy, że uzyskanie danych zajmuje 10 sekund (co zasymulujemy przez wykorzystanie funkcji time.sleep(10)). Taki program mógłby wyglądać następujaco:

| import time  def do\_calculation(value: int) -> int:  response = call\_api()  return response + value # function increases passed value by data got from API   def call\_api():  time.sleep(10) # simulation of doing some operations   return 100 |
| --- |

A test mógłby prezentować się następująco (oczekujemy w nim zwróconej wartości równej 105 dla value wynoszącego 5):

| def test\_compute():  expected = 105  actual = 5  assert do\_calculation(actual) == expected |
| --- |

No i właśnie problem pojawia się przy wykonaniu takiego testu. Sprawdzenie pojedynczego warunku będzie zajmowało około 10 sekund! Co więcej - takich prostych testów będziemy chcieli stworzyć o wiele więcej - konieczne może być bowiem sprawdzenie przypadków, gdy value jest zerem, jest wartością ujemną itd. Jednym słowem, 10 sekund to stanowczo za długi okres czasu dla wykonania się prostych testów.

Z racji tego, że nasze API zwraca zawsze tę samą wartość (100) - niezależnie od przesłanego value - rozsądnym usprawnieniem może właśnie okazać się wykorzystanie mocków. Moglibyśmy zamockować funkcję łączącą się z API (czyli stworzyć “makietę”, która zawsze będzie zachowywała się tak samo) i to z nią nawiązywać komunikację. Wówczas za każdym razem otrzymywalibyśmy określony output oraz zostalibyśmy zwolnieni z konieczności czekania 10 sekund, zanim zostaną wykonane odpowiednie wyliczenia. To właśnie zaleta tworzenia mocków.

**Zapamiętaj!**

| Mock to obiekt, którego używa się zamiast rzeczywistej implementacji w trakcie testów jednostkowych. Pozwala on na określenie jakich interakcji spodziewamy się w trakcie testów. |
| --- |

**Rozwiązanie problemu**

Tworzę ten podpunkt, aby podsumować to wszystko, co napisałem wyżej i jeszcze bardziej dobitnie pokazać Ci zalety wykorzystania mocków. Porównaj między sobą oba podejścia do testowania kodu:

1. W tym podejściu tworzymy testy sprawdzające poprawność wykonywania dowolnych funkcji, wykorzystujących call\_api(). Każdy z testów będzie wykonywał się przez okres około 10 sekund, aby sprawdzić każdą pojedynczą funkcjonalność.
2. Rozwiązanie nieco inne od wcześniejszego. Piszemy pojedynczy test sprawdzający poprawność zwracanych danych z call\_api (którego wykonanie zajmie 10 sekund), następnie call\_api mockujemy i tak stworzony obiekt wykorzystujemy w pozostałych testach (wykorzystujących call\_api). Dzięki temu każdy pozostały test, wykorzystujący mocka, wykona się natychmiastowo.

Dzięki powyższym przykładom, nie muszę Ci już chyba więcej udowadniać, dlaczego wykorzystanie mocków może tak bardzo usprawnić proces testowania. Większa szybkość i mniejsza złożoność procesu testowania, to główne zalety wynikające z mockowania obiektów.

Mając już tak solidne podstawy konceptu, możemy śmiało przejść do zapoznania się z tym, jak mockować obiekty w Pythonie.

