Pytest

# Installation

python -m pip install pytest

# Miért annyira hasznos?

Ha már írtál egységteszteket a Python kódodhoz korábban, akkor talán használtad a beépített unittest modult. Az unittest szilárd alapot nyújt a tesztrendszer kialakításához, de néhány hiányossága van.

Számos harmadik féltől származó tesztelési keretrendszer megpróbálja kezelni az unittest néhány problémáját, és a pytest bizonyult az egyik legnépszerűbbnek. A pytest egy gazdag funkciókkal rendelkező, bővítményekre épülő ökoszisztéma a Python kódod teszteléséhez.

Ha még nem volt alkalmad használni a pytest-et, akkor kellemes meglepetésben lesz részed! A filozófiája és funkciói a tesztelési élményedet hatékonyabbá és élvezetesebbé teszik. A pytest segítségével az általános feladatok kevesebb kódot igényelnek, míg a bonyolultabb feladatokat több funkció is élhetőbbé teszi. Még a meglévő tesztek is futtathatók vele, beleértve az unittest vagy selenium segítségével írtakat is.

Ahogy a legtöbb keretrendszer esetén, néhány fejlesztési minta, amelyek eleinte ésszerűnek tűnnek a pytest használatakor, kellemetlenségeket okozhatnak, ahogy a tesztkészleted növekszik. Ezért fontos megismerni néhány pytest által biztosított eszközt, ami segít a tesztelést hatékonyan tartani az alkalmazások komplexitásának növekedése mellett is.

## Kevesebb boilerplate kód

A funkcionális tesztek általában az Arrange-Act-Assert modellt követik:

1. Arrange, vagyis felkészíti a teszt feltételeit
2. Act, vagyis meghív egy függvényt vagy metódust
3. Assert, vagyis ellenőrzi, hogy egy végfeltétel igaz-e

A tesztelési keretrendszerek általában illeszkednek (rálátnak a kód futására) a tesztek ellenőrzéseihez, így információt tudnak nyújtani, ha egy ellenőrzés meghiúsul. Az unittest például számos hasznos ellenőrzési segédeszközt tartalmaz alapértelmezetten. Azonban még egy kis tesztkészlet is sok boilerplate kódot igényel.

Képzeld el, hogy szeretnél írni egy tesztkészletet annak ellenőrzésére, hogy az unittest megfelelően működik-e a projektben. Lehet, hogy szeretnél írni egy olyan tesztet, amely mindig sikeres, valamint egy olyat, amely mindig meghiúsul:

# test\_with\_unittest.py

from unittest import TestCase

class TryTesting(TestCase):

def test\_always\_passes(self):

self.assertTrue(True)

def test\_always\_fails(self):

self.assertTrue(False)

A teszteket ezután futtathatod a parancssorból az unittest discover opciójával:

(venv) $ python -m unittest discover

F.

======================================================================

FAIL: test\_always\_fails (test\_with\_unittest.TryTesting)

----------------------------------------------------------------------

Traceback (most recent call last):

File "...\effective-python-testing-with-pytest\test\_with\_unittest.py",

line 10, in test\_always\_fails

self.assertTrue(False)

AssertionError: False is not true

----------------------------------------------------------------------

Ran 2 tests in 0.006s

FAILED (failures=1)

Ahogy vártad, egy teszt sikerült, míg egy másik meghiúsult. Bebizonyítottad, hogy az unittest működik, de ennyi lépést kellett hozzá megcsinálni:

1. Importálni az unittest-ből a TestCase osztályt
2. Létrehozni a TryTesting osztályt, amely a TestCase leszármazottja
3. Minden teszthez írni egy metódust a TryTesting osztályban
4. Az unittest.TestCase-ből egyik self.assert\* metódust használni az állítások megfogalmazásához

Ez jelentős mennyiségű kódot jelent, és mivel ez a minimális szükséges kód bármely teszthez, ugyanazt a kódot ismételgetnéd. A pytest egyszerűsíti ezt a munkafolyamatot, mert lehetővé teszi a szokásos függvények és a Python assert kulcsszó közvetlen használatát:

# test\_with\_pytest.py

def test\_always\_passes():

assert True

def test\_always\_fails():

assert False

Ennyi. Nem kell foglalkoznod semmilyen importtal vagy osztályokkal. Csak be kell foglalnod egy függvényt a test\_ előtaggal. Mivel használhatod az assert kulcsszót, nem kell megtanulnod vagy megjegyezned az összes különböző self.assert\* metódust az unittest-ben sem. Ha képes vagy olyan kifejezést írni, amelyet igaznak vársz, akkor a pytest tesztelni fogja azt helyetted.

A pytest nemcsak hogy kiküszöböli a sok boilerplate kódot, hanem részletesebb és könnyebben olvasható kimenetet is nyújt számodra.

## Jobb kimenet

A tesztkészleted futtathatod a pytest parancs segítségével a projekt felső szintű mappájából:

(venv) $ pytest

============================= test session starts =============================

platform win32 -- Python 3.10.5, pytest-7.1.2, pluggy-1.0.0

rootdir: ...\effective-python-testing-with-pytest

collected 4 items

test\_with\_pytest.py .F [ 50%]

test\_with\_unittest.py F. [100%]

================================== FAILURES ===================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_always\_fails \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_always\_fails():

> assert False

E assert False

test\_with\_pytest.py:7: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TryTesting.test\_always\_fails \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <test\_with\_unittest.TryTesting testMethod=test\_always\_fails>

def test\_always\_fails(self):

> self.assertTrue(False)

E AssertionError: False is not true

test\_with\_unittest.py:10: AssertionError

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

FAILED test\_with\_pytest.py::test\_always\_fails - assert False

FAILED test\_with\_unittest.py::TryTesting::test\_always\_fails - AssertionError:...

