מתודות מיוחדות ב-

- חלק שני -

אלי כהן-נחמיה

מבוא

בחלקו הראשון של המסמך הראינו את המוטיבציה לשליטה במתודות המיוחדות ב-Python ואף תרגלנו את המתודות הבסיסיות. בחלק הנוכחי נסקור מתודות נוספות ומתקדמות יותר; גם סקירה זו תלווה בבניית מחלקה חדשה אשר תממש מתודות מיוחדות לצורך המחשת העמסת האופרטורים.

המחלקה אותה נממש בחלק זה של המסמך מצריכה ידע מתמטי ברמה גבוהה אך במעט מעל רמה תיכונית. המחלקה תייצג פולינום סופי במשתנה אחד אשר מגדיר פונקציה פולינומית. לצורך יישור קו נזכיר את תכונותיו הבסיסיות של הפולינום ומושגים בסיסיים הקשורים בו:

- פולינום הוא ביטוי מתמטי השווה לסכום חזקות של משתנה מסויים; המשתנה יכול להופיע כחלק ממכפלה עם מקדם מספרי, לדוגמה: $p(x) = 2x^3 + x^2 + 4x + 3$
- פולינום מיוצג על-ידי סדרת מקדמי המשתנה כך שמיקום המקדם בסדרה מעיד על מעריך החזקה של המשתנה בו הוא מוכפל: מקדם המשתנה x^0 (המקדם החופשי) יופיע במקום ה-0 בסדרה וכך הלאה. סדרת המקדמים (משמאל לימין) עבור הפולינום לעיל הינה [3, 4, 1, 2]
- נשים לב כי מקדמים השווים לאפס המופיעים לאחר המקדם שאינו אפס במיקום הגבוה ביותר אינם משפיעים כלל על הפולינום; תיאורטית כל פולינום מיוצג על-ידי סדרת מקדמים אין-סופית אשר "מרופדת" במקדמי אפס חסרי השפעה. סדרת המקדמים הבאה מייצגת את אותו הפולינום בדיוק: [3, 4, 1, 2, 0, 0, 0]
- מעלת הפולינום (מכונה גם "דרגה") היא המעריך הגבוה ביותר שלו מקדם שונה מאפס; אם נתרגם זאת לייצוג על-פי סדרת מקדמים, מעלת הפולינום היא האינדקס הגבוה ביותר בסדרה בו קיים איבר שונה מאפס.

הערות נוספות:

- ניתן לחבר, לחסר, להכפיל ואף לחלק פולינום אחד באחר.
- פונקציות פולינומיות ניתנות להרכבה ולגזירה.
- פונקציה פולינומית ממעלה אפס (כלומר שהמקדם היחיד שאינו אפס בה הוא המקדם החופשי) נקראת גם "פונקציה קבועה"; עבור כל x תחזיר הפונקציה את אותו הערך והוא ערך המקדם.
- פונקציה ממעלה ראשונה שסדרת המקדמים שלה היא [0, 1] נקראת "פונקצית הזהות"; עבור כל x תחזיר הפונקציה את ערכו של x .
- פונקציה קבועה שהמקדם היחיד בה הוא אפס נקראת "פונקצית האפס" ותחזיר אפס עבור כל x .

בדומה לחלקו הראשון של המסמך ולמחלקת BitField שהוזכרה בו, גם כאן נגדיר הנחות מקילות על המחלקה המיועדת: מרגע אתחולו, אובייקט פולינומי לא יהיה ניתן לשינוי – כלומר סדרת מקדמיו תשאר קבועה. כפועל יוצא מכך מעלתו של הפולינום ואופן הצגתו לא ישתנו גם הם.

גם הפעם ניתן להוריד את קוד המקור המלא בליווי הערות ודוגמאות נוספות מכאן.