**PyTest-Mock**

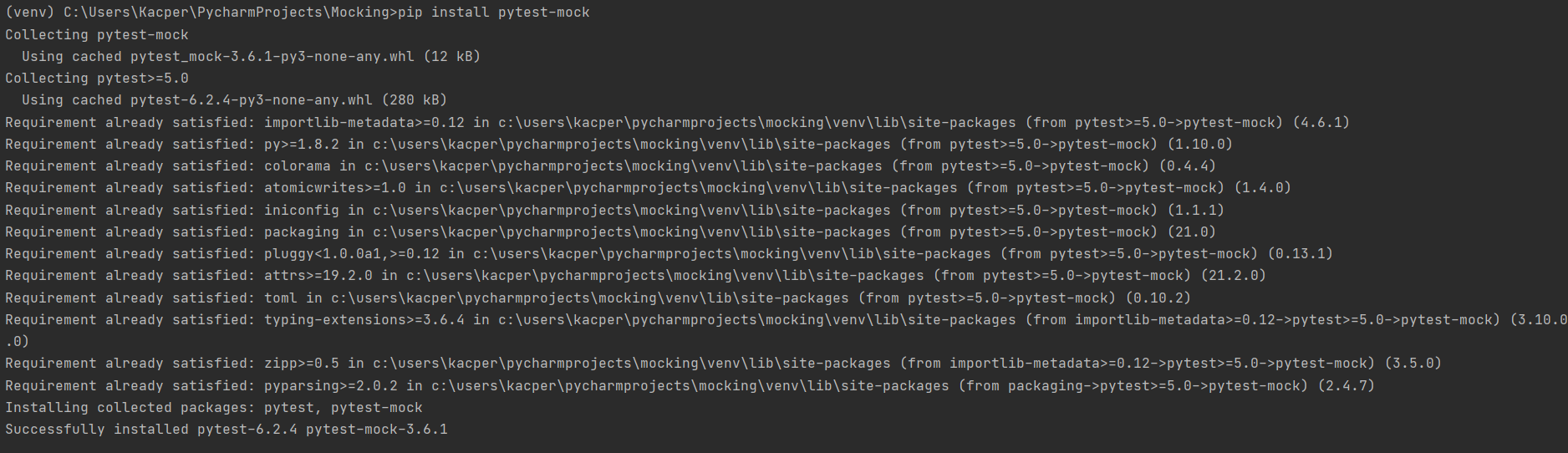
W celi mockowania funkcjonalności, wykorzystamy bibliotekę pytest-mock. Jest to paczka zintegrowana z modułem pytest, dlatego instalacja samego pytest-mock, spowoduje również dostarczenie paczki pytest.

Jak przeprowadzić proces instalacji pytest-a, omówiłem we wcześniejszym szkoleniu, dlatego zajrzyj tam w razie potrzeby.

**Polecenie**

| pip install pytest-mock |
| --- |

**Efekt**



*Instalacja pytest-mock*

**Mockowanie funkcji**

Poznamy kilka sposób na mockowanie różnych struktur, np. stałych, funkcji, metod czy całych klas.

Zacznijmy od nawiązania do wyżej przedstawionego przykładu związanego z API i funkcją do\_calculation. Zaprezentuję Ci, w jaki sposób stworzyć “makietę” funkcji call\_api.

Zacznijmy od wydzielania projektu na odpowiednie pliki. Przypominam, że kod produkcyjny aplikacji powinniśmy trzymać w osobnym pliku niż testy napisane do takiego oprogramowania.

Utwórzmy zatem plik application.py i umieśćmy w nim poznaną już wcześniej funkcjonalność:

| *# application.py*  import time  C |
| --- |

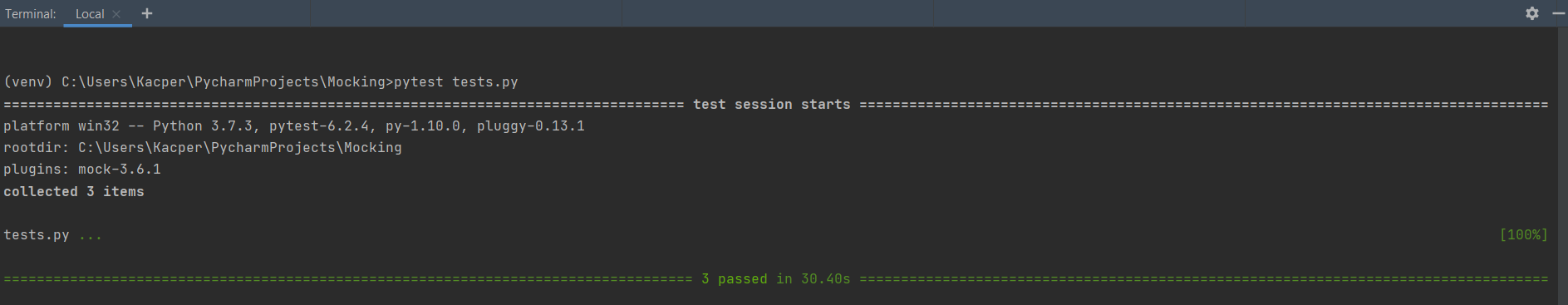
A do pliku tests.py wrzućmy następujące funkcje testujące:

| # tests.py  from application import do\_calculation  def test\_compute\_for\_positive\_value():  expected = 105  actual = 5  assert do\_calculation(actual) == expected  def test\_compute\_for\_negative\_value():  expected = 90  actual = -10  assert do\_calculation(actual) == expected  def test\_compute\_for\_zero():  expected = 100  actual = 0  assert do\_calculation(actual) == expected |
| --- |

