**Pytest - początki**

**Wstęp**

Niezwykle się cieszę, że w końcu możemy przejść do pisania testów w praktyce. Bądź świadomy tego, że, jeżeli chodzi o Pythona, to do dyspozycji masz wiele różnych frameworków umożliwiających testowanie kodu produkcyjnego.

Poniżej przedstawiam Ci kilka z nich:

1. **Pytest** - wspiera testowanie jednostkowe, ale również rodzaje testowania wyższego poziomu - integracyjne, E2E. W kursie zajmiemy się poznawaniem zagadnień właśnie z jego zakresu.
2. **Robot Framework** - to ogólna platforma automatyzacji testów do testowania akceptacyjnego i rozwoju opartego na testach akceptacyjnych. Jest to struktura testowa oparta na słowach kluczowych, która wykorzystuje składnię danych testowych w formie tabel.
3. **PyUnit (Unittest)** - był początkowo zainspirowany frameworkiem JUnit z Javy, wspiera testy automatyczne, agregację testów w kolekcję i niezależność między testami.
4. **TestProject**
5. **Nose2**

**Instalacja i integracja z Pytest:**

**PyCharm:**

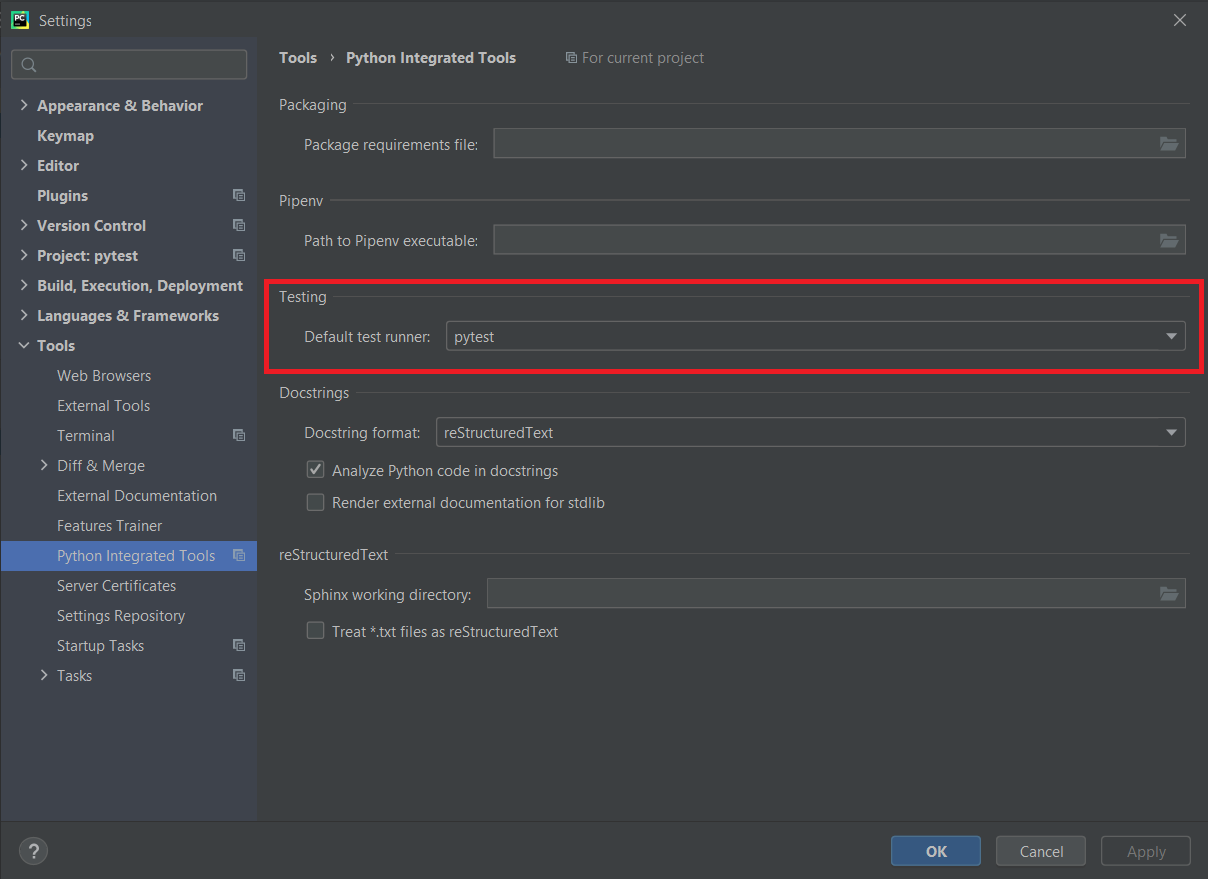
Jak będzie przebiegał proces instalacji omawianego frameworka, zależy od kilku czynników. Przede wszystkim od tego, w jakim środowisku programujesz oraz z jakiego systemu operacyjnego korzystasz.

Jeżeli jest to jedno z najbardziej popularnych środowisk - PyCharm - to wejdź w zakładkę instalacji zewnętrznych pakietów do Twojego virtualenv i zainstaluj bibliotekę pytest (dokładną instrukcję przeprowadzenia tej operacji znajdziesz w filmie wideo "Instalacja IDE, Debugger, IDE Review" dołączonego do kursu).

Po zakończonej instalacji, możesz korzystać już z biblioteki w swoim projekcie. **Jednak jest jeszcze jeden konieczny warunek do zrealizowania, aby móc uruchamiać Nasze testy automatyczne.**

Musimy w ustawieniach IDE poinformować środowisko, że korzystamy z modułu testującego pytest (będziemy mogli przez to automatycznie uruchamiać określone testy, a PyCharm nie będzie oczekiwał ich ręcznego wywołania w kodzie).

Wejdź w zakładkę **File** > **Settings** > **Tools** > **Python Integrated Tools** i w sekcji **Testing**, dla pozycji **Default test runner**, wybierz **pytest**.



**Zmiana domyślnego frameworka testującego**

**Instalacja z poziomu wiersza poleceń**

Jeżeli natomiast korzystasz z niezintegrowanego venv'a i chcesz zainstalować bibliotekę pytest, wykorzystaj system pip i wpisz:

pip install -U pytest

następnie sprawdź, czy proces instalacji przebiegł prawidłowo, sprawdzając, aktualną wersję modułu pytest:

pytest --version

> pytest 6.2.1

Po instalacji, aby uruchamiać już plik przechowujący dowolne testy, będziesz musiał wpisać:

pytest nazwa\_pliku.py -v

**Zasady tworzenia testów w pytest...**

I wcale nie chodzi mi o to, jakie są zasady pisania czystych testów (ten temat poruszyliśmy już we wcześniejszym szkoleniu). To, co chcę tutaj omówić, to jak poprawnie tworzyć **strukturę plików i funkcji**, które będą pełniły rolę testów.

| **Zapamiętaj!**  **Nazywanie plików**  Odnosząc się do konwencji poprawnego nazywania plików przechowujących testy, powinny one być tworzone w formacie: **test\_\*.py** lub **\*\_test.py** w katalogu głównym projektu lub jego podkatalogach (jeżeli tworzysz bardziej rozbudowany projekt).  Nie jest więc złym pomysłem, aby tworzyć testy w całkowicie **oddzielnym folderze danego projektu**. |
| --- |

Dzięki temu łatwo będziemy mogli jasno wydzielić kod produkcyjny od testującego. Przykłady spełniające to założenie, zobaczysz w następnych sekcjach omawianego tematu.

**Nazywanie funkcji testujących (testów jednostkowych)**

Jeżeli chodzi o nazywanie funkcji testujących (testów jednostkowych), to również istnieją niesamowicie istotne reguły dotyczące zasad ich nazewnictwa.

| Otóż każda funkcja (metoda) testująca, **musi zaczynać się od słowa test\_\*.** |
| --- |

Jest to warunek konieczny, ponieważ dzięki temu **moduł pytest**, wie, które funkcje **w danym projekcie są testami**, a które realizują działanie kodu, który **będzie testowany** (kod produkcyjny).

Zbierzmy całą wiedzę do kupy i stwórzmy pierwszy funkcjonalny test, który będzie spełniał wszystkie wymienione wyżej zasady i reguły.

**Założenia testu jednostkowego**

Sercem każdego testu będzie słowo kluczowe **assert**. To ono właśnie umożliwi Nam sprawdzenie, czy dana metoda (funkcja) sprawdzana przez test jednostkowy, zachowuje się **w oczekiwany sposób**.

Assert przeprowadza **mechanizm asercji** i to od niego zależy, czy test zakończy się **powodzeniem** czy nie.

