Testy i Typowanie

Gdzie jesteśmy?

- Potrafimy budować złożone, wielomodułowe aplikacje (OOP, wzorce).
- Nasz kod potrafi komunikować się ze światem zewnętrznym (pliki, API).

Nowe, ostateczne wyzwanie: Zaufanie i Utrzymanie

- Skąd mamy pewność, że nasz skomplikowany kod działa poprawnie?
- Jak wprowadzać zmiany, nie psując czegoś w innym miejscu (regresja)?
- Jak łapać proste błędy (np. **TypeError**) zanim uruchomimy program?

Cel na dziś: Opanować dwa filary profesjonalnego programowania, które budują zaufanie do kodu i ułatwiają jego rozwój.

Część 1: Wprowadzenie do pytest

- Zadanie 1: Mój pierwszy test - **assert** i konwencje nazewnicze.
- Zadanie 2: Testowanie wyjątków (**pytest.raises**).
- Zadanie 3: Reużywalny setup **(@pytest.fixture**).

Część 2: Zaawansowane techniki pytest

- Zadanie 4: Testowanie wielu przypadków (**@pytest.mark.parametrize**).
- Zadanie 5: Izolowanie testów od zależności zewnętrznych (**pytest-mock**).

Część 3: Wprowadzenie do Statycznego Typowania

- Zadanie 6: Dodawanie adnotacji typów (**type hints**).
- Zadanie 7: Statyczna analiza z **mypy**.
- Zadanie 8: Zaawansowane Typowanie (**Generics**, **Union**).
- Zadanie 9: Mierzenie Pokrycia Kodu (**pytest-cov**).

Podsumowanie: Dwuwarstwowa Siatka Bezpieczeństwa.

Zadanie 1 – Pierwszy test (assert i konwencje)

Problem: Jak w sposób automatyczny i powtarzalny sprawdzić, czy nasza prosta funkcja **dodaj(a, b)** działa poprawnie?

Rozwiązanie: pytest - nowoczesny framework do testowania.

Kluczowe Konwencje pytest:

a. **Nazwa pliku:** Musi zaczynać się od **test_** lub kończyć na **_test.py.**
   - Przykład: **test_kalkulator.py**

b. **Nazwa funkcji**: Musi zaczynać się od **test_.**
   - Przykład: **def test_dodawania_liczb_dodatnich():**

c. **Asercja:** Używamy standardowego słowa kluczowego **assert**, aby sprawdzić, czy warunek jest prawdziwy. Jeśli nie, test kończy się porażką.

Zadanie:

a. **Instalacja:** W terminalu wpisz **pip install pytest**.

b. **Stwórz plik kalkulator.py** z prostą funkcją **dodaj**.

c. **Stwórz plik test_kalkulator.py** w tym samym folderze.

d. W pliku testowym, zaimportuj funkcję **dodaj.**

e. Napisz funkcję testową **test_dodawania_liczb_dodatnich**, która użyje **assert**, aby sprawdzić, czy **dodaj(2, 3)** zwraca **5**.

f. Napisz drugą funkcję **test_dodawania_liczb_ujemnych.**

g. **Uruchom testy:** W terminalu, będąc w folderze z plikami, wpisz komendę **pytest.**

Omówienie Zadania 1 – Pierwszy test (assert i konwencje)

Cel: Analiza wyników i zrozumienie, jak pytest automatycznie odkrywa i uruchamia testy.

Wyjaśnienie:

- **pytest** przeskanował folder i znalazł plik **test_kalkulator.py**.
- Wewnątrz pliku znalazł dwie funkcje pasujące do wzorca **test_*.**
- Uruchomił każdą z nich. Asercje (**assert**) w obu testach były prawdziwe, więc oba testy **przeszły** (ang. passed).
- Dwie kropki **..** oznaczają dwa pomyślnie zakończone testy.

Problem: Potrafimy testować poprawne wyniki. A jak sprawdzić, czy nasz kod poprawnie rzuca błędy w sytuacjach wyjątkowych?

Zadanie 2 - Testowanie Wyjątków (pytest.raises)

Problem: Jak sprawdzić, czy nasz kod **poprawnie rzuca błędy** w sytuacjach wyjątkowych (np. dzielenie przez zero)?

Złe rozwiązanie: Używanie **try...except** w teście. Jest to "gadatliwe" i niejasno komunikuje intencję.