Testy uruchomiłem poniższym poleceniem (znajdując się w katalogu z projektem):

| pytest tests.py |
| --- |

**Rezultat**



Wszystkie testy przeszły, ale… Zauważ, w jakim czasie się one wykonały. Aż 30 sekund! Zgodnie z oczekiwaniami - musieliśmy przecież trzykrotnie połączyć się z API, gdzie pojedyncze nawiązanie komunikacji zajmuje czas 10 sekund. Czas zatem na znaczne skrócenie tego czasu i wprowadzenie mocka za funkcję łączącą się z API!

**Testy z wykorzystaniem mocków**

Aby zamockować dowolną funkcję, która zawsze będzie zwracała określoną wartość, wykorzystamy następujący zapis:

| mocker.patch(‘<moduł.nazwa\_funkcji>’, return\_value=<returned\_value>) |
| --- |

Mock dla naszej funkcji będzie zatem wyglądał następująco:

| mocker.patch(‘application.call\_api’, return\_value=100) |
| --- |

I będzie umieszczany w pliku tests.py.

Po zmianie naszych testów, otrzymamy następujący rezultat:

| # tests.py from application import do\_calculation   def test\_compute\_for\_positive\_value(mocker):  expected = 105  actual = 5  mocker.patch('application.call\_api', return\_value=100)  assert do\_calculation(actual) == expected   def test\_compute\_for\_negative\_value(mocker):  expected = 90  actual = -10  mocker.patch('application.call\_api', return\_value=100)  assert do\_calculation(actual) == expected   def test\_compute\_for\_zero(mocker):  expected = 100  actual = 0  mocker.patch('application.call\_api', return\_value=100)  assert do\_calculation(actual) == expected |
| --- |

Zanim jednak przystąpię do dokładniejszego omówienia wykorzystanych mocków, spójrzmy, jak możesz zrefaktorować ten kod (tak aby tworzenie mocków umieścić w fiksturze, których koncept poznałeś we wcześniejszym szkoleniu):

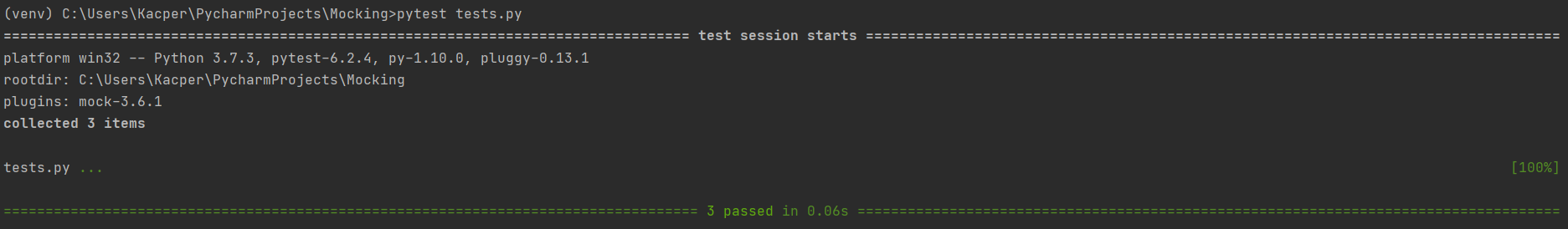
**Ten sam kod co wyżej, ale po refaktorze**

| # tests.py  import pytest from application import do\_calculation  @pytest.fixture def set\_api\_mock(mocker):  mocker.patch('application.call\_api', return\_value=100)  def test\_compute\_for\_positive\_value(set\_api\_mock):  expected = 105  actual = 5  assert do\_calculation(actual) == expected  def test\_compute\_for\_negative\_value(set\_api\_mock):  expected = 90  actual = -10  assert do\_calculation(actual) == expected  def test\_compute\_for\_zero(set\_api\_mock):  expected = 100  actual = 0  assert do\_calculation(actual) == expected |
| --- |

Stanowczo lepiej! To teraz zwróćmy uwagę na to, jak mockujemy elementy:

1. Gdy tworzymy funkcję wykorzystującą mock-a, musimy w pierwszej kolejności umieścić na jej liście parametrów słowo kluczowe mocker, np. def set\_api(mocker)
2. Chcąc zamockować konkretną już funkcję, zapiszemy mocker.patch('application.call\_api', return\_value=100)

Teraz, gdy uruchomimy te same testy…



… otrzymamy o wiele krótszy ich czas wykonania! Ze zbędnych 30 sekund wykonywania, doszliśmy do czasu równego 0.06 s.