**Syntax:**

Po omawianym słowie kluczowym, umieszczamy warunek, który chcemy sprawdzić, np.

assert 5 > 3 # (TEST PASSED)

assert 3 == 5 # (TEST FAILED)

**Tworzenie pierwszego testu i analiza logów**

**UWAGA:**

Realizując przykłady w tym temacie, bazuję na pracy z PyCharmem, ale jeżeli korzystasz z terminala lub uruchamiasz programy z poziomu linii poleceń, wykorzystuj wcześniej wspomniane poleceniepytest test\_compare.py -v w celu uruchomienia danego pliku z testami.

**Przykład:**

Wyobraź sobie, że musisz dostarczyć klientowi funkcje, które realizują dwie operacje matematyczne - **pierwsza to potęgowanie**, a druga nieco bardziej skomplikowana, bo obliczanie **sumy ciągu arytmetycznego o różnicy 1** dla dowolnego przedziału liczb całkowitych *(dla przypomnienia ciąg arytmetyczny o różnicy 1 to zbiór takich liczb, które zawierają się w przedziale [start, koniec], np. dla start = 1, koniec = 10 ciąg będzie miał postać 1, 2, 3, 4, 5, 6, 7, 8, 9, 10; suma dla konkretnego ciągu arytmetycznego to nic innego jak obliczenie sumy wszystkich jego wyrazów; suma dla powyższego przykładu to: 55).*

Zgodnie z założeniami **idei TDD**, zaczniemy od napisania pierwszego testu, który będzie sprawdzał wyniki implementowanej wkrótce funkcji **potęgującej**.

1. **Faza czerwona**

1. Tworzymy projekt o dowolnej nazwie, np. **math\_operations**

2. Wewnątrz projektu tworzymy **package**, w której będziemy umieszczali kod produkcyjny. Pakiet nazywamy: **functionality**.

3. Do projektu dodajemy jeszcze jedną paczkę, w której będziemy przechowywali testy, np. o nazwie **tests.py**.

4. W katalogu **functionality** możemy stworzyć plik, w którym będziemy umieszczali kod produkcyjny (np. o nazwie **operations.py**), jednak zgodnie z zasadą TDD (najpierw testy, później funkcjonalność), plik **pozostawiamy pusty**!

5. W katalogu tests tworzymy pierwszy plik o nazwie **test\_power** (pamiętaj, jak poprawnie nazywać pliki testujące pytest), który będzie przechowywał **funkcje testujące** dla **operacji potęgowania**.

6. W utworzonym pliku - test\_power - zaczynamy od pisania pierwszych funkcji testujących. Będą to kolejno testy sprawdzające: **poprawność wyniku dla wykładników będącymi całkowitymi wartościami dodatnimi**, **całkowitymi wartościami ujemnymi**, **ułamkami i wartością 0**.

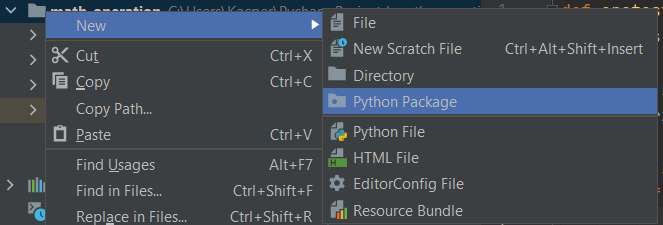
**UWAGA:**

1. Pamiętaj o zasadzie wstawiania słowa **test\_** przed każdą tworzoną funkcją testującą!

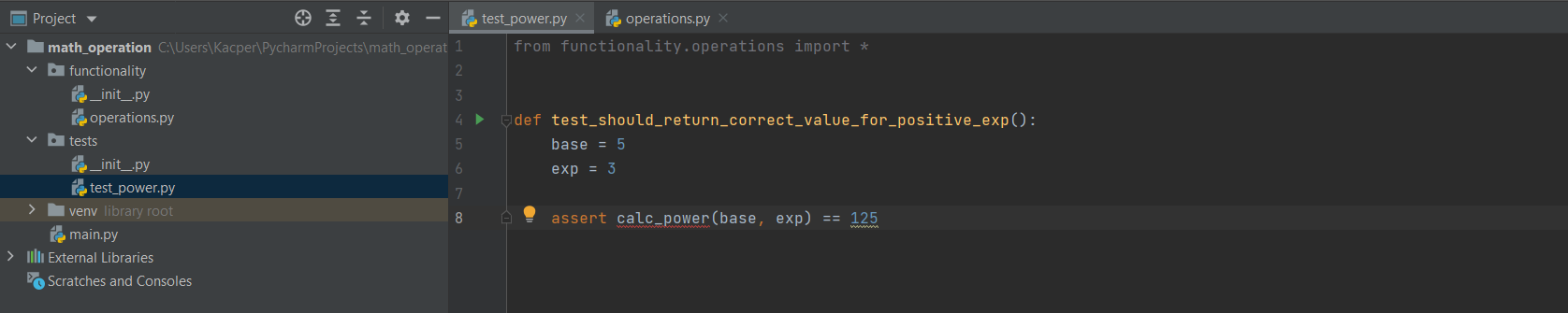
2. Zwróć również na 1. linię poniższego przykładu. Z racji tego, że funkcjonalność programu i testy jednostkowe umieszczone są w oddzielnych plikach, aby móc wywołać wkrótce zaimplementowaną przez Nas funkcję **calc\_power** z pliku **operations.py**, musimy umieścić w kodzie poniższą linię

**from functionality.operations import \***

która oznacza, że chcemy dać możliwość wywoływania dowolnej funkcji umieszczonej w **operations.py** z katalogu **functionality**.



**Utworzenie paczki na pliki w projekcie**

**Realizacja pdpkt 1 - 6**

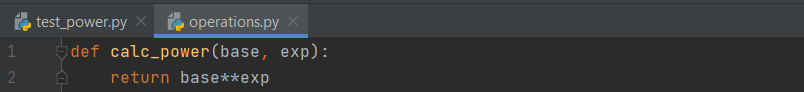
Tak stworzony test, możemy uruchomić klikając **przycisk interpretacji** umieszczony przy 4. linii kodu.

Jednak proces uruchomienia zakończy się błędem, ponieważ jesteśmy nadal w **fazie czerwonej TDD** i musimy zaimplementować poprawnie działającą funkcję **calc\_power** przyjmującą liczbę i jej dodatni wykładnik całkowity.

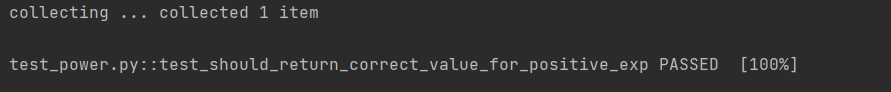
1. **Faza zielona**

Teraz nastąpił moment na zaimplementowanie funkcji testowanej, która realizowałaby założenia wcześniej napisanego przez Nas testu.

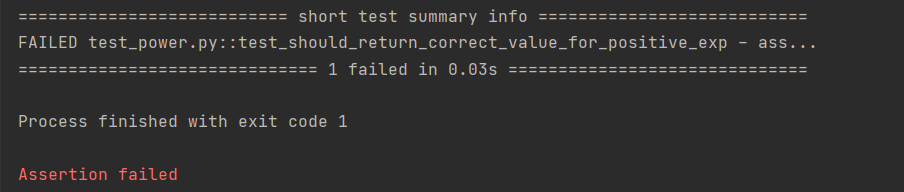
Przechodzimy do pliku **operations.py** znajdującego się w paczce **functionality** i przystępujemy do implementacji funkcji.



# Wracamy do pliku test\_power.py i ponownie uruchamiamy test. Voilà! Jeżeli zrobiłeś wszystko dobrze, to kod powinien wykonać się bez błędów i powinieneś ujrzeć komunikat o stuprocentowym zaliczeniu testu.

**test\_should\_return\_correct\_value\_for\_positive\_exp() zakończony sukcesem**

Gdyby natomiast, czysto hipotetycznie, test nie zakończył się powodzeniem, konieczne by było poprawienie logiki funkcji calc\_power (zakładając, że określiliśmy poprawne oczekiwania w teście).

**test\_should\_return\_correct\_value\_for\_positive\_exp() zakończony niepowodzeniem**

1. **Faza niebieska**

Nie zapominajmy o refaktorze kodu, jako końcowym kroku pojedynczego cyklu TDD.

Oczywiście z racji tego, iż Nasz przykład jest prosty i jego cele jest przede wszystkim zapoznanie Cię z biblioteką pytest, to Nasze przykłady nie wymagają żadnych poprawek stylistycznych.

