# איך לכתוב בדיקות למערכות תוכנה בפייתון

כל מערכת-תוכנה רצינית כוללת אוסף מקיף של בדיקות אוטומטיות. בתהליך הפיתוח, בכל פעם שמבצעים שינוי במערכת, גם אם זה שינוי קטן, מריצים את כל הבדיקות מחדש, כדי לוודא שלא קלקלנו שום דבר. בשיעור זה נראה שלוש דרכים לכתוב בדיקות אוטומטיות בפייתון.

1. doctest – בדיקות אוטומטיות בתוך התיעוד של כל פונקציה. משמשות בעיקר להדגמת אופן השימוש בפונקציה.
2. unittest – ספריה תקנית של פייתון לכתיבת בדיקות אוטומטיות מקיפות, בתוך מחלקה ייעודית. התחביר דומה ל JUnit שלמדתם (כנראה) ב-Java.
3. pytest – ספרייה חיצונית, המאפשרת גם להריץ באופן אוטומטי את שני הסוגים הקודמים, וגם לכתוב בדיקות בתחביר פשוט וקריא יותר. זו השיטה הנפוצה ביותר כיום לכתיבת בדיקות.

## *1.* doctest

המודול doctest הוא מודול תקני של פייתון. כשמריצים אותו, הוא מחפש חלקים של טקסט שנראים כמו קוד פעיל של פייתון, ומבצע את אותם חלקים כדי לוודא שהם עובדים כפי שהם מציגים. זה שימושי במיוחד כדי לבחון אם הקוד שביצענו באמת מבצע את מה שהיה מוטל עליו לעשות, וגם מאפשר לנו להציג למתכנת אחר את צורת השימוש במחלקה.   
אז איך מתמשים ב-doctest?

כשאנחנו במצב האינטראקטיבי של פייתון כל פקודה מתחילה ב-'<<<' ולאחריה הפקודה עצמה ואז אמור להתקבל איזשהו ערך בהתאם. ה-doctest עובד בצורה דומה, אנחנו כותבים את הקוד *כאילו* הוא נכתב במצב האינטראקטיבי של פייתון בתוך מחרוזת התיעוד (docstring), ואז ב-main אנחנו מפעילים את ה-doctest דרך module שנקרא באותו השם ומפעילים את הפונקציה testmode() של המודול. הפונקציה לוקחת כל מקום במחרוזת של הdocstring ומפעילה אותו כאילו הוא היה במצב האינטראקטיבי של פייתון ובודקת האם הקלטים זהים. רק אם הם לא זהים נקבל שגיאה:

"""

This is the "example" module.

The example module supplies one function, factorial(). For example,

>>> factorial(5)

120

"""

def factorial(n):

"""Return the factorial of n, an exact integer >= 0.

>>> [factorial(n) for n in range(6)]

[1, 1, 2, 6, 24, 120]

>>> factorial(30)

265252859812191058636308480000000

>>> factorial(-1)

Traceback (most recent call last):

...

ValueError: n must be >= 0

Factorials of floats are OK, but the float must be an exact integer:

>>> factorial(30.1)

Traceback (most recent call last):

...

ValueError: n must be exact integer

>>> factorial(30.0)

265252859812191058636308480000000

It must also not be ridiculously large:

>>> factorial(1e100)

Traceback (most recent call last):

...

OverflowError: n too large

"""

import math

if not n >= 0:

raise ValueError("n must be >= 0")

if math.floor(n) != n:

raise ValueError("n must be exact integer")

if n+1 == n: # catch a value like 1e300

raise OverflowError("n too large")

result = 1

factor = 2

while factor <= n:

result \*= factor

factor += 1

return result

if \_\_name\_\_ == "\_\_main\_\_":

import doctest

doctest.testmod()

בדוגמא לעיל לא קיבלנו שום שגיאה בהרצה של הקוד, זה אומר שהdoctest עבר, אבל אם היינו משנים את התשובות של אחד הטסטים היינו מקבלים שגיאה, למשל נשנה את הטסט הראשון ב- factorial(5) מ-120 ל-125, נריץ ונקבל:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

File "\_\_main\_\_", line 7, in \_\_main\_\_

Failed example:

factorial(5)

Expected:

125

Got:

120

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1 items had failures:

1 of 1 in \_\_main\_\_

\*\*\*Test Failed\*\*\* 1 failures.

אם אחרי שעברנו את הטסט בכל זאת נרצה לראות מה הפונקציה ניסתה לבחון , אפשר להוסיף- '-v' לשורת ה'קימפול' של הקובץ והוא יציג את התוצאה.

python my\_doctest.py -v

Trying:

factorial(5)

Expecting:

120

ok

Trying:

[factorial(n) for n in range(6)]

Expecting:

[1, 1, 2, 6, 24, 120]

ok

Trying:

factorial(30)

Expecting:

265252859812191058636308480000000

ok

Trying:

factorial(-1)

Expecting:

Traceback (most recent call last):

...