אתחול ותכונות בסיסיות

נפתח בהגדרת המחלקה ובמתודת האתחול. מתודת האתחול תקבל את סדרת המקדמים (Coefficients) המייצגים את הפולינום ומכיוון שאלו לא ישתנו, תחשב את מעלתו ואת ייצוגו המחרוזתי (מושג זה יוסבר בהמשך) כבר עתה. נשים לב כי לאחר מציאת מעלת הפולינום ניתן יהיה להשמיט את מקדמי האפס שלאחריה מבלי לפגוע בפולינום. כמו כן פולינום המאותחל ללא סדרת מקדמים יחשב כ-"פונקצית האפס".

```
 1 class Polynomial:
 2
 3   def __init__(self, *coefs):
 4     if not coefs:
 5       coefs = (0,)
 6
 7     self.deg = Polynomial._calculate_deg_(coefs)
 8     self.str = Polynomial._calculate_str_(coefs, self.deg)
 9
10     self.coefs = coefs[:self.deg + 1]
```

פונקציות העזר לחישוב המעלה והייצוג המחרוזתי יוגדרו כפונקציות סטאטיות במחלקה. כיוון שפונקציות אלה אינן מעניינו הישיר של נושא המסמך נדלג על הסבר אופן פעולתן. הסברים, דוגמאות והערות בגוף הקוד ניתן למצוא בקובץ קוד המקור.

```
11 @staticmethod
12 def _calculate_deg_(coefs):
13   nonzeros = tuple(i for i, c in enumerate(coefs) if c)
14   degree = nonzeros[-1] if nonzeros else 0
15
16   return degree
17
18 @staticmethod
19 def _calculate_str_(coefs, deg):
20   str_elements = ['%dx^%d' % (c, i) for i, c in enumerate(coefs)]
21
22   str_elements[0] = str_elements[0][:-3]
23
24   if deg > 0:
25     str_elements[1] = str_elements[1][:-2]
26     str_elements = [elem for elem in str_elements if not ↵
   elem.startswith('0')]
27     str_elements = [elem[1:] if elem.startswith('1x') else elem for elem ↵
28 in str_elements]

29   return ' + '.join(reversed(str_elements))
```

נציג מספר דוגמאות לאתחול מופעי Polynomial:

```
>>> f = Polynomial(4, 6, 8)       # f(x) = 8x^2 + 6x + 4
>>> g = Polynomial(4, 6, 8, 0, 0) # g(x) = 8x^2 + 6x + 4
>>> h = Polynomial()              # h(x) = 0 (for every x)
```

ייצוג מחרוזתי ובוליאני

בדומה למעלתו של הפולינום, ייצוגו המחרוזתי מחושב גם הוא בזמן אתחול האובייקט. ייצוגו המחרוזתי של אובייקט נקבע על-פי המחרוזת המוחזרת מהמתודה המיוחדת__str__ . מתודה זו נקראת בכל פעם בה נדרש ייצוגו המחרוזתי של אובייקט כגון ציון שמו של האובייקט כחלק מפקודת הדפסה, שרשור האובייקט למחרוזת או בקשה מפורשת לייצוג מחרוזתי באמצעות הפונקציה המובנית str. לאחר בניית הייצוג במתודת האתחול מימוש __str__ עצמה פשוט:

```
30 def __str__(self):
31   return self.str
```

דוגמה לייצוגו המחרוזתי של פולינום:

```
>>> p = Polynomial(6, 0, 2, 1)
>>> print 'p(x) = %s' % str(p)
p(x) = x^3 + 2x^2 + 6
```

כפי שהוזכר בחלק הקודם, "אורכו" של מופע נקבע על-פי הערך המוחזר ממתודת __len__ שלו. בנוסף לכך, ייצוגו הבוליאני נשען גם הוא על אותה מתודה כך שמופע שווה ל-False במידה ומתודת __len__ מחזירה עבורו אפס. בהקשר הפולינומי ניתן לומר כי "אורכו" של פולינום הוא מעלתו; מכיוון שמעלתו של הפולינום חושבה עוד בזמן אתחול האובייקט, נגדיר את המתודה __len__ באופן דומה להגדרת __str__:

```
32 def __len__(self):
33   return self.deg
```

דוגמה:

```
>>> f = Polynomial(3, 0, 0, 1, 0, 0, 0)
>>> print 'degree of f is %d' % len(f)
degree of f is 3
```