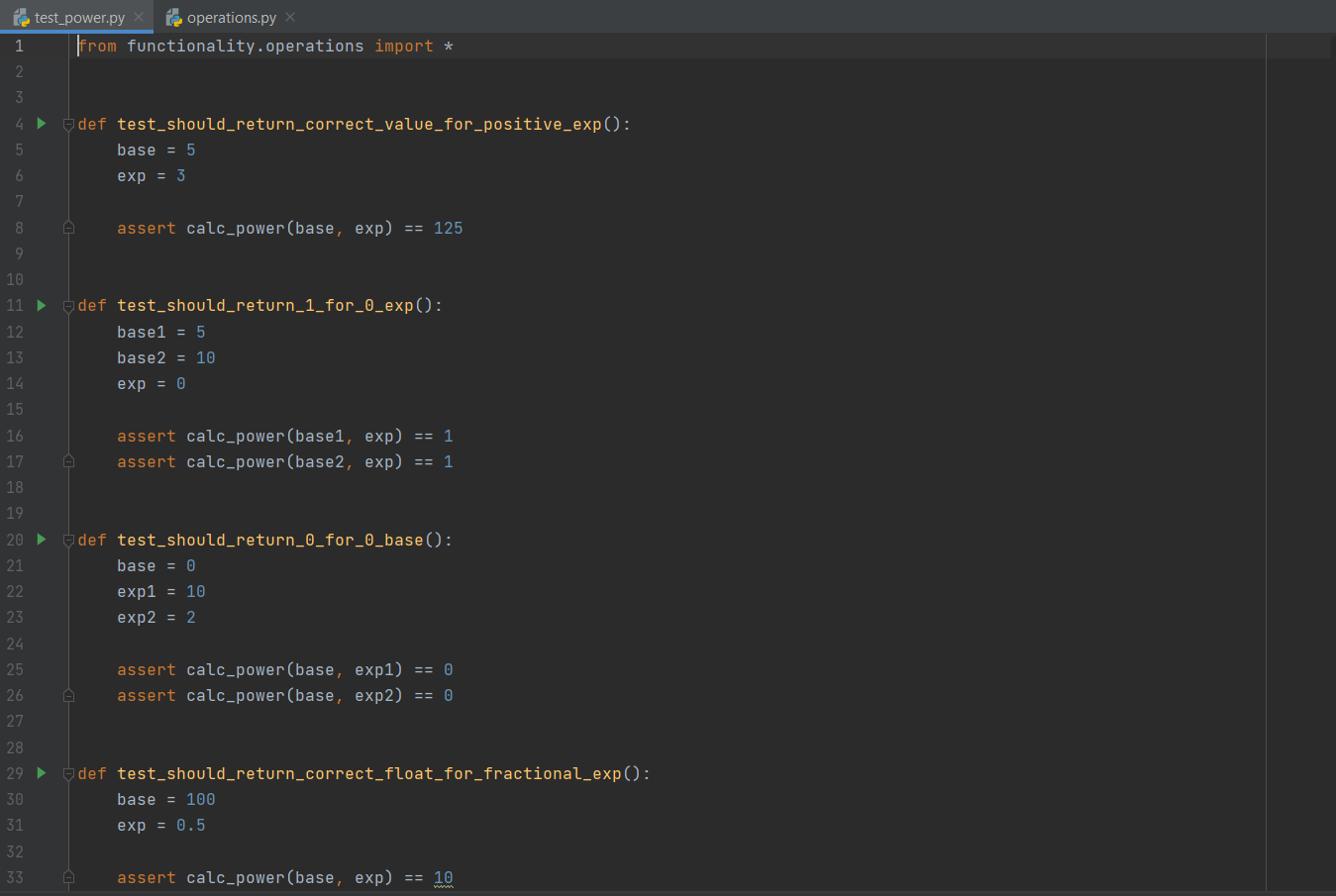
Po fazie niebieskiej możesz z czystym sumieniem rozpocząć kolejny cykl TDD - wrócić do fazy czerwonej i przystąpić do wprowadzania kolejnych testów (wykładnik ujemny, w postaci ułamka lub jako wartość 0).

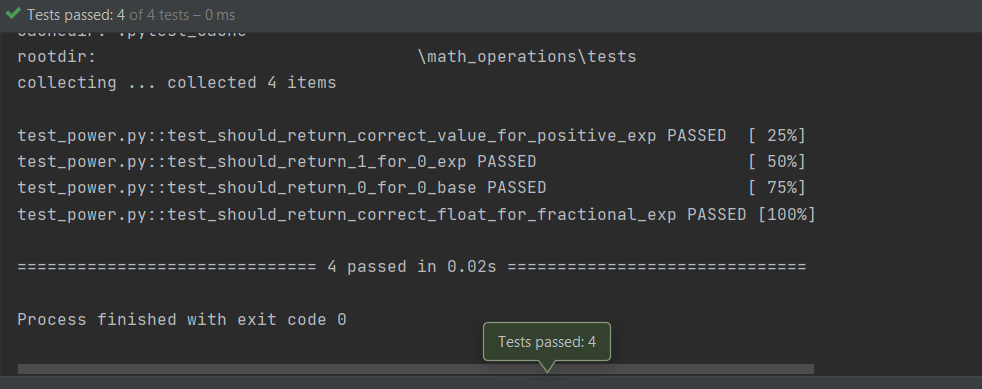
**Dalsze kroki...**

W normalnej sytuacji, kolejne testy zmuszałyby Nas do rozszerzania działania testowanej funkcji, ale z powodu jej trywialności i całego przykładu, wiemy, że w zasadzie stworzyliśmy już w pełni działającą funkcjonalność i nie ma potrzeby jej rozszerzania.

Jednak dla zasady dopiszemy pozostałe testy, które będą sprawdzały, czy funkcja zwraca poprawne wyniki dla **specyficznych wykładników potęg**.

Po przeprowadzeniu kilku cykli TDD, otrzymamy testy tak jak poniżej:

