

IT BIZTONSÁG

(VIHIAC07)

LABORATÓRIUMI GYAKORLAT

---

## Szoftverek biztonsági tesztelése

---

*Szerző:*

GAZDAG András és KOLTAI Beatrix



2025. február 22.

# Tartalomjegyzék

<b>1. Oktatási célok</b>	<b>2</b>
<b>2. Háttéranyag</b>	<b>2</b>
2.1. Típusok alkalmazása . . . . .	3
2.1.1. Jó-e vagy rossz a dinamikus típusosság? . . . . .	3
2.1.2. A típusinformáció dokumentációként is szolgál . . . . .	3
2.1.3. Az IDE számos funkciója típusokon alapul . . . . .	4
2.1.4. A hibák érthetőbbek lesznek . . . . .	4
2.1.5. Python kiegészítése típusokkal . . . . .	5
2.2. Statikus tesztelés . . . . .	5
2.2.1. Mypy . . . . .	5
2.3. Dinamikus tesztelés . . . . .	6
2.3.1. Pytest . . . . .	6
<b>3. Környezet feállítása</b>	<b>8</b>
3.1. VM használata . . . . .	8
3.2. Otthoni környezet használata . . . . .	8
3.3. Kiinduló kód beállítása . . . . .	9
<b>4. Feladatok</b>	<b>9</b>
4.1. Demo . . . . .	9
4.1.1. Mypy használata . . . . .	9
4.1.2. Pytest használata . . . . .	11
4.2. Mypy . . . . .	11
4.3. Pytest . . . . .	12
4.4. Pytest Cont'd . . . . .	12

# 1. Oktatási célok

Ezen a laboratóriumi gyakorlaton a széles körben elterjedt biztonsági tesztelési technikákat mutatjuk be. A megismert statikus és dinamikus tesztelési technikák hatékony eszközök a szoftverekben előforduló hibák azonosítására, és ezáltal az általános biztonság növelésére. A feladatok során a korábban, az input validációs mérésen megismert alkalmazás tesztelését kell elvégezni. Ennek során egy hibákat tartalmazó implementációból kell kiindulni, és az egyes módszerek segítségével folyamatosan javítani az alkalmazást. A laboratóriumi gyakorlat elvégzésével a résztvevő képessé válik a legfontosabb tesztelési technikák használatára szoftverfejlesztés során.

## 2. Háttéranyag

A szoftvertesztelés egy kiemelten fontos terület az implementáció során előforduló hibák kiszűrésére. Megfelelő tesztelések használatával jelentősen növelhető az elkészült termék minősége, és számottevően csökkenthető a hibákból adódó felhasználói problémák és egyéb anyagi veszteségek mértéke.

A szoftvertesztelésnek több célja is van:

- **Funkcionális tesztelés** során azt ellenőrizzük, hogy a szoftver a meghatározott specifikációnak megfelelően működik-e. Ebben az esetben a követelményekből szisztematikusan levezetett tesztek segítségével nagy bizonyossággal megállapíthatjuk, hogy az elkészült szoftver teljesíti az összes elvárást.
- Ezzel szemben **biztonsági tesztelés** esetén arról szeretnénk megbizonyosodni, hogy az alkalmazás a funkcionális tesztelés során nem ellenőrzött, összes többi esetben is elfogadhatóan működik. Mivel kimerítő tesztelés a végtelen lehetséges esetszám miatt nem megvalósítható, ezért sikeres biztonsági tesztelést végezni egy jóval nehezebb és több szakértelmet igénylő tevékenység. Szerencsére a statisztikák alapján a biztonsági hibák döntő többsége hasonló okokra vezethető vissza, így a leghasznosabb tesztelési módszerek elsajátításával hatékony eredmények érhetőek el ezen a területen is. A biztonsági tesztelési technikákat *statikus* vagy *dinamikus* elemzéseként csoportosíthatjuk.