========================= 2 failed, 2 passed in 0.20s =========================

A pytest a unittest-től eltérően mutatja be a teszteredményeket, és a test\_with\_unittest.py fájl is feltüntetésre kerül. A report a következőket mutatja:

1. A rendszer állapotát, ideértve a telepített Python, pytest és bármely telepíthető bővítmény verzióját
2. A rootdir-t, azaz a konfiguráció és a tesztek kereséséhez használt könyvtárat
3. A futtató által felderített tesztek számát

Ezek az elemek az első részben jelennek meg a kimenetben:

============================= test session starts =============================

platform win32 -- Python 3.10.5, pytest-7.1.2, pluggy-1.0.0

rootdir: ...\effective-python-testing-with-pytest

collected 4 items

Az output ezután az egyes tesztek állapotát jelzi egy unittest-hez hasonló szintaxis segítségével:

* Egy pont (.) azt jelenti, hogy a teszt sikeres volt.
* Egy F azt jelenti, hogy a teszt bukott.
* Egy E azt jelenti, hogy a teszt váratlan (kezeletlen és elváratlan) kivételt váltott ki.

A speciális karakterek megjelennek a név mellett, a tesztkészlet általános haladását pedig jobbra mutató jelzésekkel jelzi:

test\_with\_pytest.py .F [ 50%]

test\_with\_unittest.py F. [100%]

A bukott tesztek esetén a jelentés részletes lebontást ad a hiba okáról. Az előző példában a tesztek azért hiúsultak meg, mert az assert False mindig meghiúsul:

================================== FAILURES ===================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_always\_fails \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_always\_fails():

> assert False

E assert False

test\_with\_pytest.py:7: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TryTesting.test\_always\_fails \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <test\_with\_unittest.TryTesting testMethod=test\_always\_fails>

def test\_always\_fails(self):

> self.assertTrue(False)

E AssertionError: False is not true

test\_with\_unittest.py:10: AssertionError

Ez a plusz kimenet rendkívül hasznos lehet hibakeresés közben. Végül a jelentés átfogó állapotjelentést ad a tesztkészletről:

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

FAILED test\_with\_pytest.py::test\_always\_fails - assert False

FAILED test\_with\_unittest.py::TryTesting::test\_always\_fails - AssertionError:...

========================= 2 failed, 2 passed in 0.20s =========================

Az unittest-hez képest a pytest kimenete sokkal informatívabb és olvashatóbb.

## Szabványos felépítés

Az assert kulcsszó használata is rendkívül erős. Ha már használtad korábban, akkor nincs új dolog, amit meg kellene tanulnod. Íme néhány példa az assert kifejezésre:

# test\_assert\_examples.py

def test\_uppercase():

assert "loud noises".upper() == "LOUD NOISES"

def test\_reversed():

assert list(reversed([1, 2, 3, 4])) == [4, 3, 2, 1]

def test\_some\_primes():

assert 37 in {

num

for num in range(2, 50)

if not any(num % div == 0 for div in range(2, num))

}

Az assert kifejezések nagyon hasonlítanak a szokásos Python függvényekre. Mindez azt eredményezi, hogy a pytest egyszerűen tanulható, mint az unittest, mert nincs szükség új szerkezetek megtanulására a kezdéshez.

Fontos megjegyezni, hogy minden teszt meglehetősen kicsi és önálló. Ez gyakori jelenség - hosszú függvénynévvel találkozhatsz, és a függvényen belül nem történik sok dolog. Ez főként arra szolgál, hogy az egyes tesztek elkülönüljenek egymástól, így ha valami meghibásodik, pontosan tudod, hol van a probléma. Egy szép következménye ennek, hogy a tesztek címkézése sokkal jobb lesz az outputban.

## Állapotok és függőségek kezelése

A tesztjeid gyakran függenek adattípusoktól vagy teszt-double-öktől, amelyek a kódodban valószínűleg előfordulnak, például dictionary-k vagy JSON fájlok.

Az unittest esetében ezeket a függőségeket .setUp() és .tearDown() metódusokba helyezheted, hogy a tesztosztály minden tesztje használhassa őket. Ezeket a speciális metódusokat használni is lehet, de ahogy a tesztosztályok egyre nagyobbak lesznek, könnyen előfordulhat, hogy a teszt függősége teljesen **implicit** lesz. Más szóval, ha csak az egyik izolált tesztet nézed, nem láthatod azonnal, hogy mire támaszkodik.

Az idővel kialakuló implicit függőségek bonyolult teszt halmazhoz vezethetnek (spagetti kód), amit ki kell bogoznod, hogy értelmezhetővé tedd a tesztjeidet. A teszteknek segíteniük kell a kódod jobb megértésében. Ha maguk a tesztek nehezen érthetőek, akkor baj van!

A pytest más megközelítést alkalmaz. Az **explicit** függőségdeklarációk felé terel, amelyek még mindig újrafelhasználhatók a fixture-ökön keresztül. A pytest fixture-ök olyan függvények, amelyek létrehozhatnak adatokat, teszt-duplákat vagy inicializálhatják a rendszerállapotot a tesztkészlet számára. Bármely teszt, amely szeretné használni a fixture-t, explicit módon meg kell adja ezt a fixture függvényt argumentumként a tesztfüggvénynek, így a függőségeket mindig az elején tüntethetjük fel:

# fixture\_demo.py

import pytest

@pytest.fixture

def example\_fixture():

return 1

def test\_with\_fixture(example\_fixture):

assert example\_fixture == 1

A tesztfüggvényre nézve azonnal láthatod, hogy függ egy fixture-től, anélkül, hogy az egész fájlt kellene átnézned a fixture definíciók miatt.

Megjegyzés: Általában érdemes a teszteket a saját tests nevű mappájukba helyezni a projekt gyökérkönyvtárában.

A fixture-ek használhatnak más fixture-eket is, újra azokat explicit módon deklarálva függőségként. Ez azt jelenti, hogy idővel a fixture-ek terjedelmessé és modulárisabbá válhatnak. Bár a lehetőség, hogy fixture-eket helyezzünk be más fixture-ekbe, hatalmas rugalmasságot nyújt, ugyanakkor nehezebbé teheti a függőségek kezelését, ahogy a tesztkészlet nő.

## Tesztek csoportosítása

Ahogy nő a tesztkészleted, előfordulhat, hogy csak néhány tesztet szeretnél futtatni egy adott funkcióhoz, és a teljes készletet máskor futtatnád csak. A pytest néhány módot biztosít erre:

* Név alapú szűrés: Korlátozhatod a pytestet arra, hogy csak azokat a teszteket fusson, amelyek teljesen kvalifikált nevei megegyeznek egy adott kifejezéssel. Ezt a -k paraméterrel teheted meg.
* Könyvtár hatáskörzése: Alapértelmezés szerint a pytest csak azokat a teszteket futtatja, amelyek a jelenlegi könyvtárban vagy annak alatt vannak.
* Teszt kategorizálása: A pytest be tudja fogadni vagy kizárni a teszteket a saját általad definiált kategóriákból. Ezt a -m paraméterrel teheted meg.

