

Szoftvertesztelés Pythonban  
Számológép projekt  
Szabó Huba



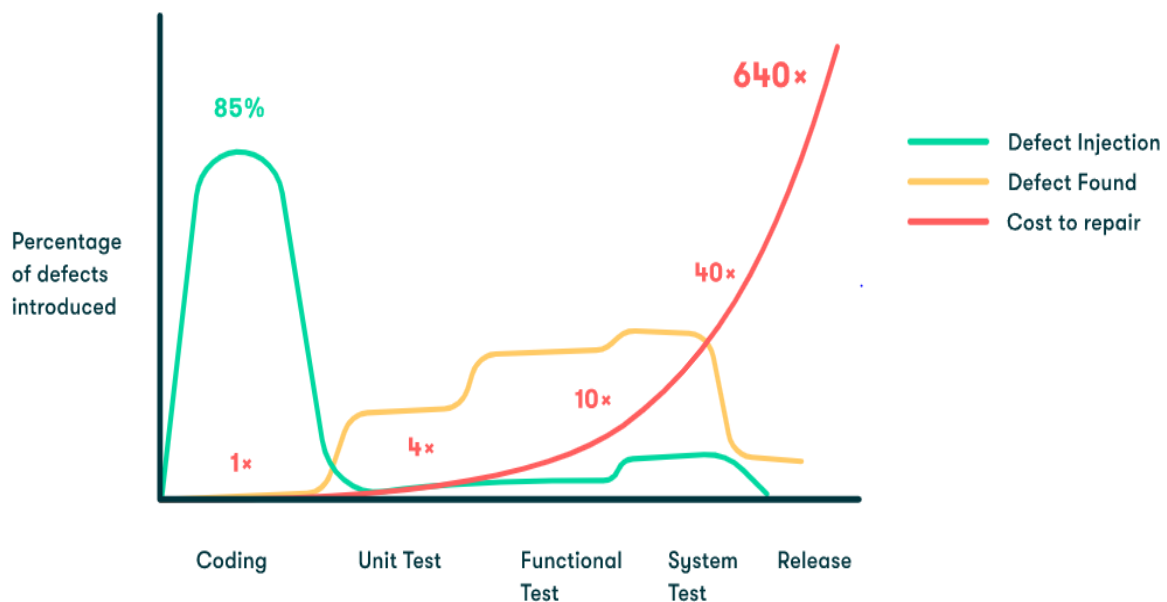
## Bevezetés:

Napjainkban a világ fejlődése következtében a digitális eszközök világa is elért egy magas szintet. Ezen gyors fejlődés miatt a megrendelők egyre igényesebbek lettek mely következtében azok a termékek(mi esetünkben szoftverek) amelyeket megvásárolnak minél jobb, magasabb elvárásoknak kell megfelelnie. A jó működés érdekében bevezették a szoftverek tesztelését.

A tesztelések bevezetésére azért volt szükség, mivel egy megírt szoftver sosem biztonságos, rengeteg hiba található benne. A hibák emberi szemmel nem mindig felismerhetők, viszont a tesztelés segítségével több hibát lehet kiszűrni.

Azt tudni kell, hogy egyetlen szoftver sem lesz tökéletes. Attól mert a fejlesztők, teszterek véghez visznek tesztek nem feltétlen lesz biztonságos a szoftver, mert egyszerűen annyi esetet kellene letesztelni, hogy effektív idő és pénz sincs rá. Viszont ha egyáltalán nem használják a teszteléseket, és kikerül az emberiség kezébe, akkor a szoftver valószínűleg nem megfelelő működést fog mutatni, ami által az emberek nem fogják használni.

A szoftvertesztelés nagyon fontos egy termék gyártásában, hiszen a hibák kiküszöbölésében segít. Ez azért fontos mert ezek időben való felfedezésével, csökkenthetjük azok kijavításának költségeit.



