**תרגיל בית מספר 2 בשפות תכנות סמסטר ב 2018**

הגשה בזוגות דרך moodle

רק אחד מכל זוג יגיש את התרגיל (וכמובן יציין את שמות שני המגישים).

**יש להגיש את כל הפתרונות בקובץ אחד שנקרא ex2.py**

עבור כל אחד מהפתרונות יש לכתוב unittest (תוך שימוש ב- import unittest) עם שתי דוגמאות לפחות.

בכל השאלות מותר להניח שהארגומנטים הם תקינים.

**שאלה 1**

כתבו decorator לפונקציות שנקרא memoize .

ה- decorator נועד לשימוש עם פונקציות שבהן בכל הקריאות משתמשים רק

ב- positional arguments (לא keyword arguments). ה- decorator מאחסן ב- cache

(הממומש כ- dictionary) את תוצאות הקריאות הקודמות לפונקציה אותה הוא מקשט. במקרה שקוראים לפונקציה שוב עם אותם הארגומנטים, התוצאה שמאוחסנת ב- cache מוחזרת וכך חוסכים חישוב חוזר של התוצאה.

ה- cache לא צריך להיות משתנה גלובלי (כלומר אין להגדירו ברמה של ה- module). במקום זאת השתמשו ב- closure.

דוגמא (<<< זה ה- prompt של ה- REPL)

@memoize

def sum2(x, y):

print('calling sum2')

return x + y

>>> sum2(5, 3)

calling sum2

8

>>> sum2(10, 20)

calling sum2

30

>>> sum2(5, 3)

8

שימו לב שבקריאה השניה ל- sum2(5, 3) גוף הפונקציה לא בוצע.

**שאלה 2**

כתבו מחלקה בשם Rational שמייצגת מספרים רציונליים.

המחלקה תתמוך בחיבור, חיסור, כפל וחילוק של מספרים רציונליים.

בנוסף היא תתמוך באופרטורים המאפשרים להשוות בין שני מספרים רציונליים (==, !=, <, <=, >, >=).

לצורך כך יש לממש את ה- special methods הבאים:

\_\_add\_\_

\_\_sub\_\_

\_\_mul\_\_

\_\_truediv\_\_

\_\_lt\_\_

\_\_le\_\_

\_\_eq\_\_

\_\_ne\_\_

\_\_gt\_\_

\_\_ge\_\_

הסבר קצר על ה- special methods נמצא בקישור https://docs.python.org/3/reference/datamodel.html

בנוסף לכך יש לכתוב גם את ה- special method \_\_str\_\_. זו תציג מספר רציונלי בצורה מצומצמת. למשל 24/60 יוצג כ- 2/5. לצורך כך יש למצוא את המכנה המשותף המקסימלי של שני המספרים. ראו https://en.wikipedia.org/wiki/Euclidean\_algorithm

(יש שם מספר אלגוריתמים -- לצרכים שלנו לא חשוב איזה מהאלגוריתמים תבחרו)

(נזכיר שב- Python3 חלוקה של שני מספרים שלמים מניבה תוצאה שהיא float.

כדי לקבל תוצאה שלמה אפשר להשתמש באופרטור floordiv (שמסומן //)

\_\_str\_\_ לעולם לא תציג את המכנה כמספר שלילי (במקרה הצורך, המונה יהיה שלילי).

כתבו גם את המתודה המיוחדת \_\_repr\_\_. גם היא תציג מספרים רציונליים בצורה מצומצמת והמכנה לעולם לא יהיה שלילי. למשל 24/60 יוצג כ- Rational(2, 5).

בנוסף לכך הגדירו class attribute בשם Rational.max שערכו יהיה המספר הרציונלי הגדול ביותר שנוצר בינתיים. (אם עדיין לא נוצר מספר רציונלי ערכו יהיה None).

המתודה \_\_init\_\_ תקבל שני ארגומנטים (בנוסף ל- self) -- המונה והמכנה.