**Implementacja pozostałych testów**

**4 tests PASSED**

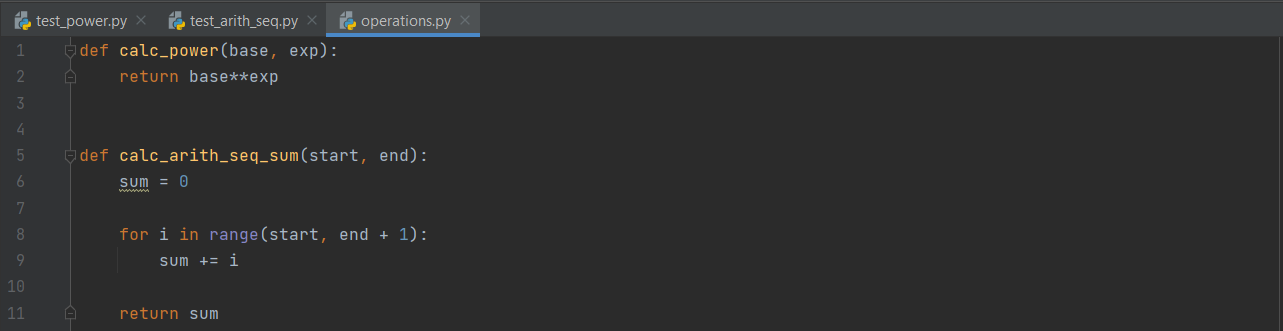
**Ćwiczenie**

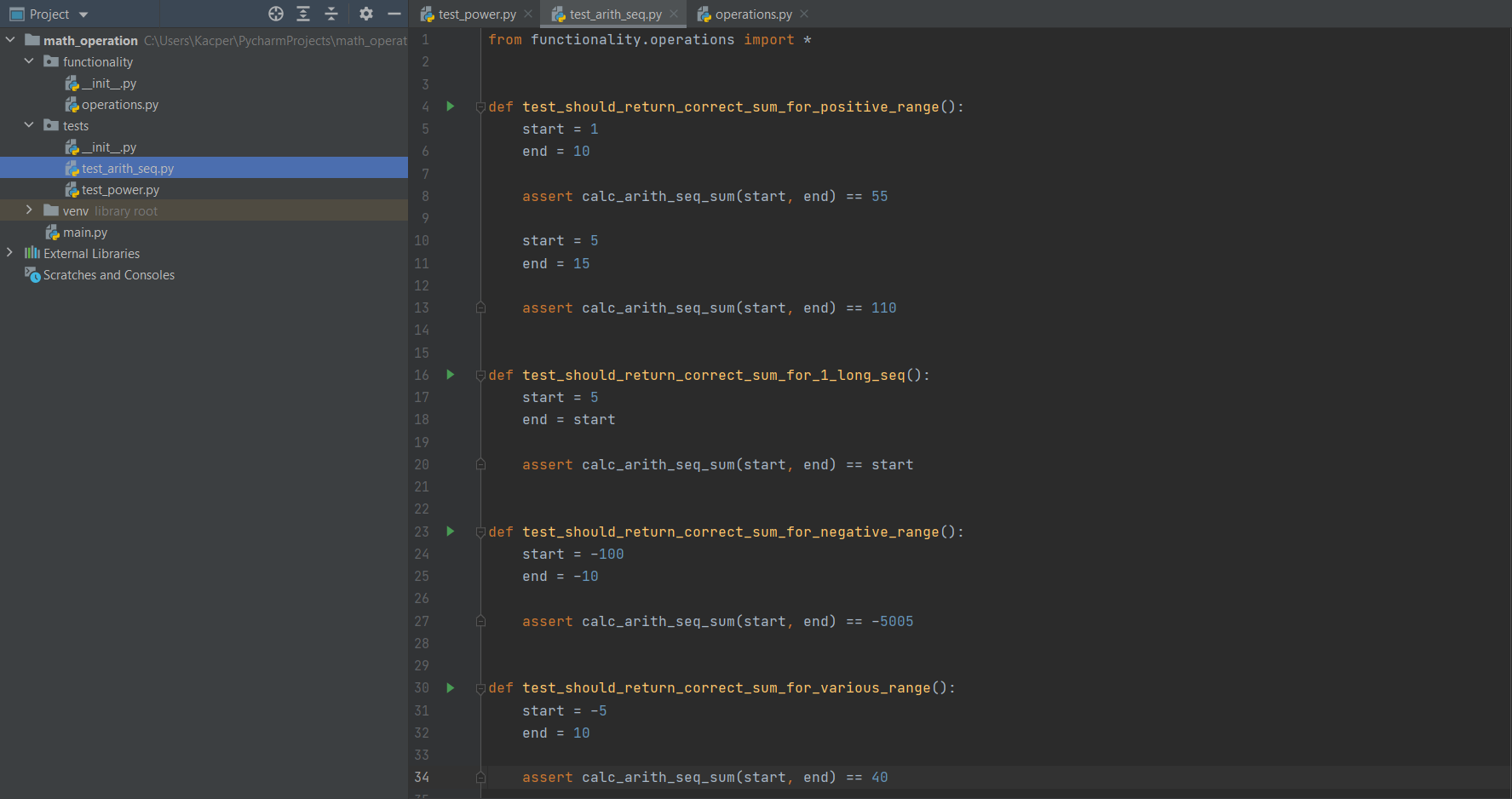
Jakiego przypadku nie pokryliśmy, pisząc powyższe testy? Dopisz pozostały przypadek.

**Podpowiedź**

A co z ujemnymi wykładnikami? : )

Na koniec jeszcze pokażę Ci, jak powinna wyglądać finalna wersja testów i druga zaimplementowana funkcjonalność (pamiętasz, że klient oczekuje jeszcze od Nas program obliczający sumę ciągu arytmetycznego? :)

**Implementacja funkcji calc\_arith\_seq\_sum**

**Testy z pliku test\_arith\_seq.py**

**Podsumowanie**

Na koniec chciałbym podsumować to, czego udało dokonać się w powyższych sekcjach.

Pokazałem Ci ogólne zasady i sposób tworzenia **efektywnego oprogramowania** (testy + kod produkcyjny) i mimo że przykład był dość trywialny, to daje on gruntowną wizję jak tworzyć bardziej rozbudowane projekty (= więcej kodu produkcyjnego, więcej testów).

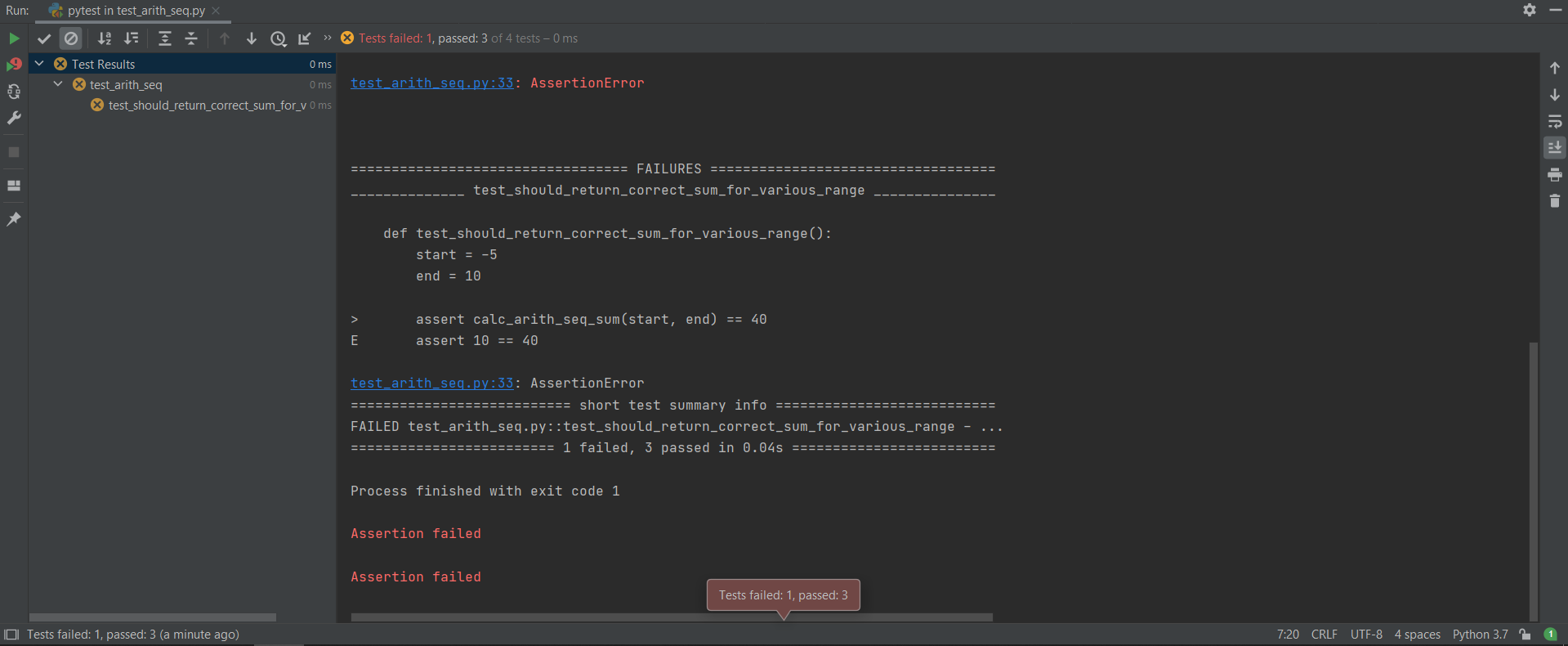
**Zasady, o których rzadko się mówi, mimo że powinieneś ich sztywno przestrzegać:**

1. Dbaj o prawidłowe rozmieszczenie plików w projekcie - zauważ, że pliki testowe powinny być **wyraźnie oddzielone od kodu produkcyjnego**.

Zwracam na to szczególną uwagę, ponieważ wiele pseudokursów z testowania w Pythonie, ukazuje jedynie podstawy pisania testów, nie przekładając w ogóle teorii na praktykę. Co więcej, zauważyłem, że standardem w takich szkoleniach jest choćby umieszczanie testów w tym samym pliku, co kod produkcyjny. Bezsens! Ktoś powie, że takie podejście ma swoje uargumentowanie w tym, że to są jedynie przykłady mające uzmysłowić użytkownikowi sposób budowania tesów... no dobrze, ale czy nie lepiej od razu zacząć od najlepszych wzorców i nie ugruntowywać złych nawyków? :)

2. W formie podsumowania chciałbym jeszcze pokazać Ci, jaki będzie output testów, jeżeli któryś z nich zakończy się niepowodzeniem. Lekko zmodyfikowałem funkcję obliczającą sumę wyrazów ciągu arytmetycznego, aby celowo zwracała niepoprawne wyniki dla przypadku, gdy początek **ciągu jest liczbą ujemną**, a **koniec liczbą dodatnią** i ponownie uruchomiłem testy.