A hatékony biztonsági teszteléshez nagyban hozzá tud járulni, ha típusos nyelvet használunk a fejlesztés során. Első sorban a statikus elemzők számára jelent ez előnyt, mivel a típusok elemzésével pontosabb információkkal tudják segíteni a fejlesztést. Ennek ellenére, a típusos nyelvek használata egy régire visszanyúló vita a szakmában, amiről a mai napig nincs egyértelmű álláspont.

## 2.1. Típusok alkalmazása

### 2.1.1. Jó-e vagy rossz a dinamikus típusosság?

Erre a kérdésre nincs végleges válasz<sup>1</sup>. A projektől és annak céljaitól függ, hogy egy dinamikus típusokat használó nyelv megfelelő választás-e a probléma megoldására. A Python például egy *dinamikusan* típusos nyelv, nem pedig *statikusan* típusos. Ez azt jelenti, hogy a Python-interpreter nem ismeri az objektumok típusát a tényleges kódfuttatás előtt.

- A dinamikus típusokat legtöbbször script nyelvekben használják, mint például a Ruby, JavaScript, MATLAB, stb.
- A spektrum másik oldalán a statikusan típusos, általában fordított, nem pedig interpretált nyelvek találhatók. A statikusan típusos nyelvek közé tartozik például a Fortran, a C, a C++ vagy a Java.

Mivel a Python dinamikusan típusos nyelv, a Python-értelmezőnek nem kell ismernie a kezelendő objektumok típusát az inicializálásuk előtt. Egy objektum típusa függhet például egy olyan értéktől, amelyet csak futásidőben tudunk meg, ezért a Python-értelmező csak dinamikusan következtet a típusra. A Python esetében a dinamikus típuskezelés nagy mozgásteret ad a programozónak. Azonban ennek a rugalmasságnak is van néhány hátulütője. Tipikusan mikor már elég nagy méretű a kódbázis, akkor hasznossá válik a típusok bevezetése, amely számos előnnyel járhat. Ezek közül mutatunk be párat a következő néhány példa segítségével.

### 2.1.2. A típusinformáció dokumentációként is szolgál

A típusok dinamikus jellege miatt a Python függvények paraméterei bármilyen objektumot elfogadhatnak, függetlenül annak típusától. A következő kódrészlet emiatt tökéletesen helyes:

```
1 def append_to_container(c, e):  
2     # Code of the function
```

Egy ilyen kódrészlet láttán felmerül a kérdés, hogy meg lehet-e 100%-os bizonyossággal mondani, hogy hogyan kell használni ezt a függvényt anélkül, hogy látnánk a függvény megvalósítását? Nem. Itt van két “érvényes”, de egymással összeegyeztethetetlen módja a függvény használatának:

```
1 # First possibility  
2 c = [1, 2]  
3 e = 3  
4 append_to_container(c, e)  
5     # c is modified in-place in the function and the  
6     # function does not return any value  
7  
8 # Second possibility  
9 c = [1, 2]  
10 e = 3  
11 new_c = append_to_container(c, e)
```

<sup>1</sup><https://cerfacs.fr/coop/python-typing>

```
12 # c is copied internally by the function and a modified
13 # copy is returned.
```

Az egyik esetben a függvény helyben változtatja meg a bemeneti paraméterként kapott tömböt, másik esetben viszont visszatér a módosítottal. Az `append_to_container` függvény fejlécéből nem lehet kitalálni, hogy melyik a helyes használat a kettő közül. Erre a problémára egy lehetséges megoldás az lenne, ha a függvényt dokumentációval látnánk el, például egy docstringgel:

```
1 def append_to_container(c, e):
2     """Modify c in-place to append e at the end of the container."""
3     # Code of the function
```

Ez viszont további problémát vet fel: a docstringek (és általában a dokumentációk) gyakran elavulnak, nincsenek karbantartva, vagy egyszerűen csak nincsenek megfelelően megírva és nem tartalmazzák a megfelelő információt. Erre példa a következő részlet:

```
1 # Example of a useless docstring. The docstring only repeats
2 # informations already in function or arguments names.
3 def append_to_container(c, e):
4     """Append element e to the contained c."""
5     # Code of the function
```

A docstringek alkalmazásának a legnagyobb hátránya viszont, hogy azok nem elemezhetőek automatizált eszközökkel, például integrált fejlesztőkörnyezetekkel (IDE) vagy kódelemző programokkal, kivéve, ha nagyon szigorú formázási szabályokhoz igazodnak (pl [Sphinx formátum](#), [Google formátum](#) vagy [Numpy formátum](#)).