1.abra

## ***Mi is a szoftvertesztelés?***

A **szoftvertesztelés** egy rendszer vagy program kontrollált körülmények melletti futtatása, és az eredmények kiértékelése. A hagyományos megközelítés szerint a tesztelés célja az, hogy a fejlesztés során létrejövő hibákat minél korábban felfedezze. A tesztelői munka egyre inkább eltolódik a fejlesztők és a döntéshozók információkkal való támogatásának irányába.

## ***Szoftvertesztelés definíciója röviden?***

- Felhasználhatósági szemszög: "Alkalmasság a felhasználó által szánt célra."
- Tesztelői szemszög: "Egyezés a specifikációval."

## ***Szoftvertesztelés célja?***

A klasszikus szoftvertesztelés célja a **szoftverhibák felfedezése**. A fejlesztésnek minél korábbi szakaszában derül fény egy hibára, annál olcsóbb annak korrigálása. Újabb keletű elvárás a **szoftverminőség mérése**.

## ***Nincs 100%-ban hibátlan szoftver!!***

## ***Tesztelési módszerek, technikák***

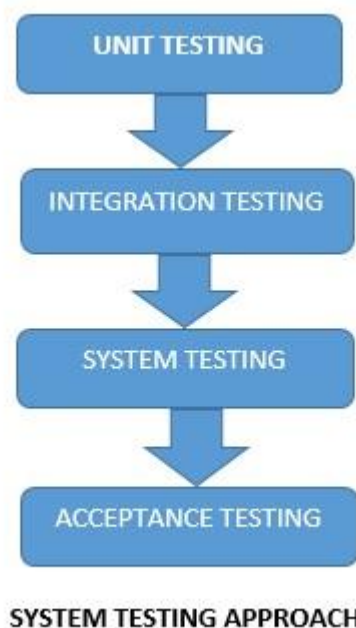
A tesztelési módszereket csoportosítani tudjuk több féle szinten, annak érdekében, hogy milyen szinten használjuk ezeket.

A megfelelő minőséget két szempont biztosítja:

- A **validációs tesztelés** (validation) a felhasználhatóságot vizsgálja: alkalmas-e a rendszer a felhasználó által szánt célokra. A megfelelő terméket készítjük el?
- A **hiányosság tesztelés** (verification) a tesztelői szemszöget képviseli és azt vizsgálja, hogy minden a specifikáció szerint működik-e. Ennek a tesztelésnek az a célja, hogy felfedezzük azokat a hibákat, amelyek a szoftver helytelen viselkedését okozzák, azaz a működése nem felel meg a specifikációjának. A terméket jól készítjük el?

## **Tesztelési szintek:**

- **Komponensteszt:**
  - Ezen a szinten a rendszer egy komponensét teszteljük önmagában.
  - Az egységteszt vagy unit-teszt a metódusokat teszteli. Megvizsgálja, ha adott paraméterekre az elvárt eredményeket kapjuk. Fontos, hogy magának az egységtesztnek ne legyen mellékhatása.
- **Integrációs teszt:**
  - A komponensek együttműködésének tesztjeit foglalja magába, az összeillesztése során keletkező hibákat keresi.
- **Rendszerteszt:**
  - A rendszerteszt a már egységes egészként, szoftverterméket (összes komponens) teszteli.
  - Megvizsgálja, ha a rendszer megfelel a rendszertervnek és a követelmény, illetve funkcionális specifikációnak.
- **Felhasználói elfogadási tesztelés:**
  - A felhasználói (alfa, béta, felhasználói átvétel, üzemeltetői átvétel) tesztek során a felhasználók a kész rendszert tesztelik, meggyőződve, hogy a termék megfelel az elvárásoknak, hibamentesen és biztonságosan használható éles körülmények között is.