Oto, co otrzymałem:

**test\_should\_return\_correct\_sum\_for\_various\_range FAILED**

Z otrzymanego outputu, na pewno możemy zauważyć, która asercja kodu spowodowała failure testu

(> assert calc\_arith\_seq\_sum(start,end) == 40).

Co więcej, w przypadku, gdybyśmy mieli do czynienia z większą ilością niezaliczonych asercji, niezwykle przydatna może okazać się sekcja

=========== **short test summary info** ============

Pokazuje ona bowiem, że **3 testy zakończyły się powodzeniem**, a 1 z nich otrzymał status FAILED.

Ponadto, dzięki poprawnemu i wyczerpującemu nazywaniu testów, np. **should\_return\_correct\_sum\_for\_vaious\_range** - jesteśmy w stanie niemal natychmiastowo wywnioskować, dla którego przypadku zaimplementowana funkcja zachowuje się nieprawidłowo.

Nie musimy w otrzymanych logach poszukiwać dokładnej linii, w której nastąpił błąd asercji, tylko od razu wiemy, w jakim przypadku kod jest wadliwy.

Z tego miejsca chcę Ci udowodnić na przykładzie, jak ważne jest trafne nazywania testów i czasami warto poświęcić chwilę na zastanowienie się, czy nazwa któregoś testu nie wymaga lekkiej modyfikacji.

**Markery i grupowanie testów**

Wydawać by się mogło, że temat grupowania testów wyczerpaliśmy już w poprzedniej sekcji, jednak chciałbym tutaj przedstawić Ci **dekorator** umożliwiający dogłębniejsze rozróżnianie rodzajów testów (nie tylko ich podział na pliki w zależności od funkcjonalności kodu, jaki testują).

Skupimy się bowiem na **markerach**.

Markery występują w formie **dekoratorów** dla danych funkcji.

Służą one do zaznaczenia, czy dany test służy, np. sprawdzeniu, czy funkcja zwraca oczekiwaną watość stringa (wówczas marker nazwiemy coś w deseń: @pytest.mark.check\_text) lub inny przypadek - sprawdzenie, czy zwrócona liczba przez daną funkcję znajduje się w określonym przedziale (np. @pytest.mark.check\_valid\_range) itd.

W praktyce markery nie tyle modyfikują funkcjonalność testów, co ułatwiają Nam jeszcze łatwiejsze rozeznanie w kodzie. Dzięki nim możliwe bowiem staje się grupowanie testów i w momencie, gdybyśmy chcieli sprawdzić ich zachowanie, gdy, np. przesyłają one tekst do funkcji testowanej, wystarczy uruchomić pojedyncze polecenie, a nie wszystkie testy z osobna.

Tworząc markery, należy nazywać je zgodnie z poniższym wzorcem:

@pytest.mark.<nazwa\_markera>

**UWAGA:**

Pamiętaj, aby w pliku, w którym umieszczasz marker zaimportować moduł pytest poleceniem:

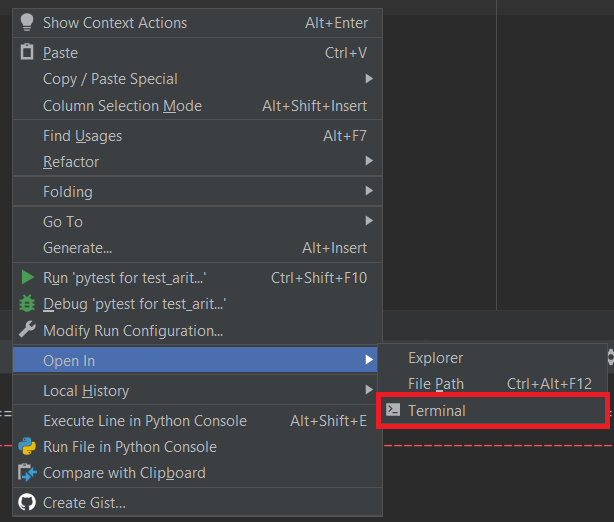
import pytest

Następnie, chcąc uruchomić grupę oznaczoną określonym markerem, musisz z poziomu terminala wywołać:

pytest -m <markername> <filename> -v

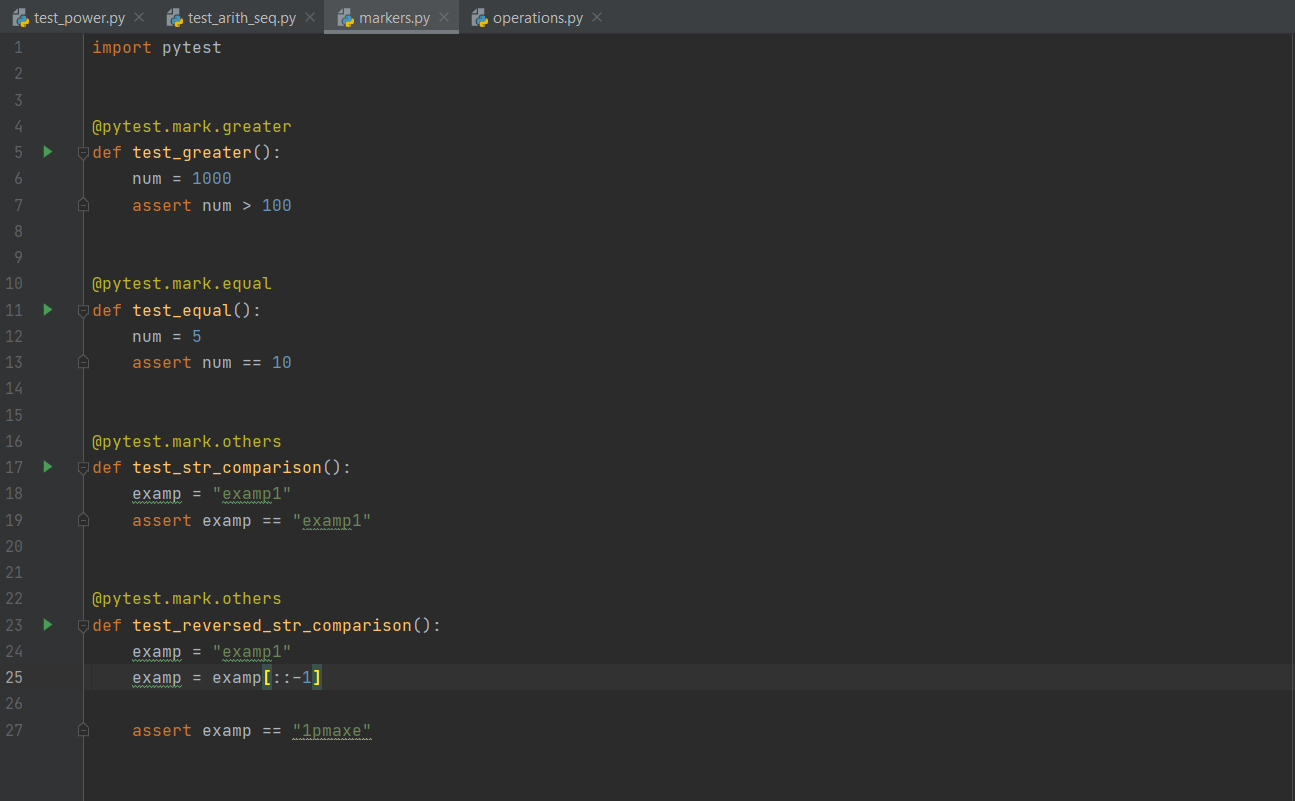
**UWAGA:**

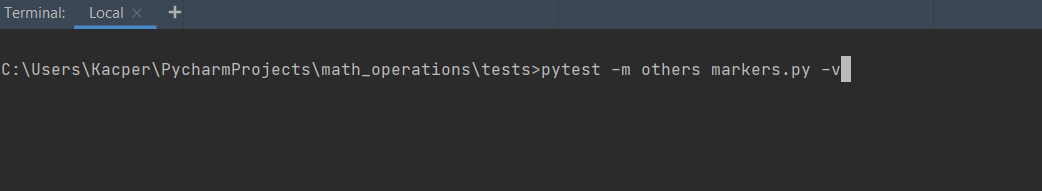
Jeżeli korzystasz ze zintegrowanego środowiska programistycznego (np. PyCharm), to nie jest możliwe uruchomienie grupy testów z danego markera przez określony przycisk. Musisz uruchomić ją ręcznie z poziomu **terminala wbudowanego** w IDE.