**DAFQ?**

Zdaję sobie jednak sprawę, że w momencie, gdy spotykasz się z implementacją takiego rozwiązania, nie do końca jesteś w stanie zrozumieć, co dzieje się pod kurtyną i jak to tak naprawdę działa, że zamiast normalnej funkcji call\_api, wywoływany jest jej mock - bez konieczności czekania na rezultat.

Otóż wyobraź sobie to tak:

| Od momentu, gdy zostanie ustawiony mock na daną funkcję w określonym teście, program przestaje rozpatrywać każde wywołanie tej funkcji w postaci: “wykonaj wszystkie operacje w niej zawarte”, ale jedynie “wykonaj tylko te czynności, które zostały określone w mocku”. |
| --- |

Czyli, np. w tworzonej makiecie dla funkcji call\_api w powyższych przykładach, określiliśmy, że każde wywołanie call\_api() ma jedynie kończyć się zwróceniem wartości 100 (return\_value=100). Mock ten “przysłania” właściwą implementację funkcji i od tej pory brane jest pod uwagę jedynie takie zachowanie, jakie określiliśmy w mock.patch.

Zwróć również uwagę na to, że dany mock żyje jedynie w scopie określonego testu (funkcji) i chcąc wielokrotnie zastępować właściwą implementację funkcji daną jej “makietą”, musimy wiele razy stosować zapis mock.patch(). Stąd też umieszczenie fikstury wewnątrz naszych testów (musimy tworzyć mock na nowo przy każdym kolejnym teście).

**Mockowanie metody w klasie**

Po poznaniu idei mockowania, nadeszła pora na doszlifowanie naszej wiedzy i zahaczenie o pozostałe aspekty tworzenia makiet. Rozważmy teraz, jak zamockować dowolną metodę z klasy.

Zacznijmy od prostego przykładu.

Wyobraź sobie następujący plik z klasę i funcją, która z tej klasy korzysta:

| # application.py  import time  class Data:  def \_\_init\_\_(self):  self.data = None   def load\_data(self):  time.sleep(6)  self.data = 'some data'  def get\_data():  data = Data()  return data.load\_data() |
| --- |

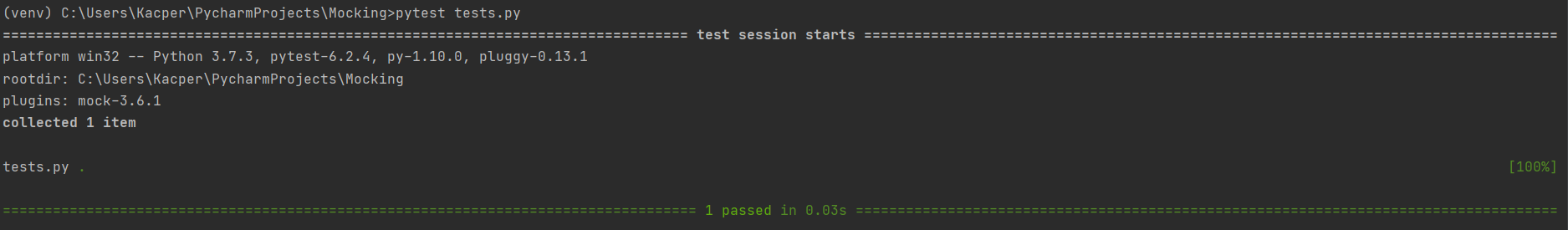
Jak już się zapewne domyślasz, będziemy chcieli mockować funkcjonalność metody load\_data. Wykonamy to w następujący sposób:

| from application import get\_data  def test\_mocking\_class\_method(mocker):  expected = 'some data'   def mock\_load\_data(self):  val = 'some'  return val + ' data' # it is implemented this "strange way" as to show we can mock whole method   mocker.patch('application.Data.load\_data', mock\_load\_data)   actual = get\_data()   assert expected == actual |
| --- |

Zwróć uwagę na 2 rzeczy:

1. Metodę load\_data zamockowaliśmy w dość niestandardowy sposób - tworząc jej całą makietę, a nie określając jedynie argument return\_value (jak to było we wcześniejszym przykładzie).
2. Chcąc odwołać się do oryginalnej metody, zastosowaliśmy zapis application.Data.load\_data, czyli <moduł>.<klasa>.<metoda>

**Uruchomienie testów**



**Mockowanie klas**

Załóżmy teraz, że chcemy zamockować całą klasę i na takiej makiecie wywoływać określone metody. Różnica między tym a wcześniejszym podejściem jest zasadnicza - nie będziemy chcieli w ogóle instancjonować naszej klasy. Takie rozwiązanie sprawdzi się w momencie, gdy tworzenie obiektu jest bardzo zasobożerne, a my jedynie chcemy wykorzystać jego niektóre metody w celu wykonania testów.

Rozważ poniższą klasę:

| # application.py  import time  class MyClass:  def \_\_init\_\_(self, some\_parameter: SomeObject) -> None:  self.some\_value = some\_parameter  time.sleep(6)   def get\_something(self) -> str:  return 'Hello' |
| --- |

Symulację “długiego procesu instancjonowania” obiektu przeprowadziłem przy użyciu biblioteki time. Tak więc utworzenie pojedynczego obiektu klasy zajmuje okres 6 sekund.

Plik tests.py będzie wyglądał następująco:

| def test\_with\_mock(mocker):  mock\_my\_class = mocker.patch('application.MyClass')  mock\_my\_class\_get\_sth = mocker.patch('application.MyClass.get\_something')  mock\_my\_class\_get\_sth.return\_value = 'Hello'   assert mock\_my\_class.get\_something() == 'Hello' |
| --- |

I to właśnie w funkcji test\_with\_mock zawarliśmy to, co najważniejsze w tym przykładzie. Pod zmienną mock\_my\_class utworzyliśmy makietę klasy MyClass, a w pozostałych liniach...

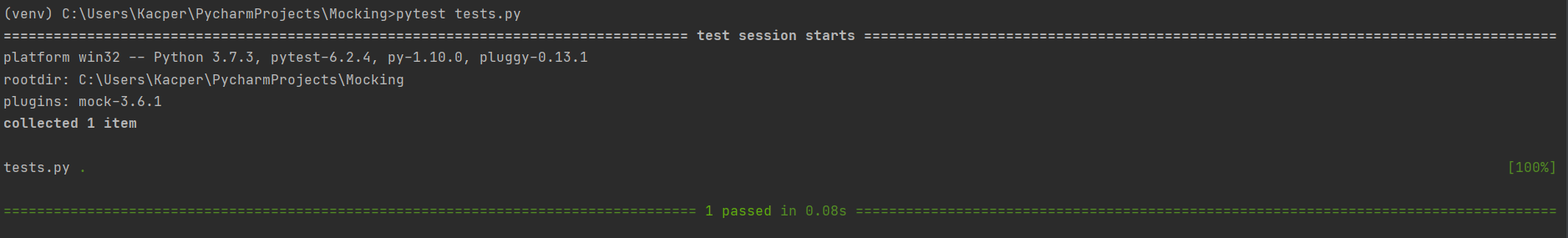
| mock\_my\_class\_get\_sth = mocker.patch('application.MyClass.get\_something')  mock\_my\_class\_get\_sth.return\_value = 'Hello' |
| --- |

… definujemy mocki dla jej metod. W tym przypadku nie tworzymy makiet pojedynczych metod, ale całej struktury klasowej. I to jest cała potęga naszego rozwiązania.

Jeżeli bowiem będziemy chcieli otestować dowolny obiekt jakiejś innej klasy, który instancjonowałby MyClass i musiałby czekać ponad 6 sekund na wykonanie kodu, wystarczy, że utworzymy właśnie mocka takiej klasy, określimy zachowanie poszczególnych metod takiej makiety i dopiero wyślemy tak przygotowany obiekt do klasy, która by z niego korzystała.

W gwoli ścisłości zmienne mock\_my\_class\_get\_sth oraz mock\_my\_class są, tzw. **MagicMocks** ([unittest.mock — mock object library — Python 3.9.6 documentation](https://docs.python.org/3/library/unittest.mock.html#unittest.mock.MagicMock)). Takie obiekty możemy w pełni dostosowywać do naszych potrzeb i wykorzystywać je, tak jakby były pełnoprawnymi obiektami. Dzięki utworzeniu MagicMock-a o nazwie mock\_my\_class, możliwe było wykonanie takiej linii: mock\_my\_class\_get\_sth.return\_value = 'Hello'.