2. ábra

## **Tesztelési technikák:**

- **Fekete doboz tesztelés:**
  - A szoftver teszteli a rendszer felépítésének és a belső modulok szerkezetének ismerete nélkül.
  - Célja a hiányzó funkcionálisok, teljesítmény és inicializálási problémák megtalálása.
- **Szürke doboz tesztelés:**
  - Átmenet a fekete illetve a fehér doboz tesztelés között, a kettő kombinációja
  - Az alkalmazás belső felépítését csak részben ismerjük.
- **Fehér doboz tesztelés:**
  - A szoftver belső felépítését teszteli.
  - A fekete dobozos tesztelés ellentétje.



3. ábra

## **Szoftver projekt bemutatása:**

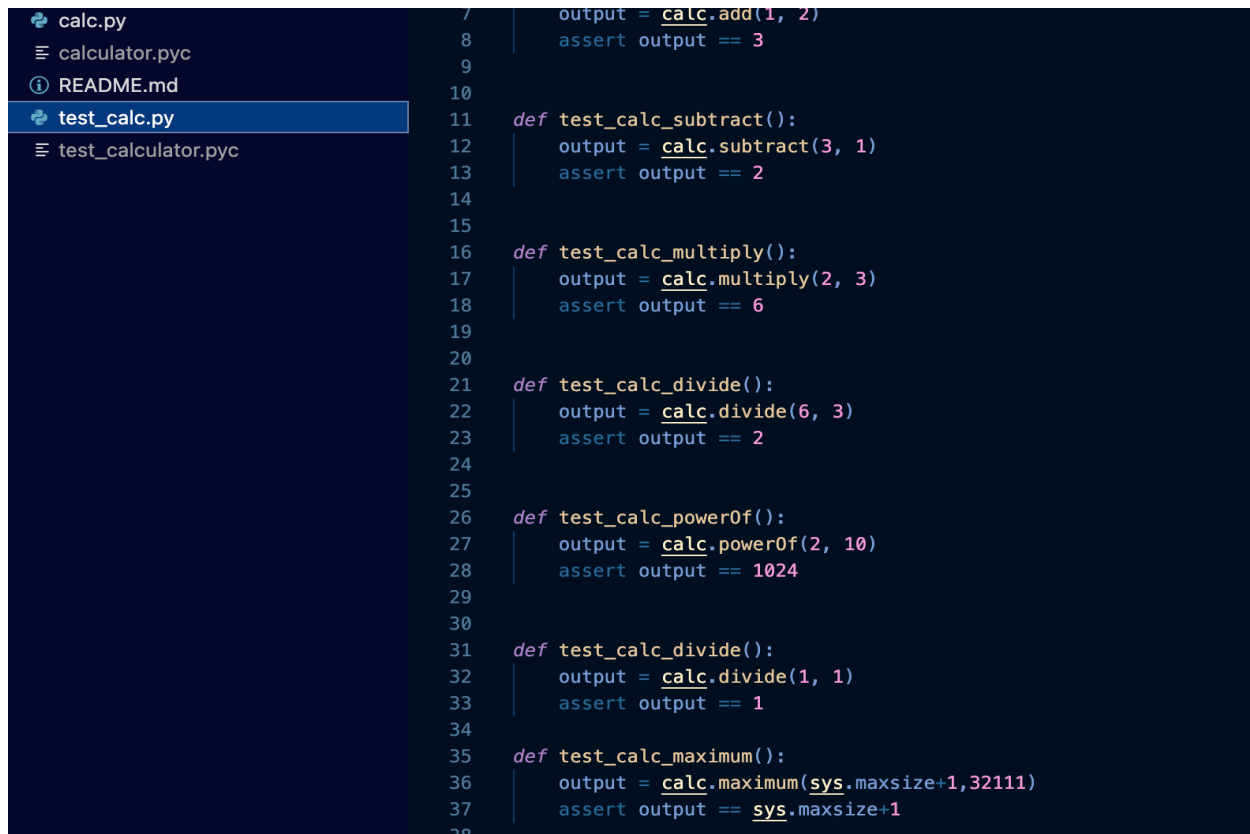
A dokumentáció további részében az általam előkészített kicsi python szoftver projekt pár metódusának a tesztelését szeretném bemutatni.