**PPM na edytor kodu -> Open In -> Terminal w PyCharm**

Przykład zastosowania markerów w praktyce:

**Pogrupowane testy według markerów; Plik markers.py**

**Uruchomienie grupy markerów others z pliku markers.py**

**Fikstury**

Fikstury to kolejny twór ułatwiający Nam, programistom i testerom, życie. Fikstury służą niejako do automatyzacji operacji - umożliwiają powtarzanie określonych operacji przed uruchomieniem każdego testu (dzięki temu nie tworzymy z kodu tzw. **boilerplate'a** - czyli ciągu powtarzających się sekwencji).

Testy mogą w ten sposób **automatycznie generować sobie dane** i pracować bez **żadnych wewnętrznych powiązań**.

Z procesem wykorzystywania fikstur często możemy wiązać, tzw. **"dependency injection",** czyli techniką, w której dany obiekt **bazuje na innych obiektach** (dependencjach).

Fikstura oznaczona jest w kodzie jako dekorator:

@pytest.fixutre

i umieszczana jest przed określoną funkcją. Dzięki temu tak naznaczony kod będzie automatycznie wywoływany przed wykonaniem określonych testów.

**Fikstura w praktyce**

Praktycznym przykładem, w którym mogłyby zostać wykorzystane fikstury, jest pisanie niezależnych testów dla operacji **przeprowadzanych przez bazę danych**.

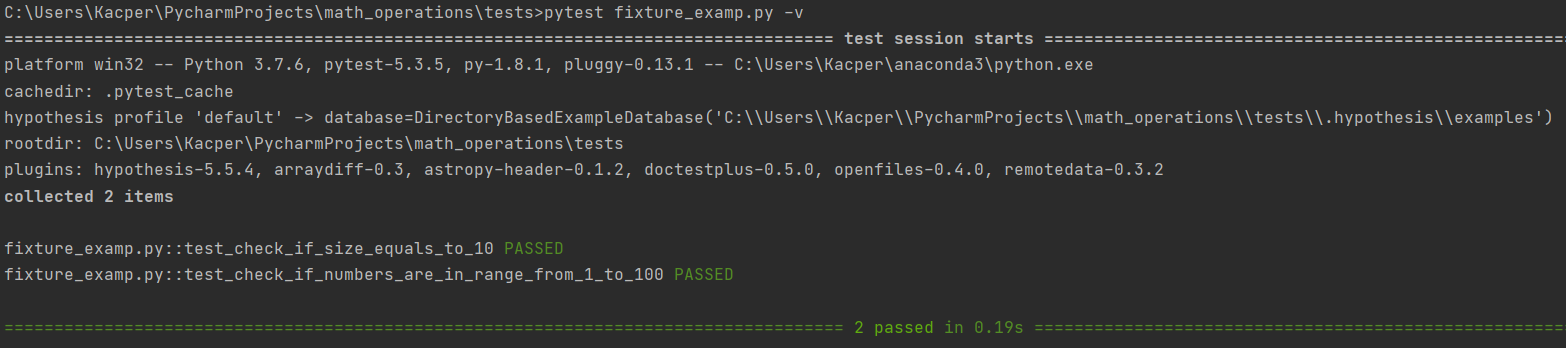
Otóż przed realizacją każdego z testów, chcielibyśmy, np. **odświeżać dane we wszystkich tabelach bazy** (tak, aby przy każdym nowo uruchamianym teście operować na różnych zasobach).

W tym celu właśnie moglibyśmy stworzyć **fiksturę w postaci metody**, która realizowałaby **odświeżanie tych danych**. Byłaby ona automatycznie wywoływana tuż przed każdym testem, co zapewniłoby wcześniej wspomnianą **niezależność i niezawodność przetwarzania danych**.

**Implementacja:**

Implementacja pierwszej fikstury w Naszym projekcie może nie będzie aż tak mocno praktyczna, jak przytoczony wcześniej przykład, ale przede wszystkim przedstawi Ci sposób na ich tworzenie i wykorzystanie w kodzie.

 **Przykład fikstury get\_list\_of\_rand\_nums()**

 **Uruchomienie testów z fiksturą**

**Wyjaśnienie:**

1. Zwróciłem już na to uwagę wcześniej, ale zauważ, jakiego dekoratora używasz w celu określenia, że dana metoda jest fiksturą **(linia 5)**

2. Zwróć również uwagę na to, jak określamy, do której fikstury **ma odnosić się dany test**. Definiujemy to na **liście argumentów** danego testu, podając nazwę **fikstury (linia 10. i 14.)**. Następnie, co jest bardzo istotne, jeżeli chcesz się odwołać do rezultatu zwracanego przez fiksturę, po prostu podajesz jej nazwę **(linia 11, 16)**. Nie traktujesz jej jakoby była funkcją, ale bezpośrednio obiektem przez nią zwracanym **(a więc nie piszemy get\_list\_of\_rand\_nums(), tylko get\_list\_of\_rand\_nums).**

**Testy sparametryzowane**

Parametryzowanie testów jest niezwykle przydatne, gdy chcemy stworzyć kilka testów **sprawdzających to samo**, ale dla **różnych danych wejściowych**.

Załóżmy, że mamy test sprawdzający, czy na **trzech otrzymanych listach** znajdują się **tylko i wyłącznie liczby nieparzyste**.

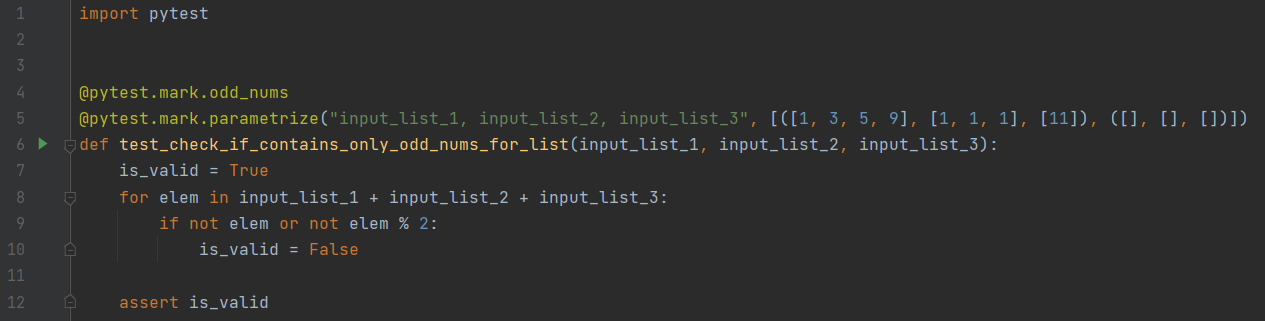
Standardowe podejście wymagałoby utworzenia kilku list o różnej zawartości i stworzenia **assert dla każdej z nich**.

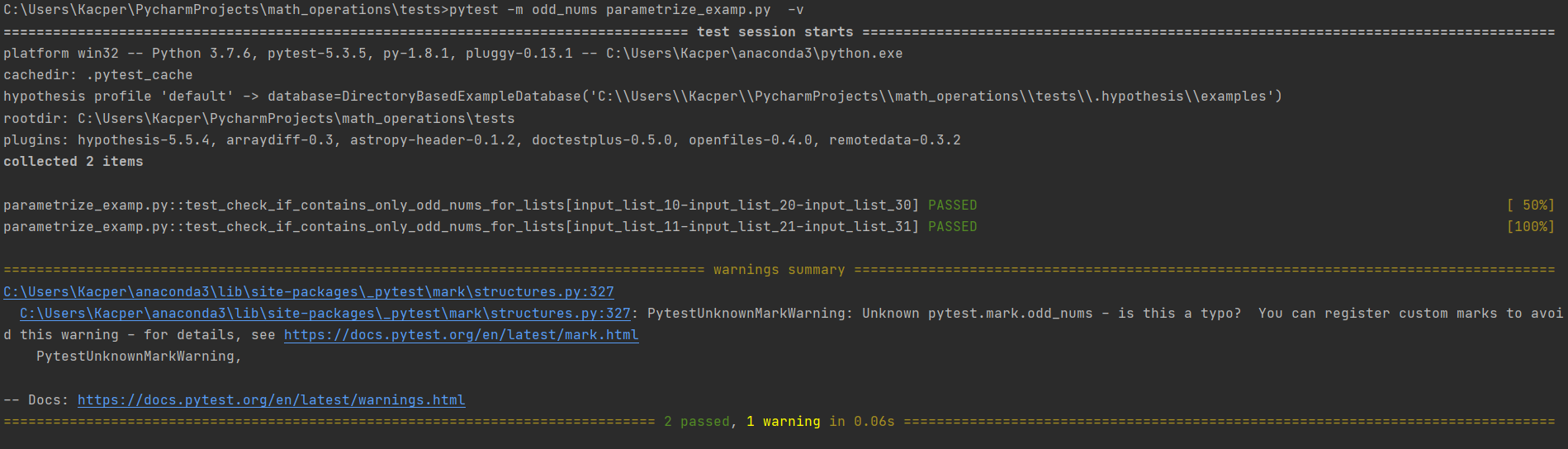
Dlatego znacznie skuteczniejszym i efektywniejszym sposobem byłoby stworzenie testu sparametryzowanego, w którym **nie tworzylibyśmy list**, tylko ich zawartość **określali przez parametr**.