### 2.1.3. Az IDE számos funkciója típusokon alapul

A következő funkciók működéséhez kifejezetten előnyös, ha típusokkal van ellátva a kód:

- Dokumentáció megjelenítése tooltipekben.
- Hibák vagy figyelmeztetések megjelenítése, ha az adott típus nem egyezik az elvárt típussal.
- A “Go to type definition” funkció alkalmazása egy új kódbázis felfedezésekor.

### 2.1.4. A hibák érthetőbbek lesznek

Ha előzetesen tudjuk, hogy egy változónak milyen típusúnak kell lennie, akkor futás közben, a változó értékadásakor ellenőrizhetjük, hogy a változóhoz rendelt objektum megfelelő típusú-e.

```
1 def count_bit_number(i):
2     return i.bit_length()
3
4 a = count_bit_number(3) # result is 2
5 b = count_bit_number(1.0) # error
```

A fenti kódrészlet utolsó sora hibát okoz:

```
1 AttributeError: 'float' object has no attribute 'bit_length'
```

ahol, ha a Python tisztában lett volna azzal, hogy a `count_bit_number` függvény csak egész számokat fogad el bemenetként, a hibaüzenet valami olyasmi lehetett volna, mint például

```
1 TypeError: 'count_bit_number' got a 'float', expected 'int'
```

ami már sokkal könnyebben érthető.

### 2.1.5. Python kiegészítése típusokkal

Az előző részben láttuk, hogy mikor és miért éri meg típusokat használni. Ahelyett viszont, hogy a Python típus kezelési rendszerét egy az egyben megváltoztatták volna (ami rengeteg nemkívánatos következménnyel és bonyodalommal járt volna), a Python fejlesztői a típus-megjelölések bevezetése mellett döntöttek (type annotations, típus annotációk), a [PEP 484](#) keretében.

Fontos, hogy az *annotáció* elnevezés arra utal, hogy a Python-értelmező a típusinformációkat nem használja fel. A típus annotációk a Python-értelmező számára olyanok, mint a megjegyzések, nem kerülnek feldolgozásra.

A típus annotációk használatáról jó összefoglaló olvasható a [mypy könyvtár oldalán](#) valamint a hivatalos [python dokumentációban](#).

## 2.2. Statikus tesztelés

A *statikus* elemzések nem hajtják végre a programokat, csak azok forráskód szintű utasításait vagy gépi kódját “olvassák” végig és értelmezik. Az elemzés az utasítások egy memóriabeli reprezentációján történik. Ezek a technikák jól skálázódnak és nagy kódbázist is képesek kezelni. Hátrányuk, hogy nem férnek hozzá futási idejű információkhoz, ezért gyakran adnak hamis pozitív válaszokat: olyan kódrészleteket is sérülékenynek jelölhetnek, amik a valós életben sosem futnának le vagy sosem lehetne hibát kihasználni bennük.

### 2.2.1. Mypy

A [Mypy](#) egy statikus típusellenőrző Pythonhoz, amely képes a típusinformációk alapján hibákat azonosítani. A típusellenőrzők segítenek biztosítani, hogy a változókat és függvényeket helyesen használjuk-e a kódban. Ezzel a problémák egy részét már futtatás nélkül, statikus elemzők segítségével is ki lehet szűrni. A következő példa bemutat egy lehetséges hibát, ami a Mypy segítségével könnyen megtalálható:

```
1 number = input("What is your favourite number?")
2 # error: Unsupported operand types for + ("str" and "int")
3 print("It is", number + 1)
```

A típusjavaslatok hozzáadása nem zavarja a program futtatását. Ezekre az információkra úgy érdemes gondolni, mint a megjegyzésekre. A Python-értelmező mindig képes az alkalmazást futtatni, még akkor is, ha a Mypy hibát jelez, bár ilyen esetben valószínűleg egy futásidejű probléma fog jelentkezni.