במקרה שהמכנה חסר -- ברירת המחדל היא 1.

אפשר להניח שהמכנה אינו אפס.

דוגמא:

>>> half = Rational(5, 10)

>>> str(half)

'1/2'

>>> repr(half)

Rational(1, 2)

>>> Rational.max

Rational(1, 2)

>>> third = Rational(8, 24)

>>> str(third)

'1/3'

>>> half+third

Rational(5, 6)

>>> third-half

Rational(-1, 6)

>>> half\*third

Rational(1, 6)

>>> half/third

Rational(3, 2)

>>> half == third

False

>>> half != third

True

>>> half < third

False

>>> half <= third

False

>>> half > third

True

>>> half >= third

True

>>> Rational.max

Rational(3/2)

>>> two = Rational(2)

>>> repr(two)

'Rational(2, 1)'

שאלה 3

כתבו generator function בשם myZip שפועלת באופן דומה ל- zip builtin function (אבל קצת פשוטה ממנה). החתימה היא:

def myZip(iterable1, iterable2, fill=None)

myZip מקבלת שני iterables כארגומנטים. היא מניבה סדרה של tuples . ה- tuple

הראשון כולל את שני הפריטים הראשונים של שני ה- iterables. השני כולל את שני הפריטים

השניים וכן הלאה. במקרה שאחד משני ה- iterables מניב סדרה קצרה יותר מהשני,

יש להשלים את הסדרה הקצרה עם הערך של הפרמטר fill.

אם הערך של fill הוא None (וזה גם ה- default) יש לעצור ברגע שממצים את הסדרה הקצרה יותר.

שימו לב שהפונקציה אמורה לפעול על iterable כלשהו. לכן לא ניתן להניח שניתן להפעיל את הפונקציה len עליו או שניתן לגשת אליו תוך שימוש באינדקס (למשל iterable1[7]).

זאת משום שיש iterables (למשל generator object שהוא הערך של generator expression) שלא תומכים בפעולות האלו.

כל מה שניתן להניח הוא שה- iterable תומך בפרוטוקול האיטרציה (ניתן להפעיל עליו iter ו- next).

דוגמא :

>>> g = (3\*i for i in range(5))

>>> for first, second in myZip(g, ["a", "b", "c"],fill="bye"):

print(first, second)

0 "a"

3 "b"

6 "c"

9 "bye"

12 "bye"

בדוגמא זו ה- iterable הראשון הוא generator object שהוא התוצאה של generator expression. ה- iterable השני הוא list.

הנחיה: כנראה שבמקרה זה לא ניתן להסתפק באיטרציה אוטומטית של ה- irterables (למשל ע"י לולאת for ) ויש צורך לקרוא ל- iter ול- next בצורה מפורשת.

שימו לב שלפונקציה next יש ארגומנט שני אופציונלי שנוח להשתמש בו אם לא רוצים לטפל

ב- StopIteration -- ראו תעוד בקישור https://docs.python.org/3/library/functions.html#next

שאלה 4

חזרו על השאלה בסעיף א אבל הפעם במקום להשתמש ב- generator function

יש להגדיר מחלקה (class) בשם MyZip. (השם מתחיל באות גדולה להבדיל משם הפונקציה של הסעיף הקודם). מחלקה זו צריכה לתמוך באיטרציה. הקוד שהובא בתור דוגמא

בשאלה הקודמת צריך לעבוד (עם השינוי המתבקש שבמקום myZip יש לכתוב MyZip).

שאלה 5

כתבו פונקציה anagram(word) המקבלת מחרוזת word כארגומנט ומחזירה רשימה

של כל המילים האנגליות המורכבות מאותן אותיות המופיעות ב- word כאשר האותיות

מופיעות בסדר כלשהו. (האורך של כל מילה ברשימת התוצאה יהיה כמו האורך של word).