**Syntax:**

@pytest.mark.parametrize(params, values)

**Przykład:**

**Test sparametryzowany**

**Wynik testów (powiększ w razie konieczności)**

W powyższym przykładzie, **w linii 4.** użyłem dekoratora, który dodał tworzony test do grupy **odd\_nums**. To tak na odświeżenie wiadomości z poprzednich sekcji, ale nie w tym rzecz.

Musimy pojąć, jakie to cuda zadziały się w **linii 5.** i dlaczego ma ona taką, a nie inną strukturę.

Otóż jak widzisz w poniższym zapisie:

@pytest.mark.parametrize("input\_list\_1, input\_list\_2, input\_list\_3", [([1, 3, 5, 9], [1, 1, 1], [11]), ([], [], [])])

dekorator @pytest.mark.parametrize przyjmuje **2 argumenty**.

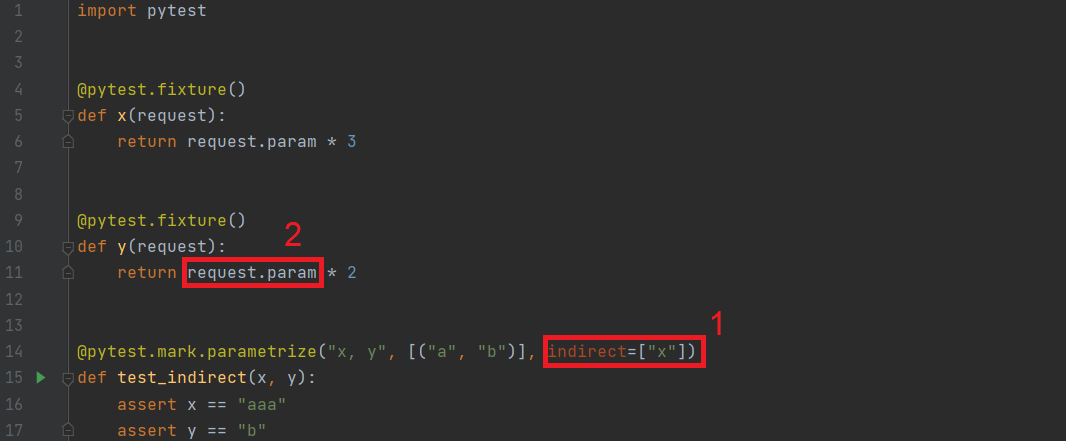
Pierwszy (params) jest stringiem i **określa nazwy argumentów**, które **parametryzujemy**, a drugi w postaci **listy**, w którym **umieszczamy tuple**, a w nich **wartości parametrów dla zdefiniowanych parametrów**.

I teraz uważaj! W zadaniu rozpatrujemy 2 przypadki, jak zachowa się test, gdy prześlemy do niego 3 różne listy (parametry).

Każdy z przypadków określamy **w oddzielnej tupli** i tak np.

* **([1, 3, 5, 9], [1, 1, 1], [1])** oznacza tuplę (przypadek) nr I, który przyjmuje **3 różne listy** jako wartości argumentów **input\_list\_1**, **input\_list\_2**, **input\_list\_3**
* **([], [], [])** oznacza II przypadek wywołania testu, który tym razem przyjmuje 3 puste listy jako swoje argumenty

**Zaawansowany przykład wykorzystujący testy parametryzowane wraz fiksturami**

****

Prezentując powyższy przykład miałem przede wszystkim na celu pokazanie Ci, jak praktyczne może być łączenie poznanych zagadnień w integralną całość!

Otóż stworzyliśmy tak naprawdę sparametryzowaną fiksturę. Konieczne jednak było podanie tych **samych nazw parametrów** dla dekoratora @pytest.mark.parametrize(), co posiada **utworzona fikstura** (w naszym przypadku x).

Zwróć również uwagę na zaznaczony na czerwono obszar w kodzie. Aby zapewnić **wywołanie fikstury z odpowiednimi parametrami** (1) oraz ich odebranie z poziomu już wewnątrz samej metody (2), niezbędne jest dodanie widocznych udogodnień w kodzie (keyword argument w postaci **indirect** oraz **name.param** jako odwołanie się do wartości przekazanego argumentu).

Z takimi smaczkami w kodzie możesz spotykać się często i nie sposób opisać każdy z nich w jednym materiale szkoleniowym. Dlatego bazuj przede wszystkim na oficjalnej dokumentacji frameworka, a omawiany tutaj temat testowania wskaże Ci najważniejsze informacje i odpowiednią drogę rozwoju!

**Pokrycie testami (Code Coverage)**

Tzw. **Code Coverage** jest niczym innym jak miarą, która określa, jak **efektywne są** tworzone testy jednostkowe i czy każda możliwa funkcjonalność kodu **produkcyjnego została sprawdzona** (pokryta, ang. covered).

**Jakich informacji dostarcza nam wskaźnik Code Coverage?**

Przede wszystkim pomaga on w znalezieniu funkcjonalności, którą **nie sprawdzają testy jednostkowe**. A więc zabezpieczają Nas przed potencjalnymi “dziurami” w kodzie produkcyjnym i jego nieprzewidywalnym zachowaniu. Umożliwiają również łatwiejszą i bardziej przemyślaną implementację kolejnych testów.

**Poziomy Code Coverage**

Z racji tego, że w strukturze dowolnego programu możemy wyróżnić różne twory, np. instrukcje warunkowe, funkcje, pętle, to musimy różnić **różne poziomy**, do których może odnosić się **miara pokrycia kodu**.

Poziomy:

* **Function Coverage** - określa odsetek zdefiniowanych funkcji w programie, które zostały wywołane i sprawdzone przez testy jednostkowe
* **Statement Coverage** - określa odsetek instrukcji warunkowych (ang. statements), które zostały przetestowane
* **Lines Coverage** - określa odsetek ilości linii kodu źródłowego, które zostały przetestowane

**Jak ręcznie obliczyć pokrycie kodu testami?**

Aby lepiej zrozumieć, jak pokrycie może być wyliczane (na poziomie **Line Coverage**), przeanalizuj poniższe kroki.

Jeżeli dany komponent oprogramowania ma **w sumie 500 linijek kodu** i testy sprawdzają jedynie **50 linijek tego komponentu** (np. testują tylko, czy pętla umieszczona w danym featurze zwraca oczekiwany wynik), to wówczas pokrycie kodu będzie równe:

(50 / 500) \* 100 = **10%**

Jak już zapewne się domyślasz, im **większy współczynnik pokrycia kodu testami, tym lepiej**. Wtedy mamy pewność, że napisane testy w pełni sprawdzają funkcjonalność kodu produkcyjnego.

**Test Coverage w praktyce - narzędzie PyTest-Cov**

Wiedząc już mniej więcej, czemu służy dbanie o wysokie pokrycie kodu produkcyjnego testami, możemy przejść do praktycznego wykorzystania narzędzia **pytest-cov.**

**Instalacja**

W celu zainstalowania pytest-cov wykorzystaj polecenie:

pip install pytest-cov

lub zainstaluj ręcznie moduł w venv.

**Uruchamianie procesu sprawdzającego Code Coverage**

Aby uruchomić zainstalowane narzędzie pytest-cov, wykorzystamy polecenie

pytest --cov=katalog

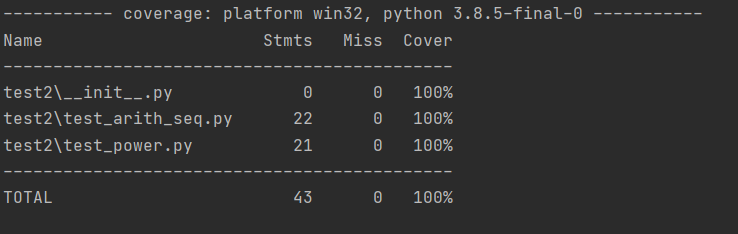
, które uruchomimy, znajdując się w katalogu projektowym (pamiętaj, aby znajdować się w katalogu bezpośrednio nadrzędnym do katalogu testowanego).