A Mypy-t úgy tervezték, hogy fokozatosan lehessen hozzáadni a kódhoz típusinformációkat, nem szükséges a kód egészét egyben annotálni. Emiatt a gyakorlatban könnyen hasznosítható. Támogat olyan funkciókat, mint a típuskövetkeztetés (type inference), a generikusok (generics), a hívható típusok (callable types), a tuple típusok (tuple types), a union típusok (union types), a structural subtyping, és még sok más. A Mypy használatával a programok könnyebben érthetővé és karbantarthatóvá válnak.

## 2.3. Dinamikus tesztelés

A *dinamikus* tesztek végrehajtás közben elemzik a szoftvert. Emiatt hozzáférnek a futásidejű információkhoz, így jóval pontosabb elemzést tudunk velük végezni. A dinamikus tesztek jellegéből fakadóan egy-két dologra figyelni kell:

- nehezen skálázódnak, mivel a teszteket gyakran manuálisan kell megírni.
- ha nem fedjük le a bemenetekkel az összes lehetséges lefutást, a nem futtatott kódrészletnek a biztonságáról nem tudunk semmit megállapítani.
- a statikus elemzéshez hasonlóan, ez is adhat hamis negatív eredményt, és nem garantálható, hogy az összes sérülékenységet meg lehet vele találni.

### 2.3.1. Pytest

A Pytest egy dinamikus tesztelésre készített Python alapú tesztelési keretrendszer, amely teszt kódok írására és végrehajtására szolgál. A Pytest előnyei a következők:

- A Pytest ingyenes és nyílt forráskódú.
- Egyszerű a szintaxisa, emiatt a Pytestet nagyon könnyű elkezdni használni.
- A Pytest több tesztet is képes párhuzamosan futtatni, ami csökkenti a tesztcsomag végrehajtási idejét.
- A Pytest automatikusan megtalálja a `test_*.py` kezdetű vagy `*_test.py` végződésű fájlokat.
- A Pytest lehetővé teszi, hogy a teljes tesztcsomagnak csak egy részhalmazát futtassuk vagy kihagyjuk a végrehajtás során.

A tesztek elkészítése során a fájlneveknek “test”-el kell kezdődniük vagy végződniük, mint például `test_pelda.py` vagy `pelda_test.py`, ahhoz, hogy a Pytest automatikusan megtalálja őket. Ha a tesztek egy osztály metódusaiként vannak definiálva, az osztály nevének “Test”-tel kell kezdődnie (például: `TestOsztaly`). Ilyen esetben, az osztálynak nem lehet `__init__` metódusa. Az osztályon belül a függvények neveinek szintén “test\_”-tel kell kezdődniük. Az olyan metódusok, amelyek neve nem felel meg ennek a mintának, nem lesznek tesztként végrehajtva.

```
pytest
===== test session starts =====
platform darwin -- Python 3.11.2, pytest-7.3.1, pluggy-1.0.0
rootdir: /Users/stanmd/Projects/MD/CCI/testing-flask-pytest
collected 1 item

test_api.py . [100%]

===== 1 passed in 0.07s =====
(venv) stanmd@silicon_savannah ~/Projects/MD/CCI/testing-flask-pytest git:(main) x
|
x ..-flask-pytest (-zsh) %1 +
```

1. ábra. Pytest futtatása

```
1 # Tesztelendo fuggveny a muveletek.py fajlban
2 def osszeg(szam1, szam2):
3     """Visszaadja ket szam osszeget"""
4     return szam1 + szam2
```

```
1 # Teszteset definicioja a test_peldak.py fajlban
2 import pytest
3
4 def test_osszeg():
5     assert osszeg(1, 2) == 3
```

Ezután a `pytest` paranccsal tudjuk lefuttatni a teszteket, vagy az egyes teszteket külön, például a `pytest test_peldak.py` paranccsal.