נשים לב כי הגדרת __len__ אינה מדוייקת מספיק; כאשר נדרשים לבחון את ייצוגו הבוליאני של הפולינום מימושה הנוכחי של מתודת __len__ יקבע כי פולינום יוגדר כ-False אם מעלתו היא אפס, כלומר הוא מייצג פונקציה קבועה. טבעי יותר להגדיר את ייצוגו הבוליאני של פולינום כשווה ערך להיותו פונקציית האפס (מקרה פרטי של פונקציה קבועה), כלומר מעלתו היא אפס אך גם המקדם הראשון (והיחיד) בו הוא אפס.

בכדי לנתק את הקשר בין מתודת __len__לבין ייצוגו הבוליאני של האובייקט נגדיר את המתודה __nonzero__ שתקבל את האחריות הבלעדית על הגדרתו הבוליאנית של האובייקט:

```
34 def __nonzero__(self):
35   return not (self.deg == 0 and self.coefs[0] == 0)
```

דוגמה לשימוש בייצוגו הבוליאני של פולינום:

```
>>> z = Polynomial(0)
>>> if z:
...   print 'z(x) = %s' % str(z)
... else:
...   print 'z(x) = zero polynomial'
z(x) = zero polynomial
```

הפעלה והרכבה

הפונקציונליות הבסיסית ביותר של פונקציה פולינומית הינה חישוב ערך הפונקציה בנקודה מסויימת. כלומר הפעלת הפונקציה f על x או בסימונה המקובל: f(x).

האפשרות ל-"הפעלת" אובייקט תוך שימוש בסימון הדומה לקריאה לפונקציה מותנית בהגדרתה של המתודה המיוחדת __call__. מתודה זו מקבלת את רשימת הפרמטרים המבוקשת בהפעלתו של האובייקט. כאשר מדובר בפונקציה פולינומית על איבר יחיד, ידרש ארגומנט אחד:

```
36 def __call__(self, x):
37   horner = lambda val, c: c + x * val
38
39   return reduce(horner, reversed(self.coefs))
```

הערה: מימוש המתודה עושה שימוש ב-שיטת Horner אשר נמצאה יעילה יותר מחישוב פשטני על-ידי צמצום מספר ההכפלות הנדרשות לצורך החישוב.

דוגמאות:

```
>>> f = Polynomial(3, 1) # f(x) = x + 3
>>> f(4)
7
>>> id = Polynomial(0, 1) # id(x) = x
>>> id(42)
42
```

הרכבת פונקציות מוגדרת כך שפונקציה אחת פועלת על תוצאת פונקציה אחרת, כלומר: בהנתן הפונקציות f ו-g, הרכבת f על g היא למעשה: f(g(x)). להרכבת פונקציות קיימת נוטציה מתמטית מקובלת נוספת אך היא אינה נתמכת כאופרטור בשפת Python; אי לכך נסתפק בהרכבה המתקבלת מהפעלת פונקציה אחת על תוצאה של אחרת. הרכבה כזו מתאפשרת, כמובן, על-ידי מתודת __call__ אותה מימשנו זה עתה.

דוגמה:

```
>>> f = Polynomial(3, 1)     # f(x) = x + 3
>>> g = Polynomial(0, 0, 1)  # g(x) = x^2
>>> f(g(5))
28
>>> g(f(5))
64
```

הערה: המתודה __call__ משמשת את הפונקציה המובנית callable לצורך אינדיקציה האם מחלקה ניתנת ל-"הפעלה".

השוואה

על פולינומים ופונקציות אמנם לא מוגדר יחס סדר כלשהו (כלומר לא ניתן להחליט מי יופיע לפני מי במיון כלשהו) כך שאין משמעות לאופרטורים "גדול מ-" ו-"קטן מ-" בהקשר זה אך בהחלט ניתן לקבוע האם שני פולינומים שווים זה לזה או שונים זה מזה.

