מתודות מיוחדות ב-

- חלק ראשון -

אלי כהן-נחמיה

מבוא

כשפת תכנות מונחה עצמים, מאפשרת Python הגדרת פונקציונליות ייחודית או התנהגות מסויימת עבור מחלקה כאשר זו מופיעה לצד אופרטורים שונים או כארגומנט לפונקציות מיוחדות - באמצעות מתודות מוסכמות. שליטה במתודות אלה, אשר מתהדרות בצמד קווים תחתונים בשני צידי שמן, מקנה את היכולת להגדיר מחדש את התנהגותה של המחלקה כלפי האופרטורים השונים; למעשה ניתן לראות במתודות מיוחדות אלו מעין העמסת אופרטורים. שימוש מושכל בהן עשוי לשפר את איכות הקוד וקריאותו וכן לאפשר מיצוי טוב יותר של Python כשפת תכנות מונחה עצמים.

בכדי להמחיש את הדברים בצורה הטובה ביותר ניצור מחלקה חדשה אליה נוסיף פונקציונליות נדבך אחר נדבך. המחלקה אותה ניצור תייצג ערך מספרי שלם אך תאפשר בנוסף פעולות שונות על ייצוגו הבינארי. נגדיר את הדרישות למחלקה BitField:

- **אובייקט מספרי** תמיכה בכל הפעולות האריתמטיות התקפות בערכים מספריים
- **ניתן לכימות** החזרת מספר הביטים המינימלי לייצוג בינארי של הערך המספרי
- **איטרטיבי** יכולת לבצע איטרציה על הביטים המייצגים את הערך המספרי
- **ניתן לדגימה** יכולת להחזיר את ערכו של ביט או רצף ביטים במיקום נתון

הערות נוספות:

- למען הפשטות נתייחס לאובייקט BitField כמייצג ערך מספרי אשר אינו ניתן לשינוי לאחר אתחולו
- למען היעילות נמנע משימוש בפונקציה המובנית bin לצורך לבצע פעולות על ייצוגו הבינארי של הערך המספרי
- את קוד המקור המלא בליווי הערות ודוגמאות ניתן להוריד מכאן; הקוד תואם גרסת 2.7.3

תמיכה בפעולות אריתמטיות

בכדי לאפשר ביצוע פעולות אריתמטיות כגון חיבור, חיסור, כפל וחילוק על אובייקט BitField, תירש המחלקה את המחלקה int ואת תכונותיה; באופן הזה תקבל BitField את היכולת להשתתף בביטויים אריתמטיים. נפתח, אם כן, בהכרזה על המחלקה החדשה:

```
1 class BitField(int):
```

ירושה זו טומנת בחובה שלל תכונות אשר את חלקן נהייה מעוניינים להחליף במימוש הייחודי ל-BitField. נכון לעכשיו BitField הוא אובייקט מספרי לכל דבר ומתוקף כך ניתן לבצע עליו פעולות אריתמטיות.

בכדי לאתחל את מופעי המחלקה נשתמש במתודה המיוחדת __init__ המפורסמת אשר מופקדת על אתחול שדות האובייקט בזמן יצירתו. נאתחל את מופעי BitField כך שיאותחל גם אביו של BitField ואיתו ערכו המספרי:

```
2 def __init__(self, n):
3   int.__init__(n)
```

אורך האובייקט

כאשר חושבים על אובייקט אשר תפקידו הוא ייצוג בינארי של ערך מספרי, טבעי להניח כי בקשה לקבלת אורך האובייקט תניב את מספר הביטים הנדרשים לאותו ייצוג בינארי. מכיוון שאובייקט BitField אינו בר שינוי וערכו המספרי קבוע נוכל לחשב את "אורכו" כבר בזמן אתחול המופע. נוסיף את הקוד הרלוונטי למתודת האתחול:

```
 4 | length = 0
 5 |
 6 | while n >> length:
 7 |   length += 1
 8 |
 9 | self._length = length
```