ValueError: n must be >= 0

ok

Trying:

factorial(30.1)

Expecting:

Traceback (most recent call last):

...

ValueError: n must be exact integer

ok

Trying:

factorial(30.0)

Expecting:

265252859812191058636308480000000

ok

Trying:

factorial(1e100)

Expecting:

Traceback (most recent call last):

...

OverflowError: n too large

ok

2 items passed all tests:

1 tests in \_\_main\_\_

6 tests in \_\_main\_\_.factorial

7 tests in 2 items.

7 passed and 0 failed.

Test passed.

אפשר גם לבצע את הטסטים בקובץ טקסט חיצוני, ולקרוא לו דרך הפונקציה בתוכנית:

The ``my\_doctest`` module

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

Using ``factorial``

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

This is an example text file in reStructuredText format. First import

``factorial`` from the ``example`` module:

>>> from my\_doctest import factorial

Now use it:

>>> factorial(6)

120

import doctest

doctest.testfile("my\_doctest.txt")  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
File "my\_doctest.txt", line 14, in my\_doctest.txt   
Failed example:  
 factorial(6)   
Expected:  
 120   
Got:  
 720

## *2.* unitTest

יוניטסט הוא חלק אינטגרלי בכל תוכנית שאנחנו כותבים בכל שפת תכנות, היוניטסט כל-כך בסיסי שכחלק מהספרייה הסטנדרטית של פייתון יש לנו מודל שקוראים לו unittest.   
אז איך בכלל משתמשים ביוניטסט? ניקח לדוגמא את המחלקה גימטרייה שעשינו בשיעור על מתודות קסם. אנחנו נרצה לבנות מסמך חדש שבוחן את מקרי הקצה של הפונקציה עם משתנים מסוימים. כמוסכמה, השם של מסמך יוניטסט הוא כשם המסמך אותו הוא בא לבחון בתוספת המילה test\_ לפני, למשל במקרה שלנו:   
test\_gymatria.py .  
כל מסמך יוניטסט מתחיל ביצירת מחלקה חדשה שיורשת מהמחלקה TestCase . כל מתודה שבה אנחנו מבצעים את הטסטים חייבת להתחיל במילה test\_ בתחילת שמה (אחרת היא לא תוכר כמתודת טסט), וחייבת להכיל self כפרמטר.   
הטסטים במתודה אמורים להיות בצורה של טסט assert כלשהו, למשל עבור האופרטור '+' נבנה פונקציה שבוחנת אותו, נקרא לה למשל test\_add ונבחן בפונקציית assert כלשהי כל מיני מקרי קצה אפשריים :

import unittest

from gymatria import Gymatria

class TestGymatria(unittest.TestCase):

    def test\_add(self):

        self.assertEqual(Gymatria('אבא')+Gymatria('אמא') , 46)

        self.assertEqual(Gymatria('אבא')+Gymatria('fi') , 4)

        self.assertEqual(Gymatria('אבא')+Gymatria('אבא') , 8)

        self.assertEqual(Gymatria('אבא')+ Gymatria('') , 4)

        self.assertEqual( Gymatria('') + Gymatria('') , 0)

אם אנחנו רוצים להריץ את הטסט יש לנו שתי אופציות: אופציה ראשונה הוא ליצור פונקציית main() שתפעיל את הפונקציה main של המודול unitest ואז לקמפל משורת הפקודה כפי שקימפלנו עד עכשיו:

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

ובשורת הפקודה: python test\_gymatria.py  
  
אפשרות שניה היא ישר להריץ משורת הפקודה, בלי פונקציית main אם אנחנו מגדירים שהקובץ יורץ ע"י המודול unittest, איך עושים את זה? ע"י הוספת הדיגלון –m שמגדיר מאיזו פונקציית main התוכנית רצה, והוספת שם המודול ושם הקובץ:

python –m unittest test\_gymatria.py  
  
אם נריץ את הסקריפט נקבל שעברנו מבחן אחד:

.

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

Ran 1 test in 0.001s

OK

וזה משום שכל מתודה במחלקה היא מבחן בפני עצמו. הנקודה למעלה מגדירה כמה מבחנים עברו מתוך כלל הטסטים, ולמטה כתוב ממש כמה מבחנים רצו בכמה זמן ,והתוצאה –עובר או לא עובר.  
אם נשנה מבחן אחד , למשל ניכשל בכוונה באחד מהטסטים תתקבל ההודעה הבאה:

...g\3.unittest\test\_gymatria.py", line 9, in test\_add

self.assertEqual(Gymatria('אבא')+ Gymatria('') , 5)

AssertionError: 4 != 5

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

Ran 1 test in 0.001s

FAILED (failures=1)

כתוב לנו כמה טעויות היו לנו במבחן, באילו שורות ומה תוכן הטעות.