A projekt megvalósítására Visual Studio Code fejlesztői környezetet használtam. E fejlesztői környezetben a szoftvertesztelés alapjait sajátítottam el, unit tesztek írtam egy számológép függvénytípusára. (4. ábra/5. ábra)

```
▼ SZOFTVERTESZTELES
  > __pycache__
  > .pytest_cache
  > pytest-env
  ◆ .gitignore
  📄 calc.py
  📄 calculator.pyc
  ⓘ README.md
  📄 test_calc.py

1  import math
2  from turtle import clear
3
4  # Function to add two numbers
5  def add(num1, num2):
6      return num1 + num2
7
8  # Function to subtract two numbers
9  def subtract(num1, num2):
10     return num1 - num2
11
```

4. ábra

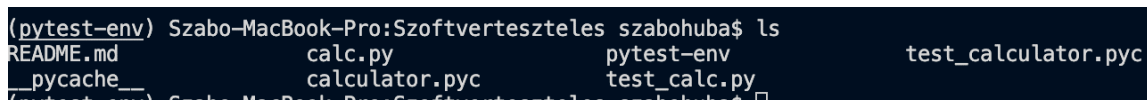


```
7 | output = calc.add(1, 2)
8 | assert output == 3
9 |
10 |
11 | def test_calc_subtract():
12 |     output = calc.subtract(3, 1)
13 |     assert output == 2
14 |
15 |
16 | def test_calc_multiply():
17 |     output = calc.multiply(2, 3)
18 |     assert output == 6
19 |
20 |
21 | def test_calc_divide():
22 |     output = calc.divide(6, 3)
23 |     assert output == 2
24 |
25 |
26 | def test_calc_powerOf():
27 |     output = calc.powerOf(2, 10)
28 |     assert output == 1024
29 |
30 |
31 | def test_calc_divide():
32 |     output = calc.divide(1, 1)
33 |     assert output == 1
34 |
35 | def test_calc_maximum():
36 |     output = calc.maximum(sys.maxsize+1, 32111)
37 |     assert output == sys.maxsize+1
38 |
```

5. ábra

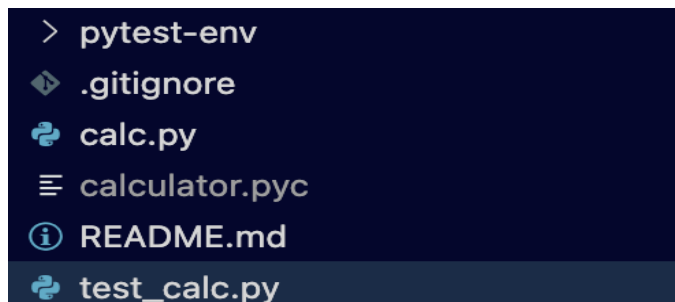
A tesztelés egyszerűsége, valamint a jó átláthatósága érdekében Python programozási nyelvet választottam az implementációhoz, ennek pedig a legújabb verzióját(3.7.4). A tesztek megvalósítására pedig az ajánlott pytest(7.1.1)-re esett a választásom.

A rendezettség, illetve a biztonság kedvéért egy virtuális környezetben valósítottam meg e kis projektet.(6,7. ábrák)



```
(pytest-env) Szabo-MacBook-Pro:Softvertesztes szabohuba$ ls
README.md          calc.py             pytest-env          test_calculator.pyc
__pycache__        calculator.pyc      test_calc.py
```

6. ábra



7. ábra

A másik ok amiért a virtuális környezetet vákaszítottam az az, hogy a Python nem a legjobb a függőségek kezelésében. Ha nem vagyunk specifikusak akkor a pip az összes külső csomagot amit instalálni akarunk a site-packages mappába instalálja a fő Python mappába.