Python הגדירה פונקציה אחידה לקבלת אורך אובייקטים בשפה; אורך אובייקט מתקבל על-ידי הפעלת len(obj) עבור אובייקט obj כלשהו. למעשה, כאשר מופעלת הפונקציה len על אובייקט, נקראת המתודה המיוחדת __len__ של אותו האובייקט, כלומר ניתן לומר כי len(obj) ↔ obj.__len__ :

נרצה, אם כן, להחליף את מימוש המתודה המיוחדת __len__ אותה ירשנו מ int-באופן הבא:

```
10 | def __len__(self):
11 |   return self._length
```

כעת נוכל לקבל את אורכו של אובייקט BitField :

```
>>> bf = BitField(0xd)
>>> print '%d' % len(bf)
4
```

המתודה המיוחדת __len__ משמשת גם למטרות נוספות כגון קביעת ערכו של המופע בהקשר הבולאני[1]. כאשר מופיעה התייחסות לאובייקט כביטוי בוליאני או בהפעלת הפונקציה bool(obj) על אובייקט כלשהו, פונקצית __len__ משמשת לקביעת ערכו הבוליאני של האובייקט: אם הערך המוחזר ממנה הוא 0, אזי האובייקט נחשב כ- False. אם הערך אינו 0, האובייקט נחשב כ-True.

דוגמאות מפורסמות לכך הן מחרוזות ורשימות: בהקשר הבוליאני מחרוזת ריקה או רשימה ריקה נחשבות ל-False ומחרוזת שאינה ריקה או רשימה שאינה ריקה נחשבות ל-True.

[1] קביעת ערכו הבוליאני של האובייקט תעשה באמצעות __len__ רק אם לא הוגדרה המתודה המיוחדת __nonzero__ עבור אותה מחלקה. מתודה זו, אם הוגדרה עבור אותו אובייקט, מקבלת קדימות על פני __len__ במקרים האלה.

איטרציה

אובייקטים המאפשרים איטרציה הם יסוד מרכזי בשפת Python: רשימות (Lists), סדרים (Tuples) ומחרוזות הן רק דוגמה חלקית לשפע האובייקטים המנצלים את כוחה של Python באיטרציות וב-List Comprehensions. בכדי לאפשר מעבר איטרטיבי על אובייקט יש לממש את המתודה המיוחדת __iter__ המחזירה איטרטור. איטרטור הינו אובייקט המממש ממשק מסויים המאפשר את קבלת האיבר הבא בכל איטרציה וכן אינדיקציה לסיום המעבר האיטרטיבי.

לצורך פישוט הדברים נעשה כאן שימוש בגנרטור (Generator) אשר מממש את הממשק הנדרש מאיטרטור והגדרתו קצרה יותר. לגנרטור יתרונות רבים על פני רשימה רגילה אך גם לא מעט חסרונות; דוגמאות מפורסמות לרשימה וגנרטור הן range ו-xrange.

נמקם את קוד הגנרטור בפונקציה נפרדת ונגדיר אותה כפונקציה סטטית; פונקציה סטטית אינה מקבלת את self כפרמטר:

```
12 @staticmethod
13 def __bits_gen__(n):
14   while n:
15     yield (n & 1)
16     n = n >> 1
```

קוד הגנרטור פשוט אך מבלבל מעט כיוון שהוא טומן בחובו פעולות סמויות: לולאת ה-while מייצגת את רצף האיטרציות או את המעבר האיטרטיבי בשלמותו; כאשר יפוג תנאי הלולאה אשר יביא לסיומה, תתרחש "שגיאה" (Exception) מיוחדת שתאותת כי הסתיים המעבר. בכל איטרציה של לולאת ה-while מתבצע yield אשר תפקידו לספק את הערך הנוכחי לאיטרציה.

