**האוניברסיטה הפתוחה**

**המחלקה למתמטיקה ולמדעי המחשב**

|  |
| --- |
|  |
| **טיפול בספריות שהוצאו משימוש בפייתון** |
| תואר שני מדעי המחשב  על-ידי  **יוני שפונד**  העבודה הוכנה בהדרכת **פרופ׳ שמואל טישברוביץ**  **2023** |

תוכן עניינים

|  |  |  |
| --- | --- | --- |
| ‏1 | הקדמה | 4 |
| ‏1.1 | פייתון – Python | 5 |
| ‏1.2 | ממשק תכנות יישומים - API | 5 |
| ‏1.3 | תכנות דינאמי מול סטטי | 5 |
| ‏2 | שאלות מחקר | 6 |
| ‏2.1 | איך ספריות מבצעות הוצאה משימוש? | 6 |
| ‏2.2 | איזה השפעות יכולות להיות להוצאה משימוש? | 8 |
| ‏2.3 | מהו אופן השימוש? | 9 |
| ‏2.4 | מהו אופן התיעוד? | 11 |
| ‏2.5 | איך הקהילה מגיבה? | 13 |
| ‏2.6 | מהם השיטות לפתרון הבעיה? | 15 |
| ‏2.6.1 | החלפה לממשק תכנות יישומים אחר | 15 |
| ‏2.6.2 | לתפוס שגיאות מסוגים שונים | 15 |
| ‏2.6.3 | הוספה או שינוי מיקום או שם של פרמטרים | 16 |
| ‏2.6.4 | בדיקת גרסא של ספריה | 16 |
| ‏3 | החלטות תיכון בפיתוח הכלי | 18 |
| ‏3.1 | בסיס | 18 |
| ‏3.2 | לוגים | 18 |
| ‏3.3 | פרמטרים לפונקציה | 19 |
| ‏3.4 | איסוף מידע | 20 |
| ‏3.5 | ניתוח | 23 |
| ‏3.5.1 | בדיקת פרמטרים | 24 |
| ‏3.5.2 | בדיקת הזהרות | 26 |
| ‏4 | סיכום | 28 |
| ‏5 | ביביליוגרפיה | 30 |
|  |  |  |

רשימת איורים

[איור 1: תבניות שינויים של ממשקי תכנות יישומים [3] 6](#_Toc163757556)

[איור 2: שינויים של חתימות לאורך גרסאות שונות ב-TensorFlow [3] 6](#_Toc163757557)

[איור 3: הוצאה משימוש של חתימה של פונקציה [10] 7](#_Toc163757558)

[איור 4: קורלציה בין סימפטום לסיבה [3] 8](#_Toc163757559)

[איור 5: שימוש בממשקי תכנות יישומים שהוצאו משימוש בפרויקטי משתמשים [10] 10](#_Toc163757560)

[איור 6: ממשקי תכנות יישומים שהוצאו משימוש ללא תיעוד 11](#_Toc163757561)

[איור 7: ממשקי תכנות יישומים שהוצאו משימוש ללא אלטרנטיבות 11](#_Toc163757562)

[איור 8: דוגמא - ממשק תכנות יישומים מוגדר בצורה לא ברורה [14] 12](#_Toc163757563)

[איור 9: התפלגות של פוסטים שקשורים לממשקי תכנות יישומים שהוצאו משימוש לפי שפת תכנות 13](#_Toc163757564)

[איור 10: התפלגות קוטביות של מפתחים בקרב חמשת השפות הפופולריות ביותר 14](#_Toc163757565)

[איור 11: דוגמא - מפתח שמתלונן על צורת התיעוד [19] 14](#_Toc163757566)

[איור 12: תיקון בעיה על ידי החלפת ממשק 15](#_Toc163757567)

[איור 13: תפיסת שגיאות 15](#_Toc163757568)

[איור 14: שינוי לגישה של פרמטרים לפי keyword 16](#_Toc163757569)

[איור 15: שינוי לגישה של פרמטרים לפי מיקום 16](#_Toc163757570)

[איור 16: בדיקת גרסאת הספריה 16](#_Toc163757571)

[איור 17: תרשים זרימה של קריאת קבצי הפרויקט 19](#_Toc163757572)

[איור 18: תרשים זרימה - מעבר על קריאות לפונקציות 20](#_Toc163757573)

[איור 19: תרשים זרימה של ניתוח הממשקים 22](#_Toc163757574)

[איור 20: תרשים זרימה של בדיקת הפרמטרים 24](#_Toc163757575)

[איור 21: תרשים זרימה של בדיקת הזהרות 26](#_Toc163757576)

רשימת טבלאות

[טבלה 1: ספריות למחקר 8](#_Toc163757291)

[טבלה 2: כמויות/אחוזים של שגיאות 9](#_Toc163757292)

# הקדמה

אחת שפות התכנות הכי פופולריות בשנים האחרונות היא Python [1], כאשר נעשה שימוש בספריות עבור מכונות לומדות ומערכות לחישובים מדעיים. השפה צברה את הפופולריות העצומה שלה עקב המבחר הגדול שלה של ממשקי תכנות יישומים (API), אשר חלק מאותם ממשקים נועדו לטובת מכונות לומדות וגם לחישובים מדעיים. ממשקים אלה יוצאים מכלל שימוש בגלל עדכונים, עקב שיפורי תוכנה ותיקוני באגים, בדיוק כמו בשפות תוכנה אחרות. רוב שינויי API כוללים בתוכם הזזה או שינוי פונקציות ו/או שדות [2].