- `$ mkdir pytest_demo`
- `$ cd pytest_demo`
- `$ python3 -m venv pytest-env`

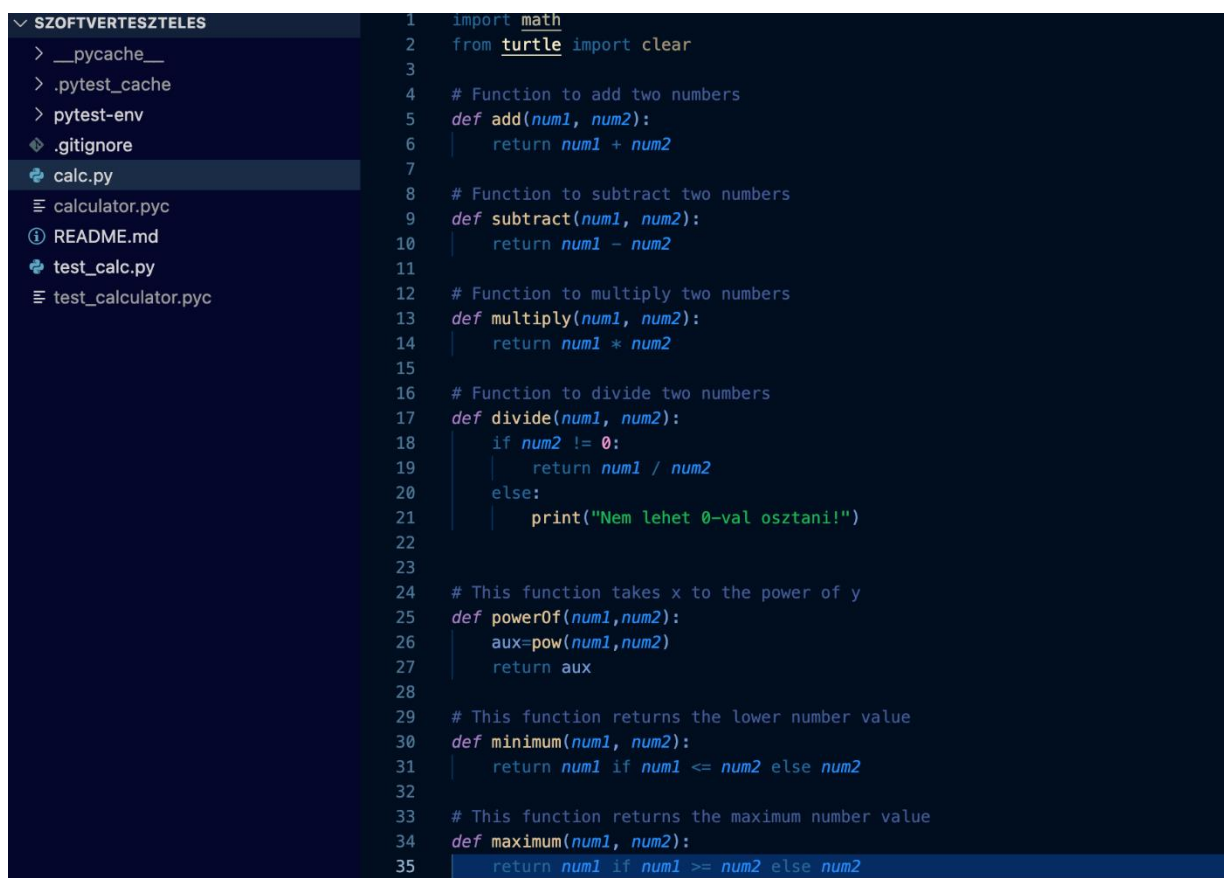
### Virtuális környezet aktiálása és használata:

- `$ source pytest-env/bin/activate`

Második lépésben instaláltam a pytestet a projektben lévő függvények tesztelésére. A pytest az egy tesztelési framework ami a python specifikus, az az python könyvtárcsomagra alapszik.(megjegyzés-a pip csomag menedzser 22.0.4 es verzióját használtam.)