הערה: באופן כללי מתודות מיוחדות המשמשות למימוש אופרטור בינארי כלשהו מניחות כי האובייקט "שלהן" הוא זה הנמצא בצידו השמאלי של האופרטור. לעומת זאת בצידו הימני של האופרטור מופיע אובייקט עליו לא ניתן להניח הנחות כלשהן; אין כל הבטחה כי בצד הימני ימצא אובייקט דומה או אפילו בר השוואה לאובייקט הנמצא בצד שמאל. מסיבה זו בדרך-כלל פותחת מתודה כזו בסדרת תהיות על קנקנו של האובייקט השני. אובייקט זה מתקבל כפרמטר למתודה – נקרא לו כאן other.

שני פולינומים יחשבו שווים אם סדרות המקדמים שלהם עד למעלתן - שוות; נשים לב שהגדרה זו טומנת בחובה גם את העובדה כי מעלותיהם שוות. מכיוון שהשוואת שתי רשימות עלולה להיות "יקרה" חישובית, נשתמש במעלותיהם כדי לאפשר אופטימיזציה קלה: ראשית נתנה את השוויון בכך שמעלותיהם של הפולינומים שוות ורק אז נשווה גם את הרשימות עצמן.

לשם העמסת אופרטור השוויון נגדיר את המתודה __eq__ באופן שהוזכר לעיל אך ראשית נוודא כי האובייקט המופיע בצידו הימני של אופרטור השוויון הוא בר השוואה, כלומר גם הוא אובייקט פולינום:

```
40 def __eq__(self, other):
41   if not isinstance(other, Polynomial):
42     return False
43
44   return (self.deg == other.deg) and (self.coefs == other.coefs)
```

באופן מפתיע, אי-שוויון בין אובייקטים אינו מוגדר כתשובה ההופכית לתוצאת המתודה __eq__. מכאן עולה כי Python מאפשרת לשני אובייקטים להיות שווים ולא שווים בו זמנית. אם כן, נגדיר במפורש את אי השוויון עבור שני פולינומים באמצעות המתודה הקודמת; אופרטור השוויון על אובייקט הפולינום self יפעיל את __eq__ אותה מימשנו זה עתה:

```
45 def __ne__(self, other):
46   return not (self == other)
```

דוגמאות:

```
>>> f = Polynomial(3, 1)
>>> g = Polynomial(3, 1, 0, 0, 0)
>>> f == g
True
```

פעולות השוואה נוספות שלא הוזכרו:

- __lt__ אופרטור "קטן מ-"
- __le__ אופרטור "קטן או שווה ל-"
- __gt__ אופרטור "גדול מ-"
- __ge__ אופרטור "גדול או שווה ל-"
- __cmp__ מתודה רב תכליתית המכסה את כל סוגי ההשוואות שלא הוגדרו באופן פרטני עבור המחלקה. מתודה זו תחזיר 1, 0, או -1 אם האופרנד השמאלי גדול, שווה או קטן מהאופרנד הימני (בהתאמה).

פעולות חשבוניות

ארבעת פעולות החשבון חיבור, חיסור, כפל וחילוק מוגדרות היטב על פולינומים ופונקציות. ארבעת הפעולות אינן מוגדרות באופן דומה אחת לשניה ויוצאת דופן במיוחד היא פעולת החילוק המסובכת בהרבה משלושת האחרות ולכן נזנח אותה במסמך זה.

הפעולות אותן נממש הן פעולות וקטוריות הפועלות על מקדמי הפולינומים. פעולות אלה דורשות כי רשימות מקדמי הפולינומים יהיו באורך שווה. מכיוון שפולינום עשוי להיות מיוצג על-ידי סדרת מקדמים באורך שונה מחברו, נממש פונקצית עזר שתפקדה הוא "ריפוד" סדרת המקדמים במקדמי אפס עד לאורך מבוקש.