Ennek hatására a `pytest` automatikusan megtalálja a teszteket a mappában, vagy a megadott fájlban belül. Megkeresi a `test` kezdetű fájlokat és az azokban található `test` kezdetű függvényeket. A fenti képen<sup>2</sup> is látszik, hogy a Pytest a sikeres tesztet zöld ponttal jelöli, a sikertelen tesztet piros F-vel. Ezen felül jelzi, hogy hány teszt ment át vagy volt sikertelen.

```
1 import pytest
2 @pytest.fixture
3 def client():
4     """Configures the app for testing
5
6     Sets app config variable ``TESTING`` to ``True``
7
8     :return: App for testing
9     """
10
11     #app.config['TESTING'] = True
12     client = app.test_client()
13
14     yield client
```

<sup>2</sup><https://circleci.com/blog/testing-flask-framework-with-pytest/#c-consent-modal>



A fenti példában látható `@pytest.fixture` annotáció megmondja a pytestnek, hogy a következő függvény létrehoz (a `yield` parancs segítségével) egy tesztelésre szánt alkalmazást. Ebben az esetben a függvény nem csinál túl sok mindent, de akár ideiglenes adatbázisfájlokat is konfigurálhat, vagy beállíthat konfigurációkat a teszteléshez (pl. a kikommentelt `app.config` sort)<sup>3</sup>.

## 3. Környezet feállítása

### 3.1. VM használata

A VirtualBox alkalmazást használva importáljuk a kiadott OVA fájlt, majd indítsuk el a VM-et. A kiadott VM-et használva semmilyen további előkészület nem szükséges.

### 3.2. Otthoni környezet használata

Ha nem a kiadott VM-et használjuk, az alábbi lépéseken lehet végigmenni, hogy a saját környezetet előkészítsük.

1. Ellenőrizzük, hogy a Python 3.7 vagy újabb verzió telepítve van-e: `python3 --version`  
Ha nincs, a kiadott python tutorial alapján telepítsük.
2. Szükséges Python csomagok telepítése
  - **tkinter**: Ez általában előre telepítve van a Pythonhoz. Ha nem:
    - **macOS**: a Tcl/Tk könyvtárak telepítése a Homebrew segítségével: `brew install python-tk`
    - **Ubuntu**: Telepítsük a következővel: `sudo apt install python3-tk` vagy `sudo apt-get install python3-tk`
  - **Pillow**: Képfeldolgozáshoz szükséges, ezt majd a virtuális környezeten belül telepítjük.
  - **mypy**: Statikus teszteléshez használt csomag, ezt majd a virtuális környezeten belül telepítjük.
  - **pytest**: Dinamikus teszteléshez használt csomag, ezt majd a virtuális környezeten belül telepítjük.

```
1 brew info python-tk
2 ==> python-tk@3.13: stable 3.13.1
3
4 brew info tcl-tk
5 ==> tcl-tk: stable 9.0.1
```

### 3. Virtuális környezet beállítása

- Hozzunk létre egy virtuális környezetet a függőségek kezeléséhez:

---

<sup>3</sup>[https://codethechange.stanford.edu/guides/guide\\_flask\\_unit\\_testing](https://codethechange.stanford.edu/guides/guide_flask_unit_testing)

```
1 python3 -m venv myenv # létrehozás
2 source myenv/bin/activate # aktiválás
```

- Miután aktiváltuk, a terminál promptban megjelenik a (myenv) felirat, ami jelzi, hogy most már azon belül adod ki a parancsokat. Ha myenv helyett más neved adunk, természetesen az jelenik meg és azon belül kell keresni a /bin mappát az aktiválásnál.

