## Simpler

כפי שראינו בשיעור, תוכנית האימות כשלעצמה מיועדת להתמודד עם syntax בסיסי. עם זאת, בעולם התוכנה, התוכניות נכתבות ע"י שימוש בsyntax מורכב מטעמי נוחות וחסכון.

לכן, בהינתן תוכנית, מטרתינו הראשונית היא לפשט אותה כך שהפונקציונאליות נותרת זהה, אך המאמת יכול להתמודד עמה.

### טיפול בIMPORTים

על פי עקרונות המודולוזיציה, מבנה סטנדרטי של ספריית פיית'ון הינו מהצורה:

Main\_project\_directory/

\_\_init\_\_.py

Secondary\_section\_1/

\_\_init\_\_.py

Code1.py

Code2.py

…

באופן טבעי, כל מודול מכיר את עצמו, ועל מנת שיכיר מודולים אחרים נעזרים בimport. ישנן שתי צורות לבצע את הפעולה:

From a import b

Import c

a יכול להיות ספריה או מודול, b יכול להיות מודול או איבר מוגדר (מחלקה, פונקציה, משתנה גלובאלי) בתוך a, c חייב להיות ספריה.

המטרה היא להפוך ספריה המורכבת ממספר מודולים, וביניהם ביטוי import למודול בודד.

ראשית, נעבור על המודול ונחליף ביטוי import z לביטויים מהצורה from x import y על פי החוקים הבאים:

* אם z הוא ספריה אז הביטוי מייבא את הקובץ \_\_init\_\_.py ולכן הfrom הרצוי הוא from z import \_\_init\_\_
* אם z הוא מהצורה a.b אנחנו בודקים האם יש ספריה בשם a שבתוכה קובץ בשם b.py. במידה וכן, אנחנו הופכים את הביטוי לfrom a import b
* אחרת – b הוא יבטוי מוגדר בתוך a ואנחנו הופכים את הביטוי לfrom a import \*

לאחר מכן, אנחנו מתמודדים עם syntax חסכוני של פיית'ון; הביטוי from **.**a import b מציין "מהספריה הנוכית שבה אנו נמצאים יבא את a.b". אנחנו הופכים את האימרה הזאת להיות אקספליסית.

לאחר ששינינו את הsyntax לכזה שנוח מבחינת הההתמודות שלנו, אנו בונים גרף תלויות. המטרה היא להבין את מערכת התלויות במודול ואז לפלוט לקובץ בודד את כל הקוד. אנו עושים זאת באמצעות בניית DAG שבו כל מודול מצביע למודול שהוא תלוי בו (ההנחה שהיא שניתן להריץ את התוכנית, ולכן אין import מעגלי).

לאחר שבנינו את הDAG אנו פולטים לקובץ אחוד את העלים, ואז עולים במעלה העץ עד אשר עוברים על כל המודולים.

לסיכום, אנחנו מסירים מהקוד את כל האיזכורים לחבילות הללו (לדוגמא כשעושים import a אז כשנרצה להשתמש בפ' מתוך a הסינטקס הנכון יהיה a.foo(). כיוון שהעתקנו את foo לקובץ אותו אנו מריצים הקריאה הנכונה צריכה להיות foo().

### פישוט הקוד

הפישוט עובד בצורה איטירטיבית כך שבכל איטרציה מתבצע תיקון בודד. כל עוד ביצענו פישוט באיטרציה הנוכחית אנחנו נמשיך לפישוט נוסף, כך שהקוד מבצע פישוטים מורכבים לפישוטים בסיסיים שהתבצעו. הקוד בנוי כך שהפישוטים מתבצעים לפי סדר שיוצר קוד נכון וקריא.

הפישוטים האפשריים:

* השמת tuple: בפיית'ון ניתן לבצע השמה לtuple, כך שהאיבר הראשון בחישוב מתמפה לאיבר הראשון, האיבר השני לשני וכו'. הפישוט מוציא את תוצאת החישוב למשתנה טמפלייטי ואז מתבצעת השמה איבר איבר.
* השמה לאיבר מורכב: איבר מורכב משמעותו פניה לאטריביוט פנימי (אטריביוט של אטריביוט, לדוגמא a.b.c), במצב כזה אנחנו עושים מוציאים את a.b למשתנה זמני, ואז מבצעים השמה לtmp.c. הסיבה שבגללה אנחנו לא מפשטים כך שתמיד תתבצע השמה למשתנה בודד היא מכיוון שניתן לייצר אטריביוטים בצורה דינמית, לדוגמא השמה a.x היא חוקית גם אם x אינו קיים, אך לא ניתן לבצע השמה tmp = a.x. כמו כן, על מנת לייצר את המשתנה a.x.y, על x להיות קיים.
* פניה לאטריביוטים: פניה לאטריביוט תמיד תתבצע ממשתנה ולא מאטריביוט. כלומר פניה של a.b.c אינה חוקית, אך פניה מהסוג tmp = a.b , tmp.c הינה חוקית. נשים לב שהדבר נכון מבחינה סמנטית כי אם a.b הוא אוביקט אזי לאחר השמה למשתנה זמני שניהם יפנו לאותו מקום בזיכרון, ואם הוא קבוע (לדוגמא מספר או מחרוזת) אזי שניהם מצביעים לאותו איבר בבריכת הקבועים.
* פישוט אריתמטי: ביטויים אריתמטיים מורכבים יהפכו לביטויים בינאריים תוך שימוש במשתנים זמניים
* CALL ARGS: הפעלת פונקציה תיעשה רק מפרמטרים שהם משתנים
* ביצוע אופרטור והשמה: פעולות כמו +=, -= וכו' מתורגמות לשתי שורות קוד – השמה למשתנה זמני של תוצאת החישוב והשמת המשתנה למטרה.
* איטרציה בלולאות For:הוצאת האיבר שעליו מתבצעת האיטרציה למשתנה.
* אופרטורים בינאריים: אופרטורים בינאריים הם בעצם פונקציה שקוצרה ע"י סוכר תחבירי. במילים אחרות, הsyntax a OP b (OP הוא פעולה אריתמטית) הוא סוכר תחבירי להפעלת הפונקציה בצורה הבאה a.\_\_OP\_\_(b).
* ערך return: פישוט הReturn Value כך שתמיד יהיה משתנה.
* פישוט list, dict, tuple*:* כל אחד מאבריהם הינו משתנה.