- `pip3 install pytest`

Miután minden kellék telepítve lett, illetve létrehoztuk a szükséges python fájlokat amiket tesztelni akarunk, nekiállhattunk a tesztfájlok létrehozásának.



```
1 import math
2 from turtle import clear
3
4 # Function to add two numbers
5 def add(num1, num2):
6     return num1 + num2
7
8 # Function to subtract two numbers
9 def subtract(num1, num2):
10    return num1 - num2
11
12 # Function to multiply two numbers
13 def multiply(num1, num2):
14    return num1 * num2
15
16 # Function to divide two numbers
17 def divide(num1, num2):
18     if num2 != 0:
19         return num1 / num2
20     else:
21         print("Nem lehet 0-val osztani!")
22
23
24 # This function takes x to the power of y
25 def powerOf(num1, num2):
26     aux=pow(num1,num2)
27     return aux
28
29 # This function returns the lower number value
30 def minimum(num1, num2):
31     return num1 if num1 <= num2 else num2
32
33 # This function returns the maximum number value
34 def maximum(num1, num2):
35     return num1 if num1 >= num2 else num2
```

8. ábra

A calc.py python fájl egy számológép műveleteit tartalmazza:

- add(összead)
- subtract(kívon)
- multiply(szoroz)
- divide(osztás)
- powerOf(hatványozás)
- minimum, maximum(két szám közül a kisebb, illetve nagyobb)

A fentebb felsorolt függvényeket szeretnénk volna tesztelni pytest segítségével. Ennek megfelelően létre kellett hoznunk egy python fájlt aminek a neve a „calc.py” – t kellett tartalmaznia, kiegészítve a „test” – kulcsszóval. Erre azért volt szükség, mivel a környezetünk így automatikusan felismeri, hogy ebben a fájlban a calc.py függvényeit teszteljük.

Miután létrehoztam a test\_calc.py fájlt különböző tesztek írtam minden függvényre amit a számológép fájlba implementáltam. (9. ábra)

```
def test_calc_multiply():
    output = calc.multiply(2, 3)
    assert output == 6

def test_calc_divide():
    output = calc.divide(6, 3)
    assert output == 2
```

9. ábra

A továbbiakban a „test\_calc.py” nevű mappába beimportáljuk az eredeti „calc.py”-t, hogy elérjük a tesztelésre szánt függvényeinket. Ezek után a 9. ábrán láthatjuk a tesztelésre megírt függvények szerkezetét. A fentiekben(9. ábra) két tesztmetódust láthatunk implementálva(szorozás, osztás). Ezeknek a szerkezete nagyon fontos.

A metódus nevéből ki kell derülnie (a mi esetünkben), hogy melyik műveletet fogjuk tesztelni, valamint, hogy teszteljük-e. A következőkben egy változóban elmentjük a kívánt függvény visszatérítési értékét.(10. ábra)

```
output = calc.multiply(2, 3)
```

10. ábra



A továbbiakban a legfontosabb sora jön a megírt tesztfüggvényünknek.(11. ábra)

```
assert output == 6
```

11. ábra

Ebben a sorban történik meg a kiértékelése az eredményünknek. Az outputban találhatjuk az általunk kívánt számítás eredményét, amit egyenlővé teszünk az elvárt kimenettel, a fenti képen (11. ábra) ez a „6”. Visual Studio Code-ban „pytest -v” paranccsal(12. ábra) tudjuk lefuttatni a tesztelést. Ha a 11. ábrán látható egyenlőség teljesül akkor a 12. ábrán látható eredmény kapjuk a terminálra.

```
pytest-env) Szabo-MacBook-Pro:Szoftvertesztes szabohuba$ pytest -v
===== test session starts =====
platform darwin -- Python 3.7.4, pytest-7.1.1, pluggy-1.0.0 -- /Users/szabohuba/Documents/GitHub/Szoftvertesztes/pytest-env/bin/python3
cachedir: .pytest_cache
rootdir: /Users/szabohuba/Documents/GitHub/Szoftvertesztes
collected 7 items

est_calc.py::test_calc_addition PASSED [ 14%]
est_calc.py::test_calc_subtract PASSED [ 28%]
est_calc.py::test_calc_multiply PASSED [ 42%]
est_calc.py::test_calc_divide PASSED [ 57%]
est_calc.py::test_calc_powerOf PASSED [ 71%]
est_calc.py::test_calc_maximum PASSED [ 85%]
est_calc.py::test_calc_minimum PASSED [100%]

===== 7 passed in 0.22s =====
pytest-env) Szabo-MacBook-Pro:Szoftvertesztes szabohuba$
```