A teszt kategorizálása egy finom, de erőteljes eszköz. A pytest lehetővé teszi számodra, hogy címkéket, vagy egyéni jelöléseket hozz létre bármelyik tesztedhez. Egy teszt több jelöléssel is rendelkezhet, és ezeket használhatod a tesztek pontos irányításához, hogy melyeket futtasd.

## Parametrizáció

Amikor olyan függvényeket tesztelsz, amelyek adatokat dolgoznak fel vagy általános átalakításokat végeznek, gyakran hasonló tesztek írásán kapod magad. Például a tesztek csak az ellenőrzött kód bemenetében vagy kimenetében térnek el egymástól. Ez azt jelenti, hogy a tesztkód ismétlődik, és ez néha eltakarhatja azt a viselkedést, amit tesztelni szeretnél.

Az unittest lehetőséget nyújt arra, hogy több tesztet egybe gyűjts egy csoportba, de ezek nem jelennek meg különálló tesztként a jelentésekben. Ha egy teszt hibás, de a többi sikeres, akkor az egész csoport továbbra is egyetlen hibás eredményt ad. A pytest saját megoldást kínál, amelyben minden teszt önállóan sikeres vagy sikertelen lehet.

# Fixtúrák

A pytest fixture-ek olyan módszer, amellyel adatokat, teszt duplákat vagy állapotbeállításokat biztosíthatsz a tesztjeidhez. A fixture-ek olyan függvények, amelyek különböző értékeket tudnak visszaadni. Minden teszt, amely függ egy fixture-től, explicit módon meg kell fogadja azt a fixture-t argumentumként.

## Mikor csináljunk fixtúrákat?

Ebben a részben egy tipikus tesztvezérelt fejlesztés (TDD) munkamenetet fogunk tekinteni:

Képzeld el, hogy egy olyan függvényt írsz, a format\_data\_for\_display()-t, amely az API végpont által visszaadott adatokat dolgozza fel. Az adatok emberek listáját jelentik, mindegyiknek adott neve, családi neve és munkaköre van. A függvénynek olyan stringek listáját kell eredményeznie, amelyek minden személy teljes nevét tartalmazzák (a given\_name után a family\_name), kettőspontot és a munkakört:

# format\_data.py

def format\_data\_for\_display(people):

... # Implement this!

Jó TDD stílusban először tesztet szeretnél írni. Erre például a következő kódot írhatod:

# test\_format\_data.py

def test\_format\_data\_for\_display():

people = [

{

"given\_name": "Alfonsa",

"family\_name": "Ruiz",

"title": "Senior Software Engineer",

},

{

"given\_name": "Sayid",

"family\_name": "Khan",

"title": "Project Manager",

},

]

assert format\_data\_for\_display(people) == [

"Alfonsa Ruiz: Senior Software Engineer",

"Sayid Khan: Project Manager",

]

A teszt írása közben felmerül, hogy lehet, hogy szükséged lesz egy másik függvényre is, amely átalakítja az adatokat vesszővel elválasztott értékekkel az Excelben való használathoz:

# format\_data.py

def format\_data\_for\_display(people):

... # Implement this!

def format\_data\_for\_excel(people):

... # Implement this!

Növekszik a teendőid listája! Ez jó! A TDD előnye az, hogy segít megtervezni a következő feladatokat. A format\_data\_for\_excel() függvény tesztje nagyon hasonlítana a format\_data\_for\_display() függvény tesztjére:

# test\_format\_data.py

def test\_format\_data\_for\_display():

# ...

def test\_format\_data\_for\_excel():

people = [

{

"given\_name": "Alfonsa",

"family\_name": "Ruiz",

"title": "Senior Software Engineer",

},

{

"given\_name": "Sayid",

"family\_name": "Khan",

"title": "Project Manager",

},

]

assert format\_data\_for\_excel(people) == """given,family,title

Alfonsa,Ruiz,Senior Software Engineer

Sayid,Khan,Project Manager

"""

Figyelemre méltó, hogy mindkét tesztnek meg kell ismételnie a people változó definícióját, ami elég sok sor kódot jelent.

Ha több olyan tesztet írsz, amelyek mind ugyanazt az alapvető tesztadatot használják, akkor a jövőben lehet, hogy érdemes használni egy fixture-t. A megismételt adatokat egyetlen függvénybe helyezheted, amelyet a @pytest.fixture dekorátorral lát el, hogy jelezze, hogy a függvény egy pytest fixture:

# test\_format\_data.py

import pytest

@pytest.fixture

def example\_people\_data():

return [

{

"given\_name": "Alfonsa",

"family\_name": "Ruiz",

"title": "Senior Software Engineer",

},

{

"given\_name": "Sayid",

"family\_name": "Khan",

"title": "Project Manager",

},

]

# ...

A fixture-t hozzáadhatod a tesztekhez, mint egy függvényreferenciát. Fontos megjegyezni, hogy nem hívod meg a fixture függvényt. Erről a pytest gondoskodik. De a fixture függvény visszatérési értékét a fixture függvény nevével elérheted a tesztekben:

# test\_format\_data.py

# ...

def test\_format\_data\_for\_display(example\_people\_data):

assert format\_data\_for\_display(example\_people\_data) == [

"Alfonsa Ruiz: Senior Software Engineer",

"Sayid Khan: Project Manager",

]

def test\_format\_data\_for\_excel(example\_people\_data):

assert format\_data\_for\_excel(example\_people\_data) == """given,family,title

Alfonsa,Ruiz,Senior Software Engineer

Sayid,Khan,Project Manager

"""

Most minden teszt jóval rövidebb, de még mindig egyértelműen hivatkozik az adatokra, amelyekre támaszkodik. Győződj meg arról, hogy a fixture-t egy specifikus névvel látod el. Így gyorsan el tudod dönteni, hogy a jövőben melyik tesztekhez szeretnéd használni!

