נתון מטה מימוש של מחלקה בשם MinTree המייצגת עץ כללי, עם ערכים מספריים. עלה מיוצג כמופע של MinTree ללא בנים (None ב-branches) וערך מספרי ב-entry. קדקוד פנימי זה מופע של MinTree עם רשימת הבנים (list) ב-branches וערך מספרי ב-entry שהוא הערך המינימאלי בין הערכי entry של הבנים.

א. (5 נק') השלם פונקציה __repr__ בהגדרה של מחלקה:

```
class MinTree():
    def __init__(self, entry, branches = None):
        self.entry = entry
        self.branches = branches
    def __repr__(self):
```

```
def __repr__(self):
    if self.branches:
        return 'MinTree({0},{1})'.format(repr(self.entry),
                repr(list(self.branches)))
    return 'MinTree({0})'.format(repr(self.entry))
```

ב. (10 נק') השלם פונקציה *build_min_tree*, שבהנתן tuple המייצג עץ כללי עם מספרים בתור עלים, בונה ומחזירה מופע של *MinTree* עבור אותו עץ. הפונקציה חייבת להיות רקורסיבית!

```
def build_min_tree(tree):
```

```
def build_min_tree(tree):
    if type(tree) != tuple:
        return MinTree(tree)
    sons = list(build_min_tree(i) for i in tree)
    return MinTree(min(x.entry for x in sons),sons)
```

א. (8 נק') השלם את הפונקציה ***sum_filter_tree*** שבהנתן פונקציה *f* (בוליאנית של ארגומנט אחד) ועץ *tree* (מיוצג כ-tuple) מחזירה סכום של אלמנטים שלו המקיימים את התנאי של *f*. הפונקציה צריכה להיות רקורסיבית. אין להשתמש בפונקציה *filter* מובנת!

```
def sum_filter_tree(tree, f):
```

```
def sum_filter_tree(tree,f):
    if type(tree) != tuple:
        if f(tree):
            return tree
        else: return 0
    return sum(sum_filter_tree(i,f) for i in tree)
```

נתון מטה מימוש של מחלקה בשם Tree המייצגת עץ בינארי. לכל צומת בעץ יש מצביעים לשני בנים. העלים הם יכולים להיות מכל טיפוס שונה מ-Tree.

א. (5 נק') השלם פונקציה __repr__ בהגדרה של מחלקה:

```
class Tree():
    def __init__(self, left, right):
        self.left = left
        self.right = right
def __repr__(self):
    return 'Tree({0},{1})'.format(repr(self.left), repr(self.right))
```

ב. (10 נק') השלם פונקציה *mirror*, שבהנתן עץ בינארי, מחזירה עץ חדש (גם מופע של Tree) שבו סדר של בנים הוא הפוך (לפי "השתקפות ראי"). הפונקציה חייבת להיות רקורסיבית!

```
def mirror(tree):
```

```
def mirror(tree):
    if type(tree) != Tree:
        return tree
    return Tree(mirror(tree.right), mirror(tree.left))
```

א. (8 נק') יש להשלים את הפונקציה ***get_depth*** המחשבת עומק של העץ-ארגומנט, מיוצג כ-tuple. הפונקציה צריכה להיות רקורסיבית.

```
def get_depth(tree):
```

```
def get_depth(tree):
    if type(tree) != tuple:
        return 1
    return 1 + max(map(get_depth,tree))
```

א. (17 נק') השלם את הפונקציה *map_tree_reverse* כפונקציה רקורסיבית שבהינתן עץ *t* ופונקציה (של ארגומנט אחד) *f* מחזירה עץ חדש שמתקבל מהפעלת פונקציה על עלי *t* תוך הפיכת סדר של בנים בכל צומת בעץ. עץ מיוצג כ-tuple מקונן. יש לטפל בכל תתי-עץ באופן רקורסיבי.

```
def map_tree_reverse(t, f):
```

```
def map_tree_reverse(t, f):
    if type(t) != tuple:
        return f(t)
    return tuple(map_tree_reverse(i, f) for i in t[::-1])
```

נתון מטה מימוש של מחלקה בשם Tree המייצגת עץ כללי. לכל צומת בעץ יש ערך פנימי (entry) רשימת בנים (nodes). העלים הם מופעים של Tree ללא רשימת ילדים.

