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
    print(doctest.testmod())
```

בדוגמא לעיל לא קיבלנו שום שגיאה בהרצה של הקוד, זה אומר שה doctest 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 ב context 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.

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).

שימו לב: אם הפקודה pytest לא עובדת לכם, ייתכן שיש בעיה במסלולים במערכת ההפעלה. במקרה זה ייתכן שתצליחו להריץ:

python -m pytest

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.

אם רוצים לכתוב רק pytest בלי שנצטרך להוסיף את ה doctest-modules– כל הזמן (וכן אפשרויות נוספות שאולי נרצה להוסיף בהמשך), אפשר להשתמש בקובץ קונפיגורציה: מוסיפים בשורש של הפרוייקט קובץ בשם **pyproject.toml**, ובתוכו כותבים את כל האפשרויות שרוצים להוסיף, מתחת לכותרת המתאימה, למשל:

[tool.pytest.ini_options]

minversion = "6.0"

addopts = "--doctest-modules --ignore=__pycache__"

4. שילוב בדיקות אוטומטיות בתהליך הפיתוח – פעולות גיטהאב

ביצוע בדיקות אוטומטיות אחרי כל שינוי הוא מאד חשוב, ולכן ישנם כלים המאפשרים לנו לבצע זאת באופן אוטומטי. אנחנו נראה כלי אחד – GitHub Actions. הכלי הזה אמנם ייחודי לגיטהאב, אבל גם במערכות קוד אחרות שתעבדו בהם, יהיו כלים דומים (אולי עם תחביר שונה). המטרה שלי כאן היא ללמד את העיקרון של בדיקות אוטומטיות תמידיות.

אז מה זה GitHub Actions? – זה קובץ ששמים במאגר-גיטהאב שלנו, תחת התיקיה:

.github/workflows

נותנים לו שם כלשהו שמבטא את התפקיד שלו במערכת, עם סיומת **yml**. בתוך הקובץ, כותבים פקודות המסבירות לגיטהאב מה צריך לעשות בכל שלב מתהליך העדכון של המערכת. ניתן לראות דוגמה פשוטה בספריה prtpy:

https://github.com/erelsgl/prtpy/blob/main/.github/workflows/pytest.yml

הקובץ נקרא pytest.yml, והמטרה שלו היא להריץ pytest בכל פעם שמישהו מבצע push או pull request. הנה תוכן הקובץ בשלבים:

```
name: pytest
```

שם הפעולה.

```
on:
  push:
    branches:
      - main
      - master
  pull_request:
    branches:
      - main
      - master
```

באיזה מקרים הפעולה מתבצעת. במקרה זה, היא מתבצעת בכל פעם שמישהו דוחף שינויים חדשים לענף main או לענף master (לא לענפים אחרים, כי בענפים אחרים עושים שינויים שעדיין לא הסתיימו, ואין טעם לבדוק אותם). כמו כן, מבצעים את הפעולה בכל פעם שמישהו מבצע בקשת-משיכה.

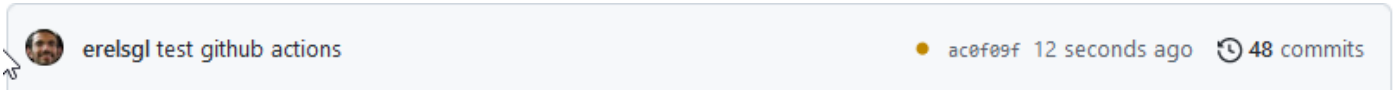
```
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      max-parallel: 4
      matrix:
        python: ["3.8", "3.9"]
```

כאן מתארים את הסביבות שעליהן נבצע את הפעולה. במקרה זה, אנחנו רוצים לבצע את הפעולה על הגירסה האחרונה של אובונטו, עם פייתון 3.8 ופייתון 3.9. זה מקובל לבדוק ספריות על כמה גירסאות של פייתון, כדי לוודא שאפשר להשתמש בהן גם בסביבות שבהן מותקנת גירסה ישנה יותר מהאחרונה. כמו כן, אנחנו מבקשים לבצע את הבדיקות בעזרת עד 4 תהליכים במקביל.