Amikor először felfedezed a fixture-e-et, könnyű túlzásba vinni a használatukat, de mint minden másban, itt is fenntartani kell az egyensúlyt.

## Mikor ne csináljunk fixtúrákat?

Fixture-k nagyszerűek arra, hogy kinyerjük azokat az adatokat vagy objektumokat, amelyeket több tesztben is használunk. Azonban nem mindig ideálisak azok a tesztek számára, amelyek kis változtatásokat igényelnek az adatokban. A tesztfájl telepakolása fixture-ekkel nem jobb, mint ha sima adatokkal vagy objektumokkal lenne tele. Sőt, lehet, hogy még rosszabb, mert hozzáad egy réteget az indirekcióhoz.

Mint a legtöbb absztrakciónál, itt is gyakorlásra és gondolkodásra van szükség ahhoz, hogy megtaláljuk a megfelelő fixture használati szintet.

Mindazonáltal a fixture-ek valószínűleg az alapvető részét fogják képezni a tesztcsomagnak. Ahogy a projekt terjedelme növekszik, úgy kezd felmerülni a skálázódás kihívása. Mindenféle eszköz szembesül azzal a kihívással, hogy hogyan kezeli a növekedésből adódó komplexitást, és szerencsére a pytest rendelkezik néhány hasznos funkcióval, amelyek segítenek benne.

## Skálázás

Ahogy egyre több fixture-t kinyersz a tesztekből, előfordulhat, hogy néhány fixture további absztrakcióval lenne jól használható. A pytest-ben a fixture-ek modulárisak. A modularitás azt jelenti, hogy a fixture-ek importálhatók, importálhatnak más modulokat, és függhetnek és importálhatnak más fixture-öket. Mindez lehetővé teszi számodra, hogy összeállíts egy megfelelő fixture absztrakciót az adott felhasználási esetedhez.

Például előfordulhat, hogy két különálló fájlban vagy modulban lévő fixture közös függőséget használ. Ebben az esetben áthelyezheted a fixture-eket a tesztmodulokból általánosabb fixture-rel kapcsolatos modulokba. Így visszaimportálhatod őket bármelyik tesztmodulba, amelynek szüksége van rájuk. Ez jó megközelítés, ha folyamatosan használod a fixture-t a projektben.

Ha szeretnéd, hogy egy fixture az egész projektben elérhető legyen anélkül, hogy importálnád, a conftest.py nevű különleges konfigurációs modul ezt lehetővé teszi számodra.

A pytest a keres a conftest.py modult minden könyvtárban. Ha hozzáadod az általános célú fixture-eket a conftest.py modulhoz, akkor azt a fixture-t használhatod a modul szülőkönyvtárában és az összes almappában anélkül, hogy importálnád. Ez egy remek hely a leggyakrabban használt fixture-ek elhelyezésére.

Egy másik érdekes felhasználási eset a fixture-ök és a conftest.py használata a hozzáférés korlátozása erőforrásokhoz. Képzeld el, hogy írtál egy tesztcsomagot olyan kódhoz, amely API-hívásokkal foglalkozik. Biztosítani szeretnéd, hogy a tesztcsomag ne hajtson végre valós hálózati hívásokat, még akkor sem, ha valaki véletlenül olyan tesztet ír, ami azt teszi.

A pytest biztosít egy [monkeypatch](https://docs.pytest.org/en/latest/how-to/monkeypatch.html) fixture-t az értékek és viselkedések helyettesítésére:

# conftest.py

import pytest

import requests

@pytest.fixture(autouse=True)

def disable\_network\_calls(monkeypatch):

def stunted\_get():

raise RuntimeError("Network access not allowed during testing!")

monkeypatch.setattr(requests, "get", lambda \*args, \*\*kwargs: stunted\_get())

A disable\_network\_calls() helyezése a conftest.py-ba és az autouse=True opció hozzáadása biztosítja, hogy a hálózati hívások minden tesztben le legyenek tiltva az egész tesztcsomagban. Bármely olyan teszt, amely végrehajtja a requests.get() hívást, RuntimeError kivételt dob, ami azt jelzi, hogy váratlan hálózati hívás történt volna.

# Mark: tesztek csoportosítása

Egy nagy teszt készletben hasznos elkerülni az összes teszt futtatását, amikor gyorsan szeretnél tesztelni egy új funkciót. A pytest alapértelmezett működésén kívül, ami az összes teszt futtatását jelenti a jelenlegi munkakönyvtárban, illetve a szűrési funkcionalitáson kívül használhatod a „markereket”.

A pytest lehetővé teszi, hogy meghatározz kategóriákat a tesztek számára, és lehetőséget biztosít azok be- vagy kizárására a tesztkészlet futtatásakor. Tetszőleges számú kategóriával elláthatod a tesztet.

A tesztek jelölése (mark) hasznos lehet a tesztek besorolásához a részrendszer vagy a függőségek szerint. Például, ha néhány tesztedhez hozzáférésre van szükség az adatbázishoz, akkor létrehozhatsz egy @pytest.mark.database\_access markot számukra.

Pro tipp: Mivel tetszőleges nevet adhatsz a jeleknek, könnyű elgépelni vagy rosszul emlékezni a jel nevére. A pytest figyelmeztet, ha olyan jeleket talál a teszt kimenetében, amelyeket nem ismer fel.

A pytest parancshoz használhatod a --strict-markers jelzőt, hogy megbizonyosodj arról, hogy az összes jel a tesztekben regisztrálva van a pytest konfigurációs fájlban, pytest.ini. Ez megakadályozza a tesztek futtatását, amíg regisztrálsz minden ismeretlen jelet.

További információkért a jelek regisztrálásáról tekintsd meg a pytest dokumentációját.

Amikor eljön az idő a tesztek futtatására, az összes tesztet lefuttathatod alapértelmezés szerint a pytest paranccsal. Ha csak azokat a teszteket szeretnéd futtatni, amelyekhez adatbázis hozzáférés szükséges, akkor használhatod a *pytest -m database\_access* parancsot. Az összes teszt futtatása kivéve azokat, amelyekhez adatbázis hozzáférés szükséges, a *pytest -m "not database\_access"* paranccsal érhető el. Sőt, használhatsz egy autouse fixture-t, hogy korlátozd az adatbázis-hozzáférést azokra a tesztekre, amelyeket a database\_access jelöléssel lát el (lásd fentebb).