א. (10 נק') השלם פונקציה __repr__ בהגדרה של מחלקה:

```
class Tree():
    def __init__(self, entry, nodes=None):
        self.entry = entry
        self.nodes = nodes
    def __repr__(self):
```

```
def __repr__(self):
    if self.nodes:
        return 'Tree({0})'.format(repr(self.entry))
    return 'Tree({0},{1})'.format(repr(self.entry),repr(self.nodes))
```

ב. (15 נק') השלם פונקציה ***map_transform***, שבהנתן עץ (tree) מיוצג כ-tuple ופונקציה של 2 ארגומנטים (g), מחזירה עץ חדש (מופע של **Tree**) בעל מבנה שקול ל-tree עם entry שווה לתוצאת הפעלה של פונקציה g על ערכי entry של הילדים שלו. הפונקציה חייבת להיות רקורסיבית!

הערה חשובה: במידה והפונקציה לא מקבלת את הארגומנט השני (g) היא אמורה לחשב סכום לפי ברירת מחדל (ראה דוגמת הרצה).

```
def map_transform(tree, g = _____):
```

```
def map_transform(tree,g=lambda x,y: x+y):
    if type(tree) != tuple:
        return Tree(tree)
    nodes = list(map_transform(branch,g) for branch in tree)
    return Tree(reduce(g, (n.entry for n in nodes)),nodes)
```

נתון מטה מימוש של מחלקה בשם Expr המייצגת ביטוי אריתמטי של 2 ארגומנטים, כאשר ארגומנטים יכולים להיות ביטויים. העלים יכולים להיות מכל טיפוס שונה מ-Expr.

א. (10 נק') השלם פונקציה __repr__ בהגדרה של מחלקה:

```
class Expr():
    def __init__(self, entry, left, right):
        self.entry = entry
        self.left = left
        self.right = right
def __repr__(self):
    return 'Expr({0},{1},{2})'.format(repr(self.entry),
                                      repr(self.left),repr(self.right))
```

ב. (10 נק') השלם פונקציה *build_expr_tree*, שבהנתן tuple שבנוי מ-3 אלמנטים (שם אופרטור במקום ראשון וארגומנטים במקום שני ושלישי) ומייצג ביטוי אריתמטי, בונה ומחזירה מופע של *Expr* עבור הביטוי. הפונקציה חייבת להיות רקורסיבית!

```
def build_expr_tree(tree):
```

```
def build_expr_tree(tree):
    if type(tree) != tuple:
        return tree
    return Expr(tree[0],build_expr_tree(tree[1]),build_expr_tree(tree[2]))
```

א. (8 נק') יש להשלים את הפונקציה *map_extend* שבהנתן פונקציה f (של ארגומנט אחד) ושתי רשימות (base ו-ext) מרחיבה את base ע"י אלמנטים חדשים שהתקבלו מהפעלת פונקציה f על אלמנטים של רשימה ext. הפונקציה מחזירה את הרשימה המורחבת. הפונקציה צריכה להיות רקורסיבית. אין להשתמש בפונקציה *map* מובנת!

```
def map_extend(f, base, ext):
```

```
def map_extend(f, base, ext):
    if not ext:
        return base
    return list(map_extend(f, base + [f(ext[0])], ext[1:]))
```

ב. (10 נק') נתון מטה מימוש של מחלקה בשם Tree המייצגת עץ. לכל צומת בעץ הזה יש מצביעים לבנים ועומק של תת-עץ שמתחיל בצומת הזה:

```
class Tree():
    def __init__(self, depth, nodes=None):
        self.depth = depth
        self.nodes = nodes
    def __repr__(self):
        if self.nodes: return 'Tree({0},{1})'.format(self.depth, repr(self.nodes))
        return 'Tree(1)'
```

השלם פונקציה ***transform***, שבהנתן עץ המיוצג ע"tuple, בונה מופע של Tree

```
def transform(tree):
```

```
def transform(tree):
    if type(tree) != tuple:
        return Tree(1)
    nodes = list(map(transform, tree))
    return Tree(1+ max(n.depth for n in nodes), nodes)
```