#### 4. Python csomagok telepítése

- Telepítsük a `Pillow` képfeldolgozáshoz szükséges python csomagot a virtuális környezetben: `pip3 install Pillow`
- Telepítsük a `mypy` statikus teszteléshez szükséges python csomagot a virtuális környezetben: `pip3 install mypy`
- Telepítsük a `pytest` dinamikus teszteléshez szükséges python csomagot a virtuális környezetben: `pip3 install pytest`

### 3.3. Kiinduló kód beállítása

Töltsük le a kiindulási kódot az oldalról: <https://software.crysys.hu/it-security/security-testing>

## 4. Feladatok

A laborgyakorlat négy feladatból áll. Melyek során egy statikus és egy dinamikus tesztelési technika bemutatása és kipróbálása a cél.

### 4.1. Demo

A példa alkalmazáson keresztül demonstráljuk a `mypy` valamint a `pytest` használatát.

Nyissuk meg a `demo.py` fájlt. Ebben van egy leegyszerűsített bankszámla osztály, amire lehet pénzt tenni, lehet róla pénzt levenni, valamint ellenőrizni, hogy nem negatív-e az egyenleg. (Nem a működés helyessége a fontos nekünk most, nem foglalkozunk olyannal, hogy pénzlevételkor negatívba mehet az egyenleg, stb.)

A fájl végén látható 3 sor demonstrálja is az osztály használatát.

#### 4.1.1. Mypy használata

- Futtassuk le a `mypy`-t, hogy lássuk, hogy típus információk nélkül nem mond semmi hasznosat:

```
1 mypy --config-file mypy.ini demo.py
```

- Adjunk a fájl végére egy nyilvánvalóan rossz függvényhívást, hogy bemutassuk, hogy ez nem derül ki a kódból magából, csak futásidőben lenne belőle hiba. A rossz függvényhívás legyen:

```
1 account.deposit("test")
```

- Futtassuk le a `mypy`-t, hogy lássuk, hogy az sem találja meg így ezt a problémát:

```
1 mypy --config-file mypy.ini demo.py
```

- Kommentezzük ki ezt a hozzáadott sort, hogy ne zavarjon minket egyelőre a további munkában!
- Lássuk el az osztályt típus információkkal, hogy javítsunk a helyzeten. Ezt magunktól, illetve a használatot bemutató példából lehet jól összerakni. Ne felejtsük el importálni a `Tuple` osztályt a megfelelő működéshez! Egy helyes megoldás az annotálásra a következő:

```
1 from typing import Tuple
2
3 class BankAccount:
4
5     def __init__(self,
6                   owner: str,
7                   account_number: Tuple[int, int, int],
8                   initial_balance: int = 0) -> None:
9         self.owner = owner
10        self.account_number = account_number
11        self.balance = initial_balance
12
13    def deposit(self, amount: int) -> None:
14        self.balance += amount
15
16    def withdraw(self, amount: int) -> None:
17        self.balance -= amount
18
19    def is_overdrawn(self) -> bool:
20        return self.balance < 0
21
22    def __str__(self) -> str:
23        return f"Balance of {self.owner} (account number: {self.
        account_number}) is: {self.balance}"
```

- Futtassuk le a `mypy`-t, hogy lássuk, hogy ha helyesen adtunk meg mindent, akkor hiba nélkül lefut az ellenőrzés:

```
1 mypy --config-file mypy.ini demo.py
```

- Az elején hozzáadott hibás függvényhívást tegyük most vissza a kódba!
- Futtassuk le a `mypy`-t ismét, hogy lássuk, hogy ha helyesen adtunk meg mindent, akkor a `mypy` képes megtalálni és jelezni egy ilyen hibát:

```
1 mypy --config-file mypy.ini demo.py
```

- Töröljük a hibás sort a kódból most, hogy az ne zavarjon a folytatásban!