Néhány plugin kibővíti a mark funkcionalitást a saját védelmeivel (guard). Például a pytest-django plugin biztosít egy django\_db jelet. Azok a tesztek, amelyeknek hiányzik ez a jelölésük, és megpróbálnak hozzáférni az adatbázishoz, hibát okoznak. Az első teszt, amely megpróbálja hozzáférni az adatbázishoz, előidézi a Django tesztadatbázisának létrehozását.

A követelmény, hogy hozzáadd a django\_db jelölést, arra ösztönöz, hogy nyíltan megfogalmazd a függőségeidet. Ez a pytest filozófiája, végül is! Ez azt is jelenti, hogy sokkal gyorsabban futtathatod a teszteket, amelyek nem támaszkodnak az adatbázisra, mert a pytest -m "not django\_db" megakadályozza az adatbázis létrehozását. Az időmegtakarítás nagyon jelentős lehet, különösen, ha gyakran futtatod a teszteket.

A pytest néhány markot alapértelmezés szerint kínál:

* a skip feltétel nélkül kihagyja a tesztet.
* a skipif kihagyja a tesztet, ha az a kifejezés, amit átadnak neki, igazra értékelődik ki.
* az xfail jelzi, hogy egy teszt várhatóan nem sikerül, így ha a teszt ténylegesen nem sikerül, az összes teszt sorozat még mindig sikeres eredményt adhat.
* parametrize több változatot hoz létre a tesztnek, különböző értékekkel argumentumokként. Bővetteb lásd lentebb.

Az összes pytest által ismert jelről egy listát láthatsz a pytest --markers paranccsal.

# Parametrizálás: tesztek kombinálása

Korábban láttuk, hogyan lehet a pytest fixture-eket használni a kódduplikáció csökkentésére a közös függőségek kiemelésével. Azonban a fixture-ek nem annyira hasznosak, amikor több teszted van, amelyeknek kissé eltérő bemeneteik és várt kimeneteik vannak. Ilyen esetekben egyetlen teszt definíciót parametrizálhatsz, és a pytest létrehoz neked olyan tesztvariációkat, amelyekhez a megadott paramétereket használja.

Képzelj el egy függvényt, amely megállapítja, hogy egy karakterlánc palindrom e. Az első készlet teszt így nézhet ki:

def test\_is\_palindrome\_empty\_string():

assert is\_palindrome("")

def test\_is\_palindrome\_single\_character():

assert is\_palindrome("a")

def test\_is\_palindrome\_mixed\_casing():

assert is\_palindrome("Bob")

def test\_is\_palindrome\_with\_spaces():

assert is\_palindrome("Never odd or even")

def test\_is\_palindrome\_with\_punctuation():

assert is\_palindrome("Do geese see God?")

def test\_is\_palindrome\_not\_palindrome():

assert not is\_palindrome("abc")

def test\_is\_palindrome\_not\_quite():

assert not is\_palindrome("abab")

Az utolsó két teszten kívül az összes tesztnek ugyanolyan a felépítése:

def test\_is\_palindrome\_<in some situation>():

assert is\_palindrome("<some string>")

Ez kezd úgy kinézni, mintha kötelező elemekből készült sablon lenne. A pytest eddig segített megszabadulni a sablonkódoktól, és most sem hagy cserben. A @pytest.mark.parametrize() dekorátort használhatod annak érdekében, hogy különböző értékekkel töltse ki ezt a felépítést, jelentősen csökkentve ezzel a tesztkódodat:

@pytest.mark.parametrize("palindrome", [

"",

"a",

"Bob",

"Never odd or even",

"Do geese see God?",

])

def test\_is\_palindrome(palindrome):

assert is\_palindrome(palindrome)

@pytest.mark.parametrize("non\_palindrome", [

"abc",

"abab",

])

def test\_is\_palindrome\_not\_palindrome(non\_palindrome):

assert not is\_palindrome(non\_palindrome)

A parametrize() első argumentuma egy vesszővel elválasztott string, amely a paraméter neveket tartalmazza. Nem kell több nevet megadnod, ahogy ebben a példában is láthatod. A második argumentum egy lista, amely tartalmazhat tuple-ket vagy egyedi értékeket, amelyek a paraméter(ek) értékét képviselik. Még tovább viheted a parametrizációt, és kombinálhatod az összes tesztet egybe:

@pytest.mark.parametrize("maybe\_palindrome, expected\_result", [

("", True),

("a", True),

("Bob", True),

("Never odd or even", True),

("Do geese see God?", True),

("abc", False),

("abab", False),

])

def test\_is\_palindrome(maybe\_palindrome, expected\_result):

assert is\_palindrome(maybe\_palindrome) == expected\_result

Bár ez rövidítette a kódodat, fontos megjegyezni, hogy ebben az esetben valójában elvesztetted az eredeti függvények részletesebb jellegét. Győződj meg arról, hogy nem teszed a parametrizálással érthetetlenebbé a tesztkészletedet.

A parametrizációval elválaszthatod a tesztadatot a tesztkészlet viselkedésétől, hogy világos legyen, mit tesztelsz, és hogy a különböző tesztesetek könnyebben olvashatóak és karbantarthatóak legyenek.

# Sebesség!

Amikor a teszteket használni akarjuk például implementáció közben, megállunk a fejlesztéssel. Ha a tesztek lassan futnak, az azt tudja eredményezni, hogy teszt-fejlesztés-teszt-fejlesztés váltások kizökkentik a fejlesztőt a munkából. Hajlamosak vagyunk ilyenkor a teszteket egyre ritkábban használni.

Korábban láttuk, hogyan használhatunk markokat a lassú tesztek kiszűrésére a tesztkészlet futtatásakor, de előbb-utóbb szükségünk lesz arra, hogy lefuttasuk azokat is. Ha szeretnénk felgyorsítani a tesztek sebességét, hasznos tudni, mely tesztek azok, amelyek kiemelkedően lassúak.. A pytest automatikusan rögzíti a tesztek időtartamát és jelentheti a leglassabbakat.