Rozwiązanie: Menedżer kontekstu pytest.raises

- „Mówi” **pytest**: "Oczekuję, że kod w tym bloku rzuci konkretny wyjątek".
- Test **przechodzi**, jeśli oczekiwany wyjątek zostanie rzucony.
- Test **kończy się porażką**, jeśli nie zostanie rzucony żaden wyjątek lub zostanie rzucony inny.

Zadanie:

a. **Rozbuduj kalkulator.py** o funkcję **dziel(a, b)**, która rzuca **ValueError**, jeśli próbujemy dzielić przez zero.

b. W pliku **test_kalkulator.py**, napisz nową funkcję testową **test_dzielenia_przez_zero_powinno_rzucic_blad**.

c. Wewnątrz testu, użyj menedżera kontekstu **with pytest.raises(...)**, aby sprawdzić, czy wywołanie **dziel(10, 0)** faktycznie rzuca **ValueError**.

d. (Opcjonalnie) Dodaj test **test_poprawnego_dzielenia**, aby sprawdzić normalne działanie funkcji.

e. Uruchom **pytest** i przeanalizuj wyniki.

Omówienie Zadania 2 - Testowanie Wyjątków

Cel: Analiza wyników i zrozumienie, jak pytest.raises weryfikuje poprawne rzucanie błędów.

Wyjaśnienie:

- **pytest** znalazł i uruchomił wszystkie 4 testy (2 dla **dodaj**, 2 dla **dziel**).
- Test **test_dzielenia_przez_zero_powinno_rzucic_blad przeszedł**, ponieważ kod wewnątrz bloku **with** rzucił oczekiwany **ValueError.**
- Gdyby wyjątek nie został rzucony, test zakończyłby się porażką.

Problem: Wiele testów potrzebuje tych samych danych startowych (np. obiektu klasy, połączenia z bazą). Kopiowanie kodu przygotowawczego (setup) jest złe (łamie zasadę DRY). Jak to rozwiązać?

Zadanie 3 - Reużywalny kod przygotowawczy z Fixtures (@pytest.fixture)

Problem: Wiele testów potrzebuje tych samych danych startowych (np. obiektu klasy). Kopiowanie kodu przygotowawczego (setup) jest złe (łamie zasadę DRY).

Rozwiązanie: Fixtures (urządzenia testowe) - reużywalne funkcje, które przygotowują "środowisko" dla testu i dostarczają dane.

Jak działają Fixtures?

a. Tworzymy funkcję i dekorujemy ją **@pytest.fixture**.

b. Funkcja ta tworzy i **zwraca** (**return**) zasób (np. obiekt).

c. Funkcja testowa "prosi" o fixture, podając jej nazwę jako argument. **pytest** sam ją wywoła i przekaże wynik.

Zadanie:

a. Stwórz plik portfel.py z klasą **Portfel** i wyjątkiem **NiewystarczajaceSrodki.**

b. Stwórz plik test_portfel.py.

c. W pliku testowym, stwórz fixture o nazwie **pusty_portfel**, która tworzy i zwraca pustą instancję Portfela.

d. Napisz test **test_poczatkowego_salda,** który przyjmuje **pusty_portfel** jako argument i sprawdza, czy jego saldo początkowe wynosi 0.

e. Napisz drugi test **test_wplaty_do_portfela**, który również przyjmuje **pusty_portfel**, dokonuje wpłaty i sprawdza, czy saldo się zmieniło.

f. Uruchom **pytest** i zaobserwuj, jak oba testy korzystają z tej samej logiki przygotowawczej.

Omówienie Zadania 3 - Reużywalny kod przygotowawczy z Fixtures

Cel: Analiza poprawnego rozwiązania i utrwalenie kluczowych koncepcji:

@pytest.fixture, wstrzykiwanie zależności, izolacja testów.

Wyjaśnienie:

- **pytest** widzi, że oba testy potrzebują **pusty_portfel.**
- Przed każdym testem, **pytest** uruchamia funkcję **pusty_portfel()** i przekazuje jej wynik jako argument.
- **Izolacja Testów:** Fixture jest uruchamiana **osobno dla każdego testu. test_wplaty_do_portfela dostaje nową, czystą instancję** portfela, a nie tę samą, której używał poprzedni test.

Zadanie 4 - Testowanie wielu przypadków (Parametryzacja)

Problem: Jak przetestować tę samą logikę dla wielu różnych danych wejściowych bez pisania wielu, niemal identycznych testów?

Rozwiązanie: Parametryzacja (@pytest.mark.parametrize)

- Dekorator, który uruchamia tę samą funkcję testową wielokrotnie z różnymi argumentami.
- Działa jak pętla for dla twojego testu.