Wynik testów



**Mockowanie stałych**

Najgorsze mamy za sobą! Poznaliśmy, jak mockować funkcje, metody oraz całe klasy. W pozostałych sekcjach tego szkolenia, chciałbym przedstawić Ci pozostałe drobiazgi związane z procesem mockowania. Nadszedł teraz czas na zastanowienie się, jak mockować stałe wartości.

Jeżeli zastanawiasz się, po co mockować stałe, to wyobraź sobie następujący przykład:

| *Twoja aplikacja dokonuje odczytu stałych konfiguracyjnych z jakiegoś obszernego pliku CSV. Z racji dużej ilości danych, proces czytania jest dość czasochłonny i zajmuje kilka sekund. Podczas testowania, nie ma sensu czekać tak długo na pobranie prostych danych. Warto wówczas zamockować stałe wartości (pobierane z pliku) - tak, aby testujący programista określił je “z palca” i nie było konieczne wykorzystanie zewnętrznych zasobów.* |
| --- |

W poniższym przykładzie zastosujemy nową metodę mocker.patch.object(). Umożliwia ona mockowanie konkretnych obiektów, np. stałych liczbowych - tak, aby przy odwoływaniu się do nich, była wstawiana dowolna wartość.   
  
Myślę, że dobrze zrozumiesz to na przykładzie.

**Plik constants.py**

| # constants.py  CONSTANT\_A = 10 |
| --- |

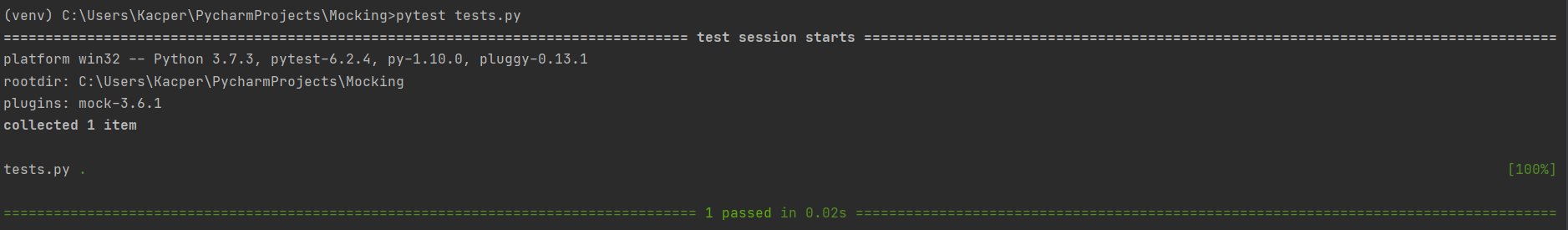
**Plik application.py**

| # application.py  from constants import CONSTANT\_A  def double():  return CONSTANT\_A \* 2 |
| --- |

**Plik tests.py**

| # tests.py  import application from application import double  def test\_mocking\_constant\_a(mocker):  mocker.patch.object(application, 'CONSTANT\_A', 2)  expected = 4  actual = double()   assert expected == actual |
| --- |

**Wynik testów**



Jeżeli zrozumiałeś, jak działa mockowanie w poprzednich sekcjach, to tutaj raczej nie powinieneś mieć problemu z odczytaniem logiki kodu.

Dzięki poniższej linii ...

| mocker.patch.object(application, 'CONSTANT\_A', 2) |
| --- |

… program rozumie, aby w każde odwołanie się do zmiennej CONSTANT\_A, wstawiać zadeklarowaną wartość równą 2.

**Podsumowanie**

W szkoleniu tym poznałeś, na czym polega i czym jest mockowanie dowolnych struktur w programie. Omówiłem przykłady, w których tworzyliśmy makiety stałych, funkcji, metod, a nawet całych klas. Choć może Ci się wydawać, że temat mocków jest zagadnieniem niezbyt często wykorzystywanym w praktyce, to uczulam Cię, że komercyjnie bardzo często korzysta się z tutaj omówionych rozwiązań. Dlatego niezwykle ważne jest, abyś dokładnie przeanalizował wszystkie przedstawione tutaj tematy i napisał pierwsze programy w oparciu o omówione zagadnienia.