Használd a --durations opciót a pytest parancsban a teszteredményeidben megjelenő időtartam-jelentésért. A --durations egy egész szám értéket, n-t vár, és jelenteni fogja a leglassabb n darab tesztet. Egy új szakasz lesz látható a teszteredményekben:

(venv) $ pytest --durations=5

...

============================= slowest 5 durations =============================

3.03s call test\_code.py::test\_request\_read\_timeout

1.07s call test\_code.py::test\_request\_connection\_timeout

0.57s call test\_code.py::test\_database\_read

(2 durations < 0.005s hidden. Use -vv to show these durations.)

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

...

Minden teszt, amely megjelenik az időtartam-jelentésben, jó jelölt a gyorsításra, mert az átlagos tesztelési időhöz képest lassú. Érdemes megjegyezni, hogy a rövid időtartamok alapértelmezetten el vannak rejtve a reportokban. Ahogy a jelentésben is ki van írva, növelheted a jelentés részletességét, és megjelenítheted ezeket a rövid időtartamokat a –vv (very verbose) kapcsolóval együtt a --durations opcióval.

Fontos tudni, hogy néhány tesztnek láthatatlan beállítási overhead-je lehet. Korábban olvastál arról, hogy az első, django\_db jelzéssel ellátott teszt előidézi a Django tesztadatbázisának létrehozását. Az időtartam-jelentés tükrözi az adatbázis beállításához szükséges időt, a tesztben, amely előidézte az adatbázis létrehozását, ami félrevezető lehet.

# Case study

## Pytest

Az egységtesztelés egy olyan tesztelési technika, amely a forráskódot felismerhető egységekre bontja (általában metódusokra vagy függvényekre) és azokat egyenként teszteli.

Alapvetően egy olyan szkriptet vagy szkriptcsoportot fogsz írni, amely minden metódust különböző bemenetekkel tesztel annak érdekében, hogy minden logikai ágat teszteljünk az adott metóduson belül. Ezt hívjuk kódlefedettségnek, és általában 100% kódlefedettségre törekszünk. Ez nem mindig szükséges vagy gyakorlatias (vagy lehetséges), de ezt most nem kerül további tárgyalásra.

Minden teszt elkülönülten kezeli a tesztelt metódust: a külső hívásokat egy ún. mockolási technika segítségével felülírjuk, hogy megbízható visszatérési értékeket kapjunk, és a teszt előtt beállított objektumokat a teszt után eltávolítjuk. Ezeket a technikákat és másokat is azért alkalmazzuk, hogy biztosítsuk a tesztelt egység függetlenségét és elkülönültségét.

Itt van néhány alapvető dolog, ami segíteni fog a PyTest használatát elkezdeni.

Alább látható egy tipikus mappastruktúra egy Python projekt/alkalmazás számára, ahol van egy tests mappa, ami a src mappán kívül helyezkedik el.

.

├── docs # Documentation files (alternatively `doc`)

├── src # Source files (alternatively `lib` or `app`)

├── tests # Automated tests (alternatively `test`)

└── README.md

Tegyük fel, hogy a src mappában található calc.py fájl egyszerű add függvényét szeretnénk tesztelni.

# src/calc.py

def add(x, y):

"""Add Function"""

return x + y

Hozz létre egy test\_calc.py nevű fájlt a tests mappában. A tesztfájl neve mindig kezdődjön vagy végződjön a "test" szóval. Én azt preferálom, hogy a struktúrát konzisztens módon tartom, például a test\_xxx.py formátumot használom, ahol az "xxx" a tesztelt fájl neve. További információkat találhatsz a [dokumentációban](about:blank) a pytest test discovery működéséről.

Írd meg a tesztesetet az add add nevű függvényhez. Amikor a teszteseteket írod, egyszerűen definiálj egy olyan függvényt, amely a "test" szóval kezdődik a nevében. Hasonlóan a fájlnévhez, én azt preferálom, hogy a struktúrát konzisztens módon tartsam, például a test\_xxx formátumot használom, ahol az "xxx" a tesztelt függvény neve. Ez nagyon egyértelmű értelmezést tesz lehetővé az olvasónak.   
Kövessük az "Arrange-Act-Assert" mintázatot a teszttartalom strukturálásához. Bár ez a példa nagyon egyszerű és egyértelmű, és egy soros assert kifejezéssel helyettesíthető, de most használjuk ezt a bemutatás céljával.

* Az **arrange** az input és a tesztelt dolog elrendezése: Szükségesek-e speciális beállítások a teszthez? Szükséges-e előkészíteni az adatbázist? Az idő nagy részében szükség van az inputok előkészítésére és a függőségek mockolására az Act lépés előtt.
* Az **act** a tesztelendő rész futtatása: Ez általában a függvény vagy metódus meghívására vonatkozik az egységtesztelési forgatókönyvben.
* Az **assert** a várt eredmények ellenőrzése: Az act lépéstől kapjuk az aktuális eredményeket. Az assert lépés ellenőrzi ezeknek az eredményeknek a helyességét vagy helytelenségét. Ez lehet számok/szövegek helyességének ellenőrzése, bizonyos típusú kivétel kiváltódásának ellenőrzése vagy bizonyos függvény meghívásának ellenőrzése. Az assert kifejezések végül meghatározzák, hogy a teszt sikeres vagy sikertelen-e.

Az assert kifejezéseknél (a "output == expected" rész) általában bármely logikai feltételt használhatsz, amit használnál egy if utasítás írásakor.

Nem kizárólag az Arrange-Act-Assert mintázat használható. Néha összevonható az Act és az Assert egy sorban például. Azonban ha a felépítés kezd feltűnően felbomlani, érdemes megállni és gondolkozni, aztán korrigálni.

import pytest

from src.calc import add

def test\_add():

# Arrange

a = 2

b = 5

expected = 7

# Act

output = add(a, b)

# Assert

assert output == expected​

Megjegyzés: Mindig a kimenetet helyezzük a bal oldalra, és az elvárt kimenetet a jobb oldalra, azaz assert output == expected, ne assert expected == output. Ez nincs hatással a Terminál/CMD-ra. De pl a PyCharm rosszul jeleníti meg, ha fordítva tesszük.

Hogyan futtassuk a PyTest-et?

A PyTest teszteseteket bármelyik alábbi parancs segítségével futtathatod.