המילים ברשימת התוצאה תהיינה ממוינות בסדר לקסיקוגרפי.

למשל anagram('restful') יחזיר את הרשימה

['fluster', 'fluters', 'restful']

anagram('rrao') יחזיר ['orra', 'roar']

הדרכה

כדאי להיעזר באחת הפונקציות המופיעות במודול itertools.

השתמשו בקובץ words.txt (המצורף לתרגיל) שמכיל למעלה ממאה אלף מילים באנגלית.

(הקובץ הורד מ- <http://greenteapress.com/thinkpython2/code/words.txt>

והוא מוזכר בספר Think Python 2 : How to Think Like a Computer Scientist של Allen Downey)

לצורך התרגיל הזה מספיק לדעת את הדברים הבאים על קריאת קבצים:

open('words.txt') fin = פותח את הקובץ ומחזיר file object שיכול לשמש לקריאת הקובץ. איטרציה על ה- file object מניבה את השורות של הקובץ.

למשל

for line in open('words'.txt):

line = line.strip()

do something with line

כאן strip משמש להורדת white space מתחילת השורה ומסופה.

(בלי קשר לקריאה מקבצים -- יתכן שתרצו להשתמש בפתרון גם ב- join שהיא method שמשרשרת מחרוזות).

שאלה 6

השאלה עוסקת ב- descriptors.

כתבו מחלקה בשם Variable. כל instance של המחלקה מייצג "משתנה" שיש לו ערך שלם למחלקה יש methods הנקראים history, readCounter, writeCounter. history() מחזירה רשימה של כל הערכים שהיו למשתנה בסדר כרונולוגי כלומר הפריט הראשון ברשימה יהיה הערך ההתחלתי . הפריט האחרון ברשימה יהיה הערך הנוכחי.

readCounter() מחזירה את מספר הפעמים בהם ניגשו למשתנה לצורך קריאה. writeCounter() מחזירה את מספר הפעמים בהם ניגשו למשתנה לצורך כתיבה.

הנה דוגמא לשימוש במחלקה Variable:

>>> v = Variable(30)

>>> v.history()

[30]

>>> v.readCount()

0

>>> v.WriteCount() # first write access was done when v was instantiated

1

>>> print(v.value) #this is a read access

30

>>> v.readCount()

1

>>> v.writeCount()

1

>>> v.value = 17 # this is a write access

>>> v.history()

[30, 17]

>>> v.WriteCount()

2

>>> v.value = 17

>>> v.writeCount()

3

>>> v.history()

[30, 17, 17]

>>> v.value # another read access

17

>>> v.readCount()

2

הדרכה:

לכל instance של Variable יהיו attributes בשם \_value, \_history . הלקוחות של המחלקה לא אמורים לגשת אליהם ישירות.

ב- \_value ישמר הערך הנוכחי. ב- \_history תשמר רשימה הכוללת את כל הערכים שהיו למשתנה. בנוסף לכך תצטרכו להגדיר instance attributes אחד או שניים נוספים.

value יהיה class attribute של המחלקה Variable. (לא לבלבל בין זה לבין \_value

שיהיה instance attribute). value זה יהיה descriptor השייך למחלקה VariableDescriptor אותה עליכם להגדיר. תצטרכו להגדיר methods בשם \_\_get\_\_ ו- \_\_set\_\_ עבור מחלקה זו.

נניח ש- v הוא instance של Variable. אז כל גישה ל- v.value לצורך קריאה

(למשל print(v.value) ) תגרום לקריאה ל- Variable.value.\_\_get\_\_(v, Variable) .

אין צורך להשתמש כאן בארגומנט האחרון של \_\_get\_\_.

גישה ל- v.value לצורך כתיבה (binding) (למשל v.value = 10) תגרום לקריאה ל- \_\_set\_\_(v, 10) Variable.value..

\_\_get\_\_ ו- \_\_set\_\_ יצטרכו לעדכן את ה- instance attributes של v .