שינוים אלה גורמים למפתחים לחוסר מוטיבציה להשתמש בפייתון עבור פיתוחים נוספים, וגם לבעיות תאימות לגרסאות אחרונות [3]. על מנת לאפשר למפתחים להסתגל לשינויים הללו בצורה חלקה, רצוי להשתמש בשיטת *“deprecate-replace-remove”* [4]. בשיטה הזו, ממשקי תכנות יישומים שלא נתמכים יותר, יוגדרו תחילה כממשקים שהוצאו משימוש, ולאחר מכן, מתווספות הודעות, כחלק מהקוד, אשר מציעות אלטרנטיבות על מנת לעזור למפתחים עם המעבר לממשקים החדשים [5].

בכל סיטואציה שבה מפתח צריך לזהות ולהחליף ממשקי תכנות יישומים (API), מדובר במשימה מונוטונית שצורכת זמן יקר, כאשר בכל פרויקט קיים מספר רב של קריאות לממשקים (API). יתרה מזאת, התיעוד של ממשקים (במסמכים) גורם למשימות הללו להפוך להיות הרבה יותר מסובכות ומאתגרות. הנושא הזה, לאורך שנים רבות היווה אתגר משמעותי עבור מפתחים.

עם התקדמות הגרסאות, ממשקי תכנות היישומים, שהוצאו משימוש, יימחקו. התהליך הזה לא תמיד ממומש בפועל (כמו שניתן לראות במחקרים שונים - [6], [7]).

חוקרים הציגו הצעות לטכניקות שמעדכנות בצורה אוטומטית את ממשקי תכנות היישומים שהוצאו משימוש - [8], [9]. אולם, רובם נועדו עבור שפות תכנות סטטיות כמו Java, C#, כאשר פייתון היא שפת תכנות דינאמית, אשר מציעה תבניות מתפתחות שונות של API בהשוואה ל-Java [3]. זה מה שנותן מוטיבציה לצורך בטכניקות וגם כלים חדשים על מנת לזהות ממשקי תכנות יישומים שהוצאו משימוש.

ממשקי תכנות יישומים שהוצאו משימוש, בספריות פייתון, לרוב מוכרזים בשלושה אופנים [10]:

* Decorator
* Hard-coded Warning
* Comment

נמצא כי מפתחים שמתחזקים את הספריות לרוב משתמשים ביותר מאחת מהשיטות הנ"ל, מה שגורם לחוסר אחידות במימוש של הספריות [10]. בשביל להימנע משימוש של ממשקי תכנות יישומים שהוצאו משימוש, במהלך פיתוח תוכנה, מפתחים צריכים להיות מודעים לממשקים הללו בתוך הפרויקט שלהם.

לכן, בהתחשב בעליית הפופולריות של פייתון, ומספר ממשקי תכנות היישומים שהוצאו משימוש שנמצאים בפרויקטי פייתון, נחוץ כלי שבעזרתו יהיה ניתן להימנע מהבעיות הנ״ל ובנוסף ניתן לחסוך המון שעות עבודה של מפתחים.

בעבודה זו, נבנה כלי אשר מציע פתרון עבור כלל המפתחים שנתקלים בבעיות שתוארו מעלה. כלי זה מנתח פרויקטים בפייתון, ומייצר סיכום של כל המקומות בהם הוא מזהה פוטנציאל לשימוש לא נכון בממשק תכנות יישומים או לחלופין מזהה ממשק בעל הזהרה מכוונת של הוצאה משימוש.

## פייתון – Python

פייתון היא שפה דינאמית, קוד פתוח (open-source) חינמי, ומפורשת (interpreted), אשר בדרך כלל משומשת לטובת בניית תוכנות, להפוך משימות לאוטומטיות, ניתוח נתונים, וכו׳. פייתון היא שפה לשימוש כללי, הכוונה היא לכך שניתן להשתמש בה לטובת מגוון רחב מאוד של תוכנות והיא לא ממוקדת לפתרון בעיות ספציפיות. הגיוון והקלות שלה עבור מפתחים צעירים, הפכו אותה לאחת השפות הכי פופולריות כיום [1]. בנוסף, פייתון תומכת בתכנות מונחה עצמים וגם תכנות פרוצדורלי.

שפות תכנות דינאמי הן קבוצה של שפות *High-Level* אשר