נזכיר כי מבחינה מתמטית מקדמי אפס הבאים לאחר מעלת הפולינום אינם משנים את התנהגותו וניתן להתעלם מהם; למעשה כל פולינום ניתן לרפד באינסוף מקדמי אפס מעל למעלתו. נדגים זאת באמצעות פולינום ממעלה שניה אשר רופד בשני מקדמי אפס; קל להבחין שאין הם משפיעים כלל על הפולינום: $p(x) = 0x^4 + 0x^3 + 2x^2 + 4x + 1$

פונקצית הריפוד תקבל את האורך המבוקש ותחזיר את סדרת המקדמים המרופדת, אם ריפוד כזה אכן נדרש:

```
47 def _pad_to_(self, size):
48   return self.coefs + (0,) * (size - len(self.coefs))
```

פעולות החיבור, החיסור והכפל מוגדרות בהתאמה באופן הבא:

- $(f + g)(x) \Leftrightarrow f(x) + g(x)$
- $(f - g)(x) \Leftrightarrow f(x) - g(x)$
- $(f \cdot g)(x) \Leftrightarrow f(x) \cdot g(x)$

בבואינו לחבר שני פולינומים או לחסר פולינום אחד מחברו נניח באופן דמיוני את סדרות מקדמי הפולינומים זו מעל זו. פעולות החיבור והחיסור למעשה מחברות או מחסרות מקדמים במקומות תואמים בסדרה. מימושן, אם כן, פשוט:

```
49 def __add__(self, other):
50   if not isinstance(other, Polynomial):
51     raise TypeError('Right hand of operator should be a Polynomial↵
   instance')
52
53   padded_self = self._pad_to_(len(other.coefs))
54   padded_other = other._pad_to_(len(self.coefs))
55
56   coefs = (a + b for a, b in zip(padded_self, padded_other))
57
58   return Polynomial(*coefs)
```

```
59 def __sub__(self, other):
60   if not isinstance(other, Polynomial):
61     raise TypeError('Right hand of operator should be a Polynomial↵
   instance')
62
63   padded_self = self._pad_to_(len(other.coefs))
64   padded_other = other._pad_to_(len(self.coefs))
65
66   coefs = (a - b for a, b in zip(padded_self, padded_other))
67
68   return Polynomial(*coefs)
```

לעומת פעולות החיבור והחיסור מכפלת שני פולינומים הינה פעולה מורכבת יותר הדורשת הכפלה תוך "פתיחת סוגריים" של שתי סדרות המקדמים. בנוסף לכך נרחיב את משמעות אופרטור הכפל: אם צידו הימני של האופרטור הוא פולינום נכפיל את שניהם באופן שתואר לעיל אך אם צידו הימני של האופרטור הוא קבוע מספרי, נבצע מכפלה פשוטה: כל מקדם יוכפל בתורו בקבוע הנתון, כלומר: $(f \cdot n)(x) \Leftrightarrow f(x) \cdot n$ עבור n קבוע מספרי כלשהו.

הערה: חדי העין יבחינו כי למעשה כפל פולינום בקבוע הוא מקרה פרטי של כפל שני פולינומים כאשר אחד מהם הוא פונקציה קבועה.

נניח הנחה מקילה כי הארגומנט השני לאופרטור הכפל יכול להיות רק אחד משתי האפשרויות שהעלינו בכדי לא להלאות במספר רב של בדיקות:

```
69 def __mul__(self, other):
70   if isinstance(other, Polynomial):
71     coefs = [0] * (self.deg + other.deg + 1)
72
73     for i, a in enumerate(self.coefs):
74       for j, b in enumerate(other.coefs):
75         coefs[i + j] += (a * b)
76   else:
77     coefs = (c * other for c in self.coefs)
78
79   return Polynomial(*coefs)
```

דוגמאות לפעולות חשבוניות על פולינומים:

```
>>> f = Polynomial(3, 1, 1)  # f(x) = x^2 + x + 3
>>> g = Polynomial(0, 1)     # g(x) = x
>>> (f + g)(5)
33
>>> (f – g)(5)
28
>>> (g * 3)(5)
15
```

פעולות חשבוניות נוספות שלא הוזכרו (רשימה חלקית):

- __div__ אופרטור החלוקה
- __mod__ אופרטור שארית החלוקה
- __pow__ אופרטור החזקה

הערה: לכל האופרטורים הבינארים שהוזכרו לעיל קיימים מקביליהם המורחבים (במקור: Enhanced) המשלבים פעולה והשמה יחד, כגון: =* ,=- ,=+ וכו'. המתודות המיוחדות עבור אופרטורים אלה ניתנות למימוש מחדש גם הן ושמותיהן דומים לשמות המתודות הרגילות מלבד זאת ששמן מתחיל ב-i, כגון: __imul__ ,__isub__ ,__iadd__ וכו'.