# run all tests

python -m pytest tests

# run single test file

python -m pytest tests/test\_calc.py

# run single test case

python -m pytest tests/test\_calc.py::test\_add​

Próbálj ki minél több helyzetet, ahol a függvényt tesztelned kell, hogy minden aspektusát lefedd.

## Mock

Korábbi részekben megtanultuk, hogy milyen egy jó egységteszt. A mockolás pedig az a technika, amely segítségével független, gyorsan ismételhető teszteseteket írhatunk.

A mockolás arra szolgál, hogy elszigeteljük és a figyelmünket a tesztelendő kódra összpontosítsuk, és ne az külső függőségek viselkedésére vagy állapotára. A mockolás olyan gondosan kezelt helyettesítő objektumokkal helyettesíti a függőségeket, amelyek a valódiak viselkedését imitálják.

### Installation

python -m pip install pytest-mock

### Példák

Kezdjünk egy egyszerű mockolási példával.

##### Példa 1:

Van egy sleep nevű függvényünk, amely pár másodpercig várakozik. Tegyük fel, hogy a várakozás után van néhány más feldolgozási lépésünk.

def sleep\_awhile(duration):

"""sleep for couple of seconds"""

time.sleep(duration)

# some other processing steps

Így néz ki a mockolt teszteset.

def test\_sleep\_awhile(mocker):

m = mocker.patch("src.example.time.sleep", return\_value=None)

sleep\_awhile(3)

m.assert\_called\_once\_with(3)

1. A teszteset bemeneteként a mocker-t kell használnunk, hogy hívhassuk a mocker.patch függvényt.
2. Létrehozunk egy hamis (mock) objektumot, amivel helyettesítjük a time modult. Ehhez megadjuk a célt: "src.example.time.sleep", ami azt jelenti, hogy a time.sleep függvényt helyettesítjük a jelenlegi teszt file-ban (src/example.py). Itt jön a mockolás alapszabálya: **Mockoljuk ott, ahol használjuk, és ne ott, ahol definiáljuk (forrás).**   
   Ez az objektum egyelőre nem csinál semmit, de beállíthatjuk, hogy hogy működjön. A sleep függvény nem ad vissza semmit, ezért a return\_value-t None-ra állítjuk.
3. Ehhez a sleep\_awhile függvényhez nincs kimenet megadva, így azt nem tudjuk ellenőrizni. Hogyan tudjuk tehát megbizonyosodni arról, hogy a teszteset megfelelően van megírva? Ellenőrizzük, hogy a mock objektumot a helyes bemenettel hívták-e meg az assert\_called\_once\_with függvénnyel. És a teszt időtartama nem kell, hogy 3 másodperc legyen. (azt akarjuk, hogy gyors legyen a tesztünk)   
   Megjegyzés: Ez az assert\_called\_once\_with függvény már önmagában egy ellenőrző függvény, így nem használjuk az "assert" kulcsszót újra előtte. Van még egy pár ilyen assert függvény, amit a mock objektumokhoz használatunk, például assert\_called(), assert\_any\_call(), assert\_not\_called(), stb. Érdemes a [dokumentációt](https://docs.python.org/3/library/unittest.mock.html) olvasni.

#### Példa 2:

Van egy get\_time\_of\_day nevű függvényünk a src/example.py fájlban, amely megadja nekünk, hogy milyen időszak van éppen. A visszatérési értéke a "Night/Morning/Afternoon/Evening" string lesz, attól függően, hogy az óra melyik tartományba esik.

# src/example.py

from datetime import datetime

def get\_time\_of\_day():

"""return string Night/Morning/Afternoon/Evening depending on the hours range"""

time = datetime.now()

if 0 <= time.hour <6:

return "Night"

if 6 <= time.hour < 12:

return "Morning"

if 12 <= time.hour <18:

return "Afternoon"

return "Evening"

Hogyan tesztelnénk ezt?

Ha csak egyszerűen definiálunk egy tesztesetet 10 órára, akkor az nem lenne jó délután. A teszt eredménye nagymértékben függne attól, hogy mikor futtatjuk a tesztet. És a teszteset nem lenne ismételhető.

Ezért szükségünk van arra, hogy „rögzítsük az időt”, bármikor is futnak tesztesetek. Nézzük meg, hogyan tehetjük ezt meg a mockolás segítségével.

Mockoljuk a *datetime* objektumot és a *now* függvény visszatérését

A mocker-t kell hozzáadnunk a függvény bemenetéhez. A mocker.patch függvényt használjuk, ahol a "src.example.datetime" az a rész, amelyet mockolni szeretnénk.

Felmerülhet a kérdés, hogy miért nem csak "datetime.datetime". A "src.example" itt a fájl elérési útjára utal, a ".datetime" pedig a könyvtárra/a get\_time\_of\_day függvényen belül használt részre utal.

A mock\_obj.function.return\_value segítségével határozzuk meg, hogy milyen visszatérési értéket szeretnénk helyettesíteni a függvény számára. (Ez több réteget is magába foglalhat, például mock\_obj.another\_obj.function.return\_value.) Itt a datetime.now() függvény visszatérési értékét 2 órára rögzítettük, ezért azt várjuk, hogy az eredmény "Afternoon" legyen.

import pytest

from datetime import datetime

from src.example import get\_time\_of\_day

def test\_get\_time\_of\_day(mocker):

mock\_now = mocker.patch("src.example.datetime")

mock\_now.now.return\_value = datetime(2016, 5, 20, 14, 10, 0)

assert get\_time\_of\_day() == "Afternoon"

A mockolást parametrizálással kombinálhatjuk tovább. Csak azt kell biztosítanunk, hogy a teszt függvény bemenete a parametrizálás bemeneteivel kezdődjön, majd a mocker következzen, például: def test\_get\_time\_of\_day(datetime\_obj, expect, mocker).

import pytest

from datetime import datetime

from src.example import get\_time\_of\_day

@pytest.mark.parametrize(

"datetime\_obj, expect",

[

(datetime(2016, 5, 20, 0, 0, 0), "Night"),

(datetime(2016, 5, 20, 1, 10, 0), "Night"),

(datetime(2016, 5, 20, 6, 10, 0), "Morning"),

(datetime(2016, 5, 20, 12, 0, 0), "Afternoon"),

(datetime(2016, 5, 20, 14, 10, 0), "Afternoon"),

(datetime(2016, 5, 20, 18, 0, 0), "Evening"),

(datetime(2016, 5, 20, 19, 10, 0), "Evening"),

],

)

def test\_get\_time\_of\_day(datetime\_obj, expect, mocker):

mock\_now = mocker.patch("src.example.datetime")

mock\_now.now.return\_value = datetime\_obj

assert get\_time\_of\_day() == expect​

Most ismerünk néhány módszert, amelyeket a harmadik féltől származó könyvtárakat kezeljünk a tesztjeinkben. Nézzünk meg néhány másik példát különböző esetekre.