### 4.1.2. Pytest használata

Nyissuk meg a `test_demo.py` fájlt:

- Eddig statikus elemzést csináltunk, mostantól viszont dinamikus tesztelés jön, vagyis most kezdjük futtatni a kódot! (Eddig csak a mypy-t futtattuk, ami csak végigolvasta a kódunkat.)
- Importálva van a másik fájlból az osztály, amit eddig néztünk.
- 4 tesztet van a fájlban, ami a bankszámlához kapcsolódó egyes funkciók tesztelését végzi.
- Az első tesztet a `test_deposit` függvény:
  - Az első sorban egy logolás van, hogy tudjuk, hogy melyik esetet teszteljük épp.
  - Aztán létrehozunk egy példa bankszámlát Dagobert bácsinak 10-es egyenleggel.
  - Elhelyezünk plusz 5-öt a bankszámlán.
  - Az érthetőség kedvéért kiíratjuk a számla aktuális helyzetét (ez a sor a háttérben a `__str__` függvényt hívja meg).
  - Végzünk egy ellenőrzést, hogy a művelet után a számlaegyenleg annyi-e, mint amennyit várunk.
- A többi teszt is ugyan erre a logikára van felépítve, csak a következőben leemelünk pénzt a számláról, aztán pedig azt ellenőrizzük, hogy a számlán található összeg pozitív vagy negatív-e egy-egy művelet után.
- Futtassuk le a teszteket, és nézzük meg, hogy azokból mi és hogy látszik a konzolban:

```
1 pytest test_demo.py
```

## 4.2. Mypy

A feladat során a cél, hogy az előző mérésből megismert `CIFF` osztályt lássa el olyan típusinformációkkal, amik segítik a fejlesztést. Ehhez hajtsa végre a következő lépéseket:

- A `ciff.py` fájlban az `__init__` függvény paramétereit és változóit lássa el típus információkkal (a setter és getter függvényeket is)!
- A projekt mappából futtassa a mypy alkalmazást a következő paranccsal:

```
1 mypy --config-file mypy.ini src/ciff.py
```

- Ha a mypy jelez valamilyen hibát, akkor javítson a kódon, amíg hiba nélkül le nem fut az ellenőrzés!

**Moodle kérdés:** A mypy által jelzett hiba kijavítása után mi a helyes sor?

### 4.3. Pytest

A feladat során a `pytest`-et alkalmazva dinamikus tesztelést kell végrehajtani:

- Első lépésként futtassa le a teszteket, és a kimenet alapján értékelje, hogy a tesztek helyes vagy helytelen eredményt adnak. A tesztek futtatása így lehetséges:

```
1 pytest test_ciff.py
```

- Mint láthatja, egy teszt hibás eredményt adott: az `invalid5.ciff` fájl feldolgozása során az alkalmazás helyesnek tekintette a fájlt, pedig az valójában nem az. Javítsa ki a `ciff.py` fájlban a hibát: a hibaüzenet alapján implementálja a hiányzó ellenőrzést!
- Futtassa újra a teszteket, és ellenőrizze, hogy most már minden teszt helyes eredményt ad-e!

**Moodle kérdés:** Melyik három tagváltozót kellett ellenőrizni, ami megoldotta a hibás tesztet? (Csak a változók nevét adja meg, vesszővel elválasztva! pl.: a, b, c)

### 4.4. Pytest Cont'd

Az előző feladat folytatásaként:

- Vizsgálja meg a tesztesetek futása során keletkezett log üzeneteket, hogy biztos minden eset megfelelően működik-e! Ha talál olyan esetet, ami bár helyes eredménnyel tér vissza, de mégsem hibátlanul fut le, akkor keresse meg szintén a `ciff.py` fájlban a problémát, és implementáljon további ellenőrzéseket.
- Az összes probléma kijavítása után futtassa ismét a `pytest` parancsot a korábbiakhoz hasonló módon, és ellenőrizze, hogy most már minden teszt megfelelően működik!

**Moodle kérdés:** Mi annak a `ciff` fájlnak a neve, ami bár helyes teszt eredményt ad, mégis helytelenül fut le?