Czyli wykonamy, np.

pytest --cov=tests/

gdzie **tests/** to katalog z testami.

Twoim oczom ukaże się coś na wzór:



**Wynik polecenia pytest --cov**

**Objaśnienia:**

**Stmts (statements)** - to ilość pokrytych wyrażeń kodu produkcyjnego przez uruchamiane testy. **Wyrażenia** to wszelkiego rodzaju operacje takie jak: imports, klasy, funkcje, definicje.

**Miss** - odnosi się do wyrażeń niepokrytych testami.

**Cover** - ostateczna miara (współczynnik) pokrycia kodu testami.

**Rozszerzenie przykładu**

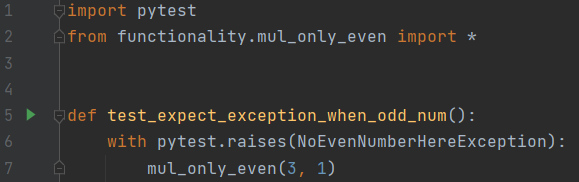
Temat może nie jest w pełni związany z Code Coverage, ale myślę, że jest to dobra pora na wtrącenie przykładu testu sprawdzającego, czy czy dana metoda (funkcja) **rzuca wyjątkiem** w określonych przypadkach.

Do tej pory bowiem sprawdzaliśmy warunki przy użyciu assert. A co w momencie, gdy od danej funkcji **będziemy oczekiwali rzucenia wyjątku** jako wynik testu?

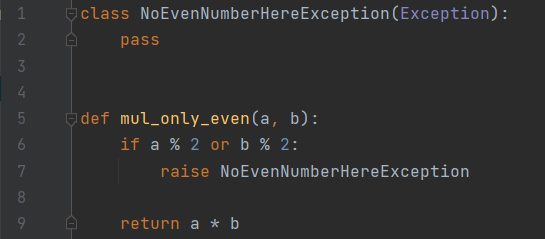
Załóżmy, że chcemy przetestować funkcję sprawdzającą wynik mnożenia tylko i wyłącznie liczb parzystych.

Logiczne jest, że dobrym pomysłem byłoby **zabezpieczenie funkcji** przed odebraniem **wartości nieparzystych**. Zabezpieczenie to moglibyśmy realizować dodając do testowanej metody **możliwość rzucenia wyjątkiem**, gdy **jedna z przesłanych** liczb nie będzie parzysta.

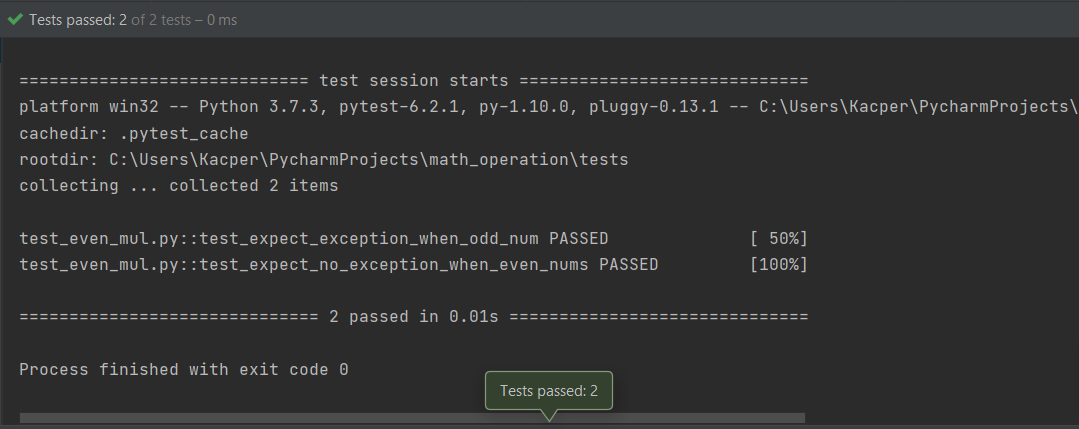
Spójrz, jak to można zrealizować w kodzie:



**Test oczekujący od metody rzucenia wyjątku**



**Funkcja mogąca rzucić wyjątek**



**Wynik testowania**

Linia 6 testu test\_except\_exception\_when\_odd\_num realizuje sprawdzenie, czy funkcja faktycznie rzuca **własnym wyjątkiem NoEvenNumberHereException** (jeżeli nie pamiętasz, czym są wyjątki w Pythonie, zobacz szkolenie Wyjątki. Obsługa wyjątków).

Realizowane jest to dzięki poleceniu:

with pytest.raises(NoEvenNumberExceptionHere):

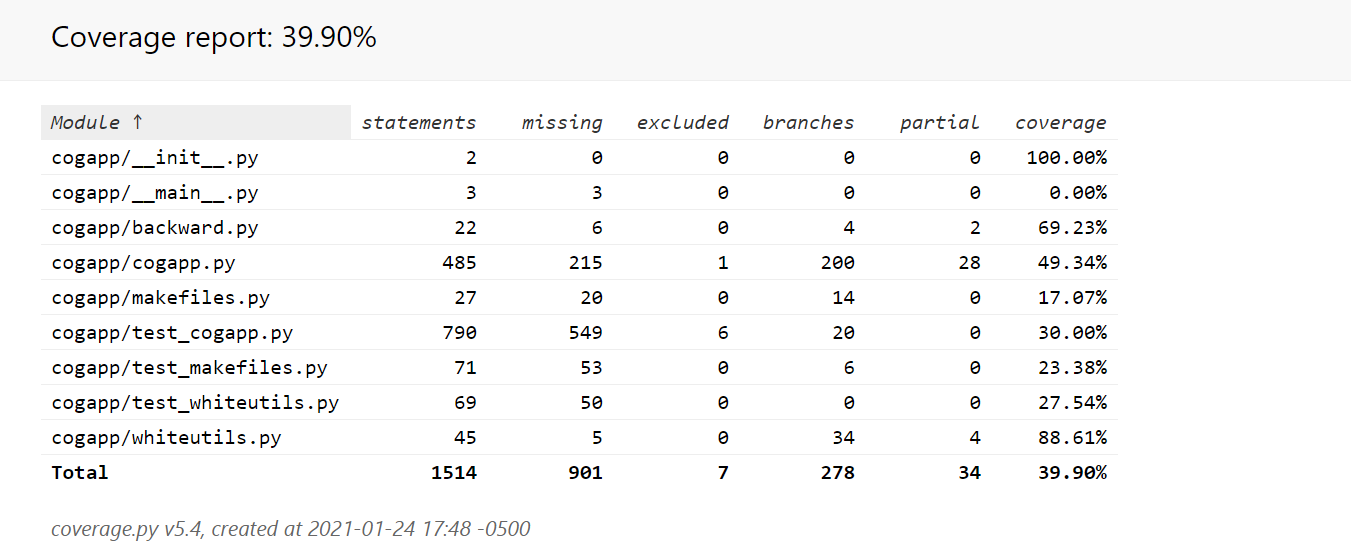
...

**Zapis raportu do pliku**

Wracając już do pokrycia testowego… Chciałbym wtrącić w to miejsce krótką dygresję.

Jeżeli chciałbyś **zapisać raport pokrycia testowego**, po wykonaniu polecenia **pytest --cov=katalog**, możesz wpisać w terminal polecenie: coverage html.

W wyniku tego wygenerujesz plik html, który będziesz mógł otworzyć z poziomu katalogu **htmlcov/index.html**.

**Przykładowy raport pokrycia testowego**

Na koniec, pytanie, na które warto postawić jednoznaczną odpowiedzieć **- Jak bardzo rzetelna jest miara pokrycia testami**?

Otóż współczynnik Code Coverage jest przydatną miarą do wyznaczania, jak efektywne są stworzone przez Nas testy, ale nie powinniśmy **bazować na nim** w 100%.

Wysoka wartość tego miernika wcale **nie gwarantuje**, że **kod jest wolny od błędów i bugów**. To, co niesie ze sobą współczynnik pokrycia kodu, to tak naprawdę informacja, czy **dana linia (wyrażenie) kodu** została sprawdzona przez test **chociaż raz.**

Miara pokrycia kodu testami wcale nie dostarcza Nam w pełni rzetelnej informacji, o tym że testy sprawdziły każdy możliwy przypadek funkcjonalności kodu produkcyjnego.

Miej na to szczególną uwagę i polegaj na własnych umiejętnościach, a wynik zwracany przez pytest-cov traktuj **jako jedynie kierunkowskaz** do tworzenia efektywnego kodu.