#### Függvény egy másik fájlból

Tegyük fel, hogy van egy load\_data nevű függvény, amely betölti az adatokat és visszaadja azokat. A time.sleep függvényt használjuk annak szimulálására, hogy mennyi időbe telik például az adatok betöltése az adatbázisból:

# src/dataset.py

import time

def load\_data():

time.sleep(4)

# loading data...

return {"key1":"val1", "key2":"val2"}

Van továbbá egy process\_data() függvény, amely betölti az adatkészletet és bizonyos lépésekkel feldolgozza azt (amelyeket itt kihagyunk). Ezután feldolgozott eredményt ad vissza, itt feltételezzük, hogy ez data["key1"].

# src/example.py

def process\_data():

data = load\_data()

# process the data in certain ways ...

processed\_data = data["key1"]

return processed\_data​

Emlékszel a szabályra? Mockoljuk azt, ahol használjuk, és ne azt, ahol definiáljuk (forrás).

Ennek teszteléséhez mockolnunk kell a "src.example.load\_data" függvényt, és nem a "src.dataset.load\_data" függvényt.

def test\_process\_data(mocker):

mocker.patch("src.example.load\_data", return\_value={"key1": "val1", "key2": "val2"})

assert process\_data() == "valy"

#### Függvény osztályban (akár az init függvény is)

Legyen most egy *DBConnector* nevű osztályunk, amely inicializálja a kapcsolatot az adatbázishoz a init függvényben, és adatokat kap az azonosító (*id*) alapján az adatbázisból.

# src/db\_connection.py

import time

class DBConnector:

def \_\_init\_\_(self):

# setup some db connection

time.sleep(3)

pass

def get(self, id):

time.sleep(5)

return 'some data'

Van egy Engine osztályunk, amely tartalmazza a DBConnector-t attribútumként.

class Engine:

def \_\_init\_\_(self):

self.connector = DBConnector()

def load\_data(self):

data = self.connector.get(123)

print(data)

# do some processing

data = data + "xxx"

return data​

Hogyan tesztelhetjük az Engine.load\_data függvényt? Amikor a init függvényt mockoljuk, a return\_value-t None-ra kell állítani.

def test\_engine\_load\_data(mocker):

mocker.patch("src.example.DBConnector.\_\_init\_\_",return\_value = None)

mocker.patch("src.example.DBConnector.get",return\_value = 'xyz')

output = Engine().load\_data()

assert output == 'xyzxxx'

API

Egy példa a requests könyvtáron keresztüli *get* kérés végrehajtására.

#src/employee.py

class Employee:

"""A sample Employee class"""

def \_\_init\_\_(self, first, last, pay):

self.first = first

self.last = last

self.pay = pay

def monthly\_schedule(self, month):

response = requests.get(f'http://company.com/{self.last}/{month}')

if response.ok:

return response.text

else:

return 'Bad Response!'

Mockoljuk a használt response objektum attribútumokat: *ok* és *text*, az alábbiak szerint.

# test\_employee.py

import pytest

from src.employee import Employee

emp\_1 = Employee("Corey", "Schafer", 50000)

def test\_mock\_api\_call(mocker):

mock\_requests = mocker.patch("requests.get")

mock\_requests.return\_value.ok = True

mock\_requests.return\_value.text = "Success"

schedule = emp\_1.monthly\_schedule("May")

mock\_requests.assert\_called\_with("http://company.com/Schafer/May")

assert schedule == "Success"

#### Környezeti változó (monkeypatch.setenv)

Néha a tesztekhez környezeti beállításokra van szükség, amelyek lehetnek adatbázis kapcsolat, hálózati hozzáférés vagy környezeti változó beállítás.

A monkeypatch fixture segít biztonságosan beállítani/törölni egy attribútumot, dictionary elemet vagy környezeti változót, vagy módosítani a sys.path-t annak importálásakor.

<https://docs.pytest.org/en/6.2.x/monkeypatch.html>

def use\_env\_var():

contract\_class = os.environ['CONTRACT\_CLASS']

if contract\_class == 'en\_cloud':

# do some processing

return "this is en\_cloud"

if contract\_class == 'en\_onprem':

# do some processing

return "this is en\_onprem"

raise ValueError(f"contract class {contract\_class} not found")

A monkeypatch.setenv funkciót használhatjuk a környezeti változó beállításához.

@pytest.mark.parametrize(

"mock\_contract\_class,expect", [("en\_cloud", "this is en\_cloud"), ("en\_onprem", "this is en\_onprem")]

)

def test\_mock\_env\_var(mock\_contract\_class, expect, monkeypatch):

# more about monkeypatch

# https://docs.pytest.org/en/6.2.x/monkeypatch.html

monkeypatch.setenv("CONTRACT\_CLASS", mock\_contract\_class)

assert use\_env\_var() == expect

#### Kivétel

Vegyük például az előzőleg bemutatott *use\_env\_var* példát, amelyet a Környezeti változó kezelésének demonstrálására használtunk, de most a kivételekkel foglalkozunk.

Itt a pytest.raises-t használjuk egy kontextuskezelőként annak érdekében, hogy elfogjuk a ValueError típusú kivételt. Itt megadhatjuk:

* a kivétel pontos típusát, például ZeroDivisionError, KeyError.
* a kívánt üzenetet, amely megfelel egy bizonyos formátumnak a reguláris kifejezés szerint.

def test\_exception(monkeypatch):

monkeypatch.setenv("CONTRACT\_CLASS", "something not existed")

with pytest.raises(ValueError, match=r"contract class something not existed not found"):

use\_env\_var()

## Integration