```
    steps:
      - uses: actions/checkout@v2
      - name: Setup Python
        uses: actions/setup-python@v2
        with:
          python-version: ${{ matrix.python }}
      - name: Upgrade pip
        run: python -m pip install --upgrade pip
      - name: Install pytest
        run: python -m pip install pytest
      - name: Install requirements
        run: python -m pip install -r requirements.txt
      - name: Run pytest
        run: python -m pytest
```

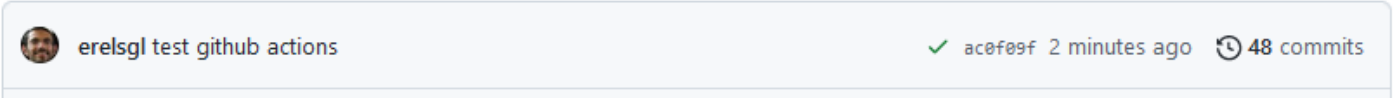
כאן מתארים את הצעדים שיש לבצע בכל אחת מהסביבות הנ"ל. קודם-כל מבצעים checkout, כלומר מורידים את הקוד לשרת של גיטהאב שעליו תתבצע הבדיקה. אחר-כך מתקינים את הגירסה המתאימה של פייתון. אחר-כך משדרגים את pip, מתקינים את pytest, וכן מתקינים את הספריות הדרושות לצורך הרצת הספריה – שנמצאות בקובץ requirements.txt. לסיום, מריצים את pytest.

כדי לראות איך זה עובד, נעשה שינוי קטן באחד הקבצים, ונדחוף אותו לגיטהאב. אם ניכנס מייד לדף של הספריה בגיטהאב, נראה עיגול כתום קטן ליד מזהה הקומיט שהגשנו:



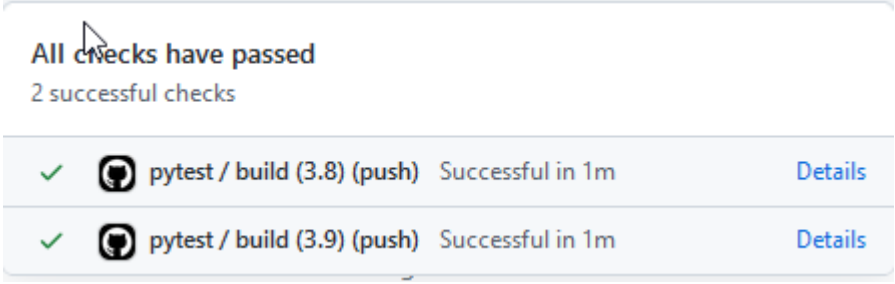
העיגול הזה אומר, שהבדיקות רצות עכשיו על השרת של גיטהאב. אם נלחץ עליו, נוכל לראות את פרטי הבדיקות המחכות בתור להרצה:

לאחר מספר דקות, אם הבדיקות יסתיימו בהצלחה, נראה סימן "וי" ירוק במקום העיגול הכתום:



הסימן הזה מעיד, שהקומיט שהגשנו עבר את כל הבדיקות. הידד!

אם נלחץ על הסימן, נראה את פרטי הבדיקות שעברו, ונוכל לראות מה בדיוק רץ ועבר:



אם חלק מהבדיקות נכשלו, במקום וי ירוק נראה איקס אדום, ושוב, בלחיצה עליו נוכל לראות איזה בדיקות נכשלו. יותר מזה, אנחנו נקבל הודעה בדואל, המודיעה לנו שיש באג בקומיט שלנו (אולי בעתיד תהיה אפשרות גם לבקש שיתקשרו אלינו לטלפון ויודיעו לנו על הבאג במענה אוטומטי...)

לסיום, דבר נחמד שאפשר להוסיף לרידמי שלנו: אפשר להוסיף קישור לתמונה, הנוצרת אוטומטית ע"י GitHub Actions, ואומרת אם הבדיקה עברה או נכשלה.

לדוגמה, אם מוסיפים את הקוד הבא בתחילת הרידמי של המאגר:

```
![Pytest result](https://github.com/erelsgl/prtpy/workflows/pytest/badge.svg)
```

(כאשר **pytest** הוא השם שנתננו לפעולה), נראה בדף הראשי של המאגר את התוית הבאה:

אם הבדיקות עברו, כך כל העולם יידע שהקוד שלנו עבר בדיקות pytest 😊

אם הבדיקות נכשלו – תופיע תוית דומה בצבע אדום, וזו תהיה אזהרה למתכנתים אחרים שירצו להשתמש בספריה שלנו, שידעו שיש בה באגים, וישתמשו בה על אחריותם בלבד.

לסיכום, ראינו כמה מערכות שימושיות מאד, שמשתלבות יחד כדי לעזור לנו לכתוב קוד איכותי, הנשאר תקין לאורך זמן.