Składnia:

```
@pytest.mark.parametrize("arg1, arg2, ..., oczekiwany_wynik", [
   (dane1_arg1, dane1_arg2, ..., dane1_wynik),
   (dane2_arg1, dane2_arg2, ..., dane2_wynik),
])
```

Zadanie:

a. Wróć do pliku **test_kalkulator.py**.

b. Usuń (lub zakomentuj) istniejące testy dla funkcji **dodaj**.

c. Napisz **jeden** nowy test **test_dodawania_wielu_przypadkow**.

d. Użyj dekoratora **@pytest.mark.parametrize**, aby przetestować co najmniej 4 przypadki:
   - Dwie liczby dodatnie.
   - Dwie liczby ujemne.
   - Liczbę dodatnią i ujemną.
   - Liczbę i zero.

e. Uruchom **pytest -v** (**-v** dla trybu "verbose") i zobacz, jak **pytest** raportuje wyniki dla każdego przypadku z osobna.

Omówienie Zadania 4 - Testowanie wielu przypadków (Parametryzacja)

Cel: Analiza rozwiązania i zrozumienie, jak **@pytest.mark.parametrize** upraszcza testowanie wielu przypadków i poprawia czytelność.

Wyjaśnienie:

- **pytest** znalazł 3 funkcje testowe, ale dzięki parametryzacji uruchomił łącznie 6 testów (4 dla **dodaj**, 2 dla **dziel**).
- Flaga **-v** (verbose) sprawia, że **pytest** pokazuje każdy przypadek parametryzacji jako osobny test.
- Nazwa testu, np. [**2-3-5**], jest automatycznie generowana z parametrów, co ułatwia identyfikację błędu.

Zadanie 5 - Izolowanie testów: Wprowadzenie do Mockowania

Problem: Jak testować kod, który ma **zależności zewnętrzne** (API, baza danych, pliki)? Taki test jest **wolny, niestabilny** i testuje **nie tylko nasz kod.**

Rozwiązanie: Mockowanie (udawanie) - zastępowanie prawdziwych, problematycznych obiektów "fałszywkami" (mockami), które w pełni kontrolujemy.

Narzędzie: Wtyczka **pytest-mock.**

Jak to działa? (mocker.patch)

- Test "prosi" o specjalną fixture **mocker.**
- Używamy **mocker.patch("ścieżka.do.obiektu", ...)** aby go podmienić.
- Definiujemy, co podstawiony obiekt ma robić (np. **return_value=...**).
- Uruchamiamy nasz kod, który myśli, że używa prawdziwego obiektu.
- Sprawdzamy, czy nasza logika poprawnie przetworzyła fałszywe dane.

Zadanie:

a. **Instalacja: pip install pytest-mock requests.**

b. **Stwórz plik kursy_walut.py** z funkcją **pobierz_cene_euro()**, która używa biblioteki **requests** do pobrania danych z API NBP.

c. **Stwórz plik test_kursy_walut.py.**

d. Napisz test, który **nie będzie łączył się z siecią**.

e. Użyj **mocker.patch**, aby podmienić **requests.get**.

f. Skonfiguruj "fałszywkę", aby zwracała kontrolowane przez Ciebie dane.

g. Sprawdź, czy Twoja funkcja poprawnie wyciąga kurs z fałszywych danych.

Omówienie Zadania 5 - Izolowanie testów: Wprowadzenie do Mockowania

Cel: Analiza rozwiązania i zrozumienie, jak **mocker.patch** pozwala izolować testy od zależności zewnętrznych.

Wyjaśnienie:

- Test **nie połączył się z internetem.** Był błyskawiczny i w pełni powtarzalny.
- **mocker.patch("requests.get", ...)** przechwyciło wywołanie. Zamiast prawdziwej funkcji **requests.get**, wykonany został nasz "fałszywy" obiekt.
- **return_value** określiło, co ma zwrócić ta fałszywka.
- Testowaliśmy **naszą logikę** (poprawne parsowanie słownika), a nie działanie API NBP.

Zadanie 6 - Dodawanie adnotacji typów (type hints)

Problem: Testy łapią błędy w **czasie działania** programu. Ale co z prostymi pomyłkami, które moglibyśmy wyłapać wcześniej?

- **def dodaj(a, b): return a + b**
- Co, jeśli ktoś wywoła **dodaj("2", "3")?** Wynik to **"23"**, a nie **5**. To logiczny błąd, który testy musiałyby wykryć.