עכשיו עבור טסט שאמור להחזיר לנו שגיאה יש שתי אופציות - אופציה אחת תהיה להשתמש בפונקציה assertRaise() שמקבלת את הארגומטים: טיפוס השגיאה, שם הפונקציה, ופרמטרים. אופציה נוספת תהיה להשתמש בcontext manger כדי לנהל את השגיאה:

    def test\_raise(self):

# first way:

        self.assertRaises(ValueError , Gymatria.get\_value , None)

        # second way:

        with self.assertRaises(ValueError):

            Gymatria.get\_value()

עוד משהו אחרון בנושא. נניח שאנחנו רוצים לשמור משתנים שיאותחלו בתחילת כל טסט, אפשרות אחת תהיה להכריז עליהם בתחילת הטסט. אבל אם יהיה לנו קובץ טסט גדול ונרצה לשנות את אחד המשתנים בקוד נצטרך לשנות אותו בכל מופע שלו, לכן נוכל להשתמש במתודה setUp() שהיא מתבצעת לפני כל טסט והנתונים שלה נשמרים גם בטסט, ובמתודה teardown() שמתבצעת אחרי כל טסט, שימושית בעיקר כשמשתמשים בקבצים שצריך לסגור לפני כל טסט.  
משום שמתודות הן מתודות של האובייקט, נצטרך לפנות למשתנים כ-self:

def setUp(self):

        self.aba = Gymatria('אבא')

        self.ab = Gymatria('אב')

        self.efes = Gymatria('')

        self.aima = Gymatria('אמא')

        self.df = Gymatria('dfא')

    def tearDown(self):

        pass

    def test\_add(self):

        self.assertEqual(self.aba + self.aima , 46)

        self.assertEqual(self.aba + Gymatria('fi') , 4)

        self.assertEqual(self.aba + self.aba , 8)

        self.assertEqual(self.aba + self.efes , 4)

        self.assertEqual( self.efes + self.efes , 0)

        self.assertEqual( self.efes + 10 , 10)

יש עוד הרבה סוגים של asserts שאפשר להשתמש בהם ביוניטסט, ואפשר למצוא את רובם [כאן](https://docs.python.org/3/library/unittest.html).

## 3. PYTEST

המודול pytest הוא מודול חיצוני שנוצר כתחליף ל unittest, כדי לאפשר תחביר קצר ופשוט יותר, וכן להוסיף עוד אפשרויות ותוספים. עם זאת, ה-pytest יודע גם לזהות בדיקות של doctest ושל unittest, כך שאפשר לראות בו מערכת אחת שכוללת את כל סוגי הבדיקות. כיום, pytest היא מערכת הבדיקות הנפוצה ביותר.

כיוון שזה מודול חיצוני (לא חלק מפייתון), יש להתקין אותו כמו כל מודול:

pip install pytest

כמו ב-unittest, גם ב pytest, כל קובץ שמכיל בדיקות חייב להתחיל ב test\_.

בניגוד ל-unittest, הקובץ לא חייב להכיל מחלקות – אפשר לשים בו פונקציות כלשהן, בתנאי שהשם שלהן מתחיל גם-כן ב-test\_.

בגוף הפונקציה, לא צריך להשתמש בassertEqual וכל שאר הסוגים השונים, אלא רק ב-assert פשוט. למשל, במקום:

class TestGymatria(unittest.TestCase):

    def setUp(self):

        self.aba = Gymatria('אבא')

        self.aima = Gymatria('אמא')

    def test\_add(self):  # test functions must start with 'test\_'

        self.assertEqual(self.aba + self.aima , 46)

        self.assertEqual(self.aba + Gymatria('fi') , 4)

נכתוב:

aba = Gymatria('אבא')  
aima = Gymatria('אמא')  
def test\_add():  # test functions must start with 'test\_'

    assert aba + aima == 46

    assert aba + Gymatria('fi') == 4

בדיקת חריגות מתבצעת באופן הבא:

def test\_raise():

    with pytest.raises(ValueError):

        Gymatria.get\_value()

כדי להריץ את הבדיקות, מריצים בשורת הפקודה:

**pytest**

הפקודה הפשוטה הזאת אוספת את כל הבדיקות מכל הקבצים ששמם מתחיל ב test\_, מריצה אותם, וכותבת סיכום צבעוני של התוצאות (ראו תיקיה 3).

בתיקיה 4, שמנו את כל הבדיקות מהפרקים הקודמים: doctest, unittest, pytest. אם רוצים להריץ את כל הבדיקות ביחד, פשוט צריך להריץ משורת הפקודה:

**pytest --doctest-modules**

וזהו! כל הבדיקות ירוצו.

## 4. שילוב בדיקות אוטומטיות בתהליך הפיתוח – פעולות גיטהאב