בקוד שלפנינו האיטרציה מתקדמת מהביט הנמוך ביותר (Least Significant Bit) אל הביט הגבוה ביותר (Most Significant Bit), דבר אשר יגרום לביטים להופיע ב-"סדר הפוך" אם נדפיס אותם בזה אחר זה. ניתן לראות כי בכל איטרציה אנו מספקים את הביט הימני ביותר ולאחר מכן דוחפים את ערכו של n ימינה במקום אחד.

מימוש המתודה המיוחדת __iter__ יהיה פשוט ויחזיר את הגנרטור המוגדר למעלה:

```
17 def __iter__(self):
18   return BitField.__bits_gen__(self)
```

כעת נוכל לבצע מעבר איטרטיבי על אובייקט BitField:

```
>>> bf = BitField(0xd)
>>> [bit for bit in bf]
[1 ,1 ,0 ,1]
>>>
>>> options = BitField(0x42)
>>> for i, b in enumerate(options):
...   print 'option bit %d is %s' % (i, ('clear', 'set')[b])
...
option bit 0 is clear
option bit 1 is set
option bit 2 is clear
option bit 3 is clear
option bit 4 is clear
option bit 5 is clear
option bit 6 is set
```

ערך במיקום או בטווח נתון

גולת הכותרת של אובייקט BitField היא היכולת להחזיר את ערכו של ביט במקום מסויים או את ערכם של סדרת ביטים בטווח מסויים. בעולם האלקטרוניקה מקובל לסמן זאת באמצעות סוגריים מרובעים, לדוגמה: bf[0:3]

השימוש בנוטציה שהוזכרה מקובל ב-Python ומתאפשר על אובייקטים המגדירים את המתודה המיוחדת __getitem__. המתודה מקבלת ארגומנט המייצג את תוכנם של הסוגריים; תוכן זה יכול להיות ערך מספרי או טווח (במקור "חתיכה", Slice) שייצג טווח ביטים. אובייקטים אחרים, כגון מילון (Dictionary), אף מאפשרים העברת מחרוזת כארגומנט. נשים לב כי במקרה דנן ערך מספרי שקול לטווח שנקודות ההתחלה והסיום שלו שוות.

נסביר את מימוש המתודה __getitem__ באמצעות מספר שלבים:

1. אם ארגומנט המפתח הוא ערך מספרי, כלומר נוטציה מסוג bf[x], נמיר אותו לטווח שערך ההתחלה וערך הסיום בו זהים.
2. אם ארגומנט המפתח בשלב זה אינו טווח, ניצור שגיאה
3. כאשר ארגומנט המפתח בידינו הוא טווח, נחשב את מסכת הביטים המתאימה בכדי לחלץ את הביטים בטווח הרצוי:

a. מכיוון שהביטים מימין לנקודת ההתחלה אינם דרושים לנו כעת, נוכל להקל על עצמינו ולהביא את הביט בנקודת ההתחלה אל המיקום הנמוך ביותר באמצעות דחיפת ערכו של האובייקט.
b. נחשב את מספר הביטים הרצוי לנו באמצעות נקודת ההתחלה ונקודת הסיום
c. נייצר מסכת ביטים באורך הרצוי
d. נשלוף את הביטים הרצויים מתוך ערכו של האובייקט

הקוד של __getitem__ יראה כך:

```
19 def __getitem__(self, key):
20   if isinstance(key, int):
21     key = slice(key, key)
22
23   if not isinstance(key, slice):
24     raise TypeError('bitfield indices must be integers')
25
26   shifted = self >> key.start
27   width   = key.stop – key.start +  1
28   mask    = (1 << width) - 1
29
30   return shifted & mask
```

כעת נוכל לקבל את ערכם של ביטים במקום או בטווח נתונים:

```
bf = BitField(0xd)
bf[0:2]
5
if bf[3]:
...  print 'bit 3 is set'
...
bit 3 is set
```

- סוף חלק ראשון -