12. ábra

```
pytest-env) Szabo-MacBook-Pro:Szoftvertesztes szabohuba$ pytest -v
===== test session starts =====
platform darwin -- Python 3.7.4, pytest-7.1.1, pluggy-1.0.0 -- /Users/szabohuba/Documents/GitHub/Szoftvertesztes/pytest-env/bin/python3
cachedir: .pytest_cache
rootdir: /Users/szabohuba/Documents/GitHub/Szoftvertesztes
collected 7 items

est_calc.py::test_calc_addition PASSED [ 14%]
est_calc.py::test_calc_subtract PASSED [ 28%]
est_calc.py::test_calc_multiply PASSED [ 42%]
est_calc.py::test_calc_divide PASSED [ 57%]
est_calc.py::test_calc_powerOf PASSED [ 71%]
est_calc.py::test_calc_maximum PASSED [ 85%]
est_calc.py::test_calc_minimum PASSED [100%]

===== 7 passed in 0.22s =====
pytest-env) Szabo-MacBook-Pro:Szoftvertesztes szabohuba$
```

13. ábra

A 12,13 as ábrákon egy sikeresen lefutott, hibaüzenetek nélküli terminálra küldött eredményt kaptunk. Látható, hogy melyek azok a tesztek amelyek helyesek, valamint az is, hogy mennyi idő alatt futottak le ezek. A jobb oldalon a százalékok pedig felosszák a 100 % feladatot, kisebb részekre, és minden teszt után kiíródik, hogy a teljes munkának hány százaléka az a függvény.

Ha esetleg valamelyik tesztünk nem teljesül akkor hibaüzenetet kapunk a terminálunkra. Futtatás után a függvény neve mellé odakerül, hogy „FAILED”, valamint lentebb a hibaüzenetet láthatjuk, hogy melyik sorban van. (14. ábra)

```

(pytest-env) Szabo-MacBook-Pro:Szoftverteszteszes szabohuba$ pytest -v
===== test session starts =====
platform darwin -- Python 3.7.4, pytest-7.1.1, pluggy-1.0.0 -- /Users/szabohuba/Documents/GitHub/Szoft
cachedir: .pytest_cache
rootdir: /Users/szabohuba/Documents/GitHub/Szoftverteszteszes
collected 7 items

test_calc.py::test_calc_addition PASSED
test_calc.py::test_calc_subtract PASSED
test_calc.py::test_calc_multiply PASSED
test_calc.py::test_calc_divide FAILED
test_calc.py::test_calc_powerOf PASSED
test_calc.py::test_calc_maximum PASSED
test_calc.py::test_calc_minimum PASSED

===== FAILURES =====
test_calc_divide
def test_calc_divide():
    output = calc.divide(1, 0)
    assert output == 1
>
E      AssertionError: assert None == 1

test_calc.py:33: AssertionError
----- Captured stdout call -----
Nem lehet 0-val osztani!

```

14. ábra

A 14. ábrán megfigyelhetjük, hogy a beépített hibaüzeneten kívül az is megjelenik ha valami le van kezelve a tesztelt függvényben „0- val nem lehet osztani” (ez az általam írt lekezelés).

A

## Könyvészet:

- <https://docs.google.com/document/d/1FUUzWrYopCJinb09hKW8EGxBSl7KeYKaA3f8zN8kQDY/edit#>
- <https://docs.pytest.org/en/6.2.x/>
- <https://realpython.com/tutorials/testing/>