Rozwiązanie: Stopniowe Typowanie (Gradual Typing) - dodawanie **adnotacji typów (type hints)** do kodu.

- To **dokumentacja** oczekiwanych typów argumentów i wartości zwracanych.
- Interpreter Pythona **całkowicie ignoruje** te adnotacje w czasie działania programu.
- Pozwalają **zewnętrznym narzędziom** (jak **mypy**) na przeprowadzenie **statycznej analizy** i znalezienie błędów **przed uruchomieniem kodu.**

Składnia:

- **zmienna: typ**
- **def funkcja(argument: typ) -> typ_zwrotu:**

Zadanie:

a. Wróć do plików **kalkulator.py** i **portfel.py**.

b. Dodaj adnotacje typów do wszystkich funkcji i metod, które stworzyłeś/aś.

c. Określ typy argumentów, atrybutów (**__init__**) oraz typy wartości zwracanych.

d. Uruchom ponownie testy (**pytest**), aby upewnić się, że adnotacje nie zmieniły działania kodu.

Omówienie Zadania 6 - Dodawanie adnotacji typów (type hints)

Cel: Analiza rozwiązania i utrwalenie, jak adnotacje typów poprawiają czytelność i przygotowują kod do statycznej analizy.

Korzyści (na razie):

- **Samo-dokumentujący się kod:** Od razu widać, jakich typów oczekuje funkcja i co zwraca.
- **Lepsze wsparcie IDE:** Edytor kodu (np. PyCharm, VS Code) używa tych informacji do lepszego autouzupełniania i wykrywania błędów "w locie".

Problem: Jak możemy **automatycznie sprawdzić**, czy nasz kod jest zgodny z tymi adnotacjami, **zanim** go uruchomimy?

Zadanie 7 - Statyczna analiza z mypy

Problem: Same adnotacje typów nie dają **żadnej gwarancji**. Interpreter Pythona je ignoruje. Jak możemy **automatycznie sprawdzić**, czy nasz kod jest zgodny z tymi adnotacjami, **zanim** go uruchomimy?

Rozwiązanie: mypy - statyczny analizator typów.

- To zewnętrzny program, który **czyta** Twój kod i adnotacje.
- **Nie uruchamia** kodu, tylko go analizuje.
- Znajduje niespójności typów i raportuje je jako błędy.

Zadanie:

a. Instalacja: pip install mypy.

b. Stwórz nowy plik bledny_kod.py.

c. Wewnątrz, napisz funkcję **powitaj(imie: str) -> str.**

d. Celowo wywołaj ją z błędnym typem, np. **powitaj(123).**

e. Uruchom analizę: W terminalu wpisz **mypy bledny_kod.py.**

f. Zaobserwuj błąd, który **mypy** wykryje, mimo że kod nie został uruchomiony.

g. Uruchom analizę na swoich poprzednich plikach (**kalkulator.py, portfel.py**). Jeśli wszystko jest dobrze otypowane, **mypy** nie powinno zgłosić żadnych błędów.

Omówienie Zadania 7 - Statyczna analiza z mypy

Cel: Analiza wyników i zrozumienie, jak mypy łapie błędy typów przed uruchomieniem kodu.

Przykładowe rozwiązanie:

```
# Plik: bledny_kod.py
def powitaj(imie: str) -> str:
    return f"Cześć, {imie}!"

# BŁĘDNE użycie - przekazujemy int zamiast str
powitaj(123)
```

Wynik uruchomienia mypy bledny_kod.py:

```
bledny_kod.py:5: error: Argument 1 to "powitaj" has incompatible
type "int"; expected "str"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

Wyjaśnienie:

- **mypy nie uruchomił kodu.** Przeanalizował go "na sucho".
- Zobaczył, że funkcja **powitaj** oczekuje **str**, a dostała **int**.
- Zgłosił precyzyjny błąd, wskazując linię i naturę problemu.
- Uruchomienie **mypy** na poprawnie otypowanych plikach (**kalkulator.py**, **portfel.py**) nie zwraca żadnych błędów, co oznacza sukces.

W ten sposób połączyliśmy dwa filary profesjonalnego warsztatu:

- **Testy (pytest):** Sprawdzają **logikę** i **zachowanie** w czasie działania. Odpowiadają na pytanie: "Czy kod robi to, co powinien?".
- **Typowanie (mypy):** Sprawdza **poprawność użycia danych** przed uruchomieniem. Odpowiada na pytanie: "Czy dane pasują do siebie?".

Zadanie 8 - Zaawansowane Typowanie (Generics i Union)

Problem: Typ list mówi tylko, że to lista. Ale lista czego? A co jeśli funkcja może zwrócić None?

- **def procesuj(dane: list): ...**
  dane[0] może być czymkolwiek.

Rozwiązanie: Typy Generyczne i Unie.

- **list[int]:** Lista zawierająca wyłącznie liczby całkowite.
- **float | None** (lub **Optional[float]**): Wartość to liczba **LUB** nic.

Zadanie:

a. W pliku **kalkulator.py** dodaj funkcję **srednia(liczby: list[float]) -> float | None**.

b. Logika: Jeśli lista jest pusta, zwróć None. W przeciwnym razie zwróć sumę podzieloną przez długość.

c. Stwórz nowy plik **analiza.py**.

d. Zaimportuj **srednia** i napisz kod:

```
wynik = srednia([])
# BŁĄD: Próba użycia wyniku bez sprawdzenia, czy nie jest None
print(f"Wynik + 10 to: {wynik + 10}")
```

e. Uruchom **mypy analiza.py**. Zobacz błąd: *"Item "None" of "float | None" has no attribute..."*.

f. Popraw kod w **analiza.py**, sprawdzając czy wynik nie jest **None**.

g. Uruchom **mypy** ponownie - teraz powinno być czysto.

Omówienie Zadania 8 - Zaawansowane Typowanie

Cel: Zrozumienie, jak typowanie wymusza obsługę przypadków brzegowych (None).

Kluczowe wnioski:

- **mypy** śledzi typy. „Wie”, że **wynik** może być **None**.
- Blok **if wynik is not None:** to tzw. **Type Narrowing**. Wewnątrz tego bloku **mypy** wie już, że **wynik** to na pewno **float**.
- Eliminujemy błędy **AttributeError: 'NoneType' object has no attribute....**

Zadanie 9 - Mierzenie Pokrycia Kodu (pytest-cov)

Problem: Mamy testy, ale nie wiemy, czy sprawdzają one każdą linijkę naszego kodu. Czy pominęliśmy jakiś **if**?

Rozwiązanie: pytest-cov - wtyczka raportująca pokrycie kodu.

Zadanie:

a. Instalacja: pip install pytest-cov.

b. Uruchom testy z raportem: pytest --cov=kalkulator.

c. Analiza: Zobaczysz, że pokrycie (Cov) nie wynosi 100%.

- Dlaczego? Bo dodaliśmy funkcję srednia w poprzednim zadaniu, ale nie dopisaliśmy do niej testów w test_kalkulator.py!

d. Uzupełnij testy: W test_kalkulator.py dodaj testy dla funkcji srednia:

- Przypadek listy z liczbami (np. [1, 2, 3] -> 2.0).
- Przypadek pustej listy ([] -> None).

e. Weryfikacja: Uruchom ponownie pytest --cov=kalkulator. Powinieneś zobaczyć 100%.

Omówienie Zadania 9 - Mierzenie Pokrycia Kodu

Cel: Zrozumienie metryki pokrycia kodu jako narzędzia do znajdowania luk w testach.

Wniosek: Pokrycie 100% nie gwarantuje braku błędów, ale pokrycie niskie gwarantuje, że mamy nieprzetestowany kod.

Podsumowanie

Co dzisiaj osiągnęliśmy?

- Zbudowaliśmy kompletną, dwuwarstwową siatkę bezpieczeństwa dla naszego kodu.

Filar 1: Testy Automatyczne (pytest)

- **Co robią?** Sprawdzają **logikę i zachowanie** kodu w czasie działania.
- **Jak?** Przez **assert**, **pytest.raises**, fixtures, parametryzację i mockowanie.
- **Odpowiadają na pytanie:** "Czy kod robi to, co powinien?".

Filar 2: Statyczne Typowanie (mypy)

- **Co robi?** Sprawdza poprawność użycia danych przed uruchomieniem.
- **Jak?** Przez adnotacje typów i analizę z mypy.
- **Odpowiada na pytanie:** "Czy dane pasują do siebie?".

Metryki Jakości (pytest-cov)

- **Co robią?** Mierzą, ile kodu jest faktycznie testowane.
- **Odpowiadają na pytanie:** "Czy o czymś nie zapomnieliśmy?".

Kluczowa Lekcja:

- Połączenie **pytest** i **mypy** to **standard w nowoczesnej inżynierii oprogramowania** w Pythonie.
- To te narzędzia dają nam **zaufanie** do kodu i pozwalają go rozwijać bez obawy, że coś zepsujemy.