DEBUGGING

Debugging (ניפוי שגיאות) הוא חלק אינטגרלי בפיתוח תוכניות קטנות כגדולות.
הניפוי הוא תהליך שיטתי של איתור והפחתת באגים בתוכנה, כדי לגרום לה לפעול כפי שהיא תוכננה.
באופן כללי, תהליך הניפוי הוא משימה מסורבלת ומעייפת, במיוחד בשפות תכנות דינאמיות כמו פייתון שבהן אין הכרזה על טיפוס המשתנים ואין קומפיילר אלא מפרש שמגדיר את טיפוס המשתנה רק בזמן ריצה.
בשיעורים הקודמים ראינו כמה כלים שאפשר לעזר בהן כדי לנפות שגיאות בפייתון, כלים כגון: הדפסות, פונקציית assert() ,טיפול בחריגות וכו'.
במסמך הזה נראה עוד כלים, יותר מסודרים, העוזרים לנו לאתר שגיאות לפי שהן מתרחשות, ואיך לטפל בהן בזמן אמת.

-DOCTEST

המודול doctest 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 פקוד 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 docstring , ואז בפונקציית maim אנחנו מפעילים את ה-dictest דרך module שנקרא באותו השם ומפעילים את הפונקציה ()testmode של המודול. הפונקציה לוקחת כל מקום במחרוזת של docstring 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 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 לגם לבצע את הטסטים בקובץ טקסט טסט 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
```

-LOGGING

לוגינג הוא כלי שמאפשר לעקוב אחרי השתלשות האירועים כשהתוכנה רצה, המפתח מוסיף קריאת לוגנג לקוד כדי לתת אינדיקציה שמאורע כלשהו התרחש.
לפייתון יש ספרייה המגיעה עם השפה של לוגינג ומספקת סט של פונקציות נוחות לשימוש.
הספרייה מאפשרת לנו להוסיף הודעות מצב (status massages) לקובץ או לאובייקט output אחר.
ההודעות האלה מכילות מידע כמו איזה חלק בקוד רץ ואילו בעיות עלולות להעלות.
לכל הודעת לוג יש רמה, חמש הרמות הבנויות מראש הן : Debug, Info , Warning, Error ו- Critical.
ניתן גם להגדיר רמות מתואמות אישית, אך רמות אלו אמורות להיות מספיקות.
כדי להוסיף אובייקט לוג חדש צריך דבר ראשון לייבא את הספרייה logging של פייתון. הספרייה היא בסיסית בשפה ואין צורך להוריד איזשהי ספרייה מתואמת אישית.
כדי ליצור אובייקט לוג חדש נצטרך דבר ראשון להגדיר היכן נפלוט את התוכן של הלוג עם הפונקציה basicCinfig() אח"כ נשמור את האובייקט שהתקבל במשתנה עם הפונקציה getLogger :

```
import logging

#create and configure logger
logging.basicConfig(filename= 'my_logging.log')
logger = logging.getLogger()
```

לכל רמה של הודעה יש פונקצייה ייעודית, למשל להוספת הודעת info נשתמש בפונקציה עם אותו שם, וכנ"ל לשאר הרמות. לכל רמה יש גם סוג של דירוג, הדירוג נועד להגדיר אילו סוגי הערות לא נראה כאשר אנחנו בקובץ לוג.
כל רמה היא גדולה מהבאה לפניה ב-10 , כלומר הדרגות מסודרות: 0=NOTSET , 10=DEBUG, 20= INFO, 30=WARRNING , 40= ERROR, ו- 50= CRITICAL. כברירת מחדל הלוגגר מוגדר להיות Warrning לכן כל ההודעות שהדירוג שלהן נמוך מ-30 לא יפלוט לקובץ.
אם עכשיו ננסה לפלוט לקובץ הודעת info או debug לא נוכל לראות אותה במסמך, לכן נצטרך להוסיף פרמטר level לפונקציה basicConfig() ולשים לו ערך קטן יותר מ-warning למשל debug:

```
import logging

#create and configure logger
logging.basicConfig(filename= 'my_logging.log' ,level= logging.DEBUG)
logger = logging.getLogger()

# Test the logger
logger.info('This is a message')
```

ואם נסתכל בקובץ לוג נראה שההודעה נשמרה כמו שצריך:

```
INFO:root:This is a message
```

ההודעה מכילה שלושה חלקים- את סוג ההודעה, במקרה זה הודעת info, את שם הלוגגר, במקרה זה root , ואת תוכן ההודעה.
אנחנו יכולים גם לשנות את מבנה ההודעה ע"י הוספה של ארגומנט format לפונקציה basicConfig() .
פייתון מספקת לנו כמה משתנים שנוכל להוסיף לפורמט של הודעת לוג למשל זמן עם המשתנה asctime או את מספר השורה lineno , אפשר למצוא את הרימה המלאה של השמות כאן.
הפורמט של ההודעה צריך להיות סוג של מחרוזת עם המשתנים בפנים:

```
import logging

LOG_FORMAT = "%(levelname)s, time: %(asctime)s , line: %(lineno)d- %(message)s "
#create and configure logger
logging.basicConfig(filename= 'my_logging.log' ,level= logging.DEBUG, format= LOG_FORMAT)
logger = logging.getLogger()

# Test the logger
logger.info('This is a message')
```

ואז הטקסט שנקבל הוא:

```
INFO:root:This is a message
INFO, time: 2021-01-31 19:40:44,441 , line: 9- This is a message
```

שימו לב שהטקסט הקודם לא נדרס, זה כי הקבוץ לוג הוא בפורמט 'r+' אם נרצה נוכל לשנות אותו לפורמט אחר ע"י הוספת ארגומנט filemode = 'w'.
ועכשיו נראה דוגמא של שימוש בלוג עם בקובץ.
נבנה פונקציה שמחשבת את נוסחת השורשים בהתאם לקלט של a,b,c:

```
def quadratic_formula(a:float, b:float ,c:float) -> float:
    import math
    """ Returns the solutions to the equation ax^2 + bx + c = 0  """
    logger.info(f'quadratic_formula({a},{b},{c})')
    logger.debug('Compute the discriminant')
    discri = b**2 - 4*a*c
    logger.debug('Compute the two roots')
    root_a = (-b + math.sqrt(discri))/(2*a)
    root_b = (-b - math.sqrt(discri))/(2*a)
    logger.debug('Return the two roots')
    return root_a , root_b
```

הפונקציה אמורה לכתוב לקובץ לוג לפני כל פעולה שמתבצעת בפונקציה.
ובאמת אם נריץ אותה על הקלט 1,0,-4 אנחנו אמורים לקבל את התשובה 2,-2 וזה אמור להופיע גם בקובץ:

```
print(quadratic_formula(1,0,-4))
with open('my_logging.log') as file:
    print(file.read())
```

```
(2.0, -2.0)
INFO, time: 2021-01-31 20:25:54,978 , line: 12- This is a message
INFO, time: 2021-01-31 20:25:59,159 , line: 4- quadratic_formula(1,0,-4)
DEBUG, time: 2021-01-31 20:25:59,159 , line: 5- Compute the discriminant
DEBUG, time: 2021-01-31 20:25:59,159 , line: 7- Compute the two roots
DEBUG, time: 2021-01-31 20:25:59,159 , line: 10- Return the two roots
```

אבל אם נוסיף ניתן לפונקציה קלט שהיא לא אמורה לזהות למשל 1,0, 1 נראה שלא נקבל יודפס ערך החזרה מהפונקציה וכך נדע היכן בדיוק קרתה השגיאה:

```
quadratic_formula(1,0,1)
```

ואז בקובץ לוג נראה:

```
INFO, time: 2021-01-31 20:31:03,650 , line: 4- quadratic_formula(1,0,1)
DEBUG, time: 2021-01-31 20:31:03,653 , line: 5- Compute the discriminant
DEBUG, time: 2021-01-31 20:31:03,653 , line: 7- Compute the two roots
```

במקרה זה הקוד קטן, אבל תחשבו על קוד גדול עם מלא שורות, לאתר את מקום התקלה יהיה קשה מאוד ללא קובץ לוג, בעזרתו נוכל לגעת ישירות באיזו פונקציה נתקנו, איזו שורה נתקנו, מה היה הקלט, מה הפלט וכך לדעת איך לתקן.

-UNITEST

ועכשיו הגענו לחלק המרגש ביותר בכתיבת קוד, הטסטים.
יוניטסט ולוגיניג דומים מאוד אך שונים תפיסתית, בשניהם אנחנו מנסים לאתר מקורות של באגים בתוכנית, אבל בעוד לוגינג עוזר לנו לאתר אותם בדיעבד , יוניטסט מאתר אותן לכתחילה. אנחנו בעצם מנסים לאתר מאיפה האויב יכול לתקוף ,ובאויב אני מתכוון בעיקר למתכנת שהכניס לפונקציה בטעות טיפוס שהיא לא יודעת לטפל בו (לרוב).
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 מתדות 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 .
כל מסמך יוניטסט מתחיל ביצירת מחלקה חדשה שיורשת מהמחלקה TestCase וכל 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 TestGymayris(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 unites 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. הנקודמה 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 שמתנים שיאותחלו בתחילת כל טסט, אפשרות אחת תיהיה 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.