למרות חשיבותן מתודות נוספות אלו לא מומשו כחלק מהמחלקה Polynomial כיוון שהשמה למופע Polynomial קיים סותרת את ההנחה המקילה אותה הנחנו בפרק המבוא האומרת כי מופע Polynomial אינו בר שינוי לאחר אתחולו.

היפוך סימן

היפוך סימן הוא פעולה בסיסית ומובנת כאשר מדובר בקבועים מספריים ובכך השלכותיו על פולינום הן דומות: היפוך סימנם של כל מקדמי הפולינום. למעשה פעולה זו שקולה להכפלת הפולינום בקבוע המספרי 1- אך קיים לה סימון יחודי והוא אופרטור החיסור האונרי. נממש את האופרטור באמצעות המתודה המיוחדת __neg__:

```
80 def __neg__(self):
81   coefs = (-c for c in self.coefs)
82
83   return Polynomial(*coefs)
```

פעולות דומות על סימן שלא הוזכרו:

- __pos__ אופרטור החיוב האונרי
- __abs__ מימוש הפונקציה המובנית abs בהקשר המחלקה הנוכחית

גזירה

כידוע פונקציות פולינומיאליות ניתנות לגזירה. פעולת הגזירה דורשת הכפלת כל מקדם במעריך החזקה שלו ולאחר מכן הפחתת המעריך באחד. אופן מימוש המחלקה וקלות הפעולות על רשימות בשפת Python הופכים את הפעולה לפשוטה במיוחד. מכיוון שמיקום המקדם בסדרה מעיד על מעריך החזקה שלו, הזזת המקדמים "שמאלה", למקום נמוך יותר ברשימה, תהווה הפחתה מערכם של מעריכי החזקה כנדרש. על-ידי הזזה כזו ישמט המקדם השמאלי ביותר; השמטת מקדם זה כמוה כמחיקת המקדם החופשי בפעולת גזירה מתמטית.

מסיבה זו נשתמש באופרטור ההזזה שמאלה בכדי לממש את פעולת הגזירה; נניח כי האובייקט מימין לאופרטור הוא ערך מספרי המסמן את מספר פעולות הגזירה הנדרשות:

```
84 def __lshift__(self, n):
85   if n > self.deg:
86     coefs = (0,)
87   else:
88     coefs = self.coefs
89
90   for j in xrange(n):
91     coefs = tuple(c * i for i, c in enumerate(coefs[1:], 1))
92
93   return Polynomial(*coefs)
```

נשים לב כי במידה ומספר הגזירות המבוקש עולה על מעלת הפולינום, תוצאת הגזירה תהייה פונקצית האפס. ניתן לראות זאת גם באופן המימוש של המחלקה: הזזת כל המקדמים שמאלה עד אשר לא נשאר אף מקדם ברשימה מקביל לרשימת מקדמי אפס בלבד.

דוגמאות:

```
>>> f = Polynomial(2, 1, 4)    # f(x) = 4x^2 + x + 2
>>> f_tag = f << 1             # f'(x) = 8x + 1
>>> f_double_tag = f << 2      # f''(x) = x
>>> f_six_tags = f << 6        # f’6(x) = 0
```

פעולות נוספות על ביטים שלא הוזכרו:

- __rshift__ אופרטור הזזת ביטים ימינה
- __or__ אופרטור Bitwise OR
- __and__ אופרטור Bitwise AND
- __xor__ אופרטור Bitwise XOR
- __invert__ אופרטור אונרי Bitwise NOT

הערה: גם לאופרטורים הבינאריים לעיל קיימים אופרטורים מורחבים המשלבים השמה יחד עם הפעולה. אופרטורים מורחבים אלה מתחילים אף הם ב-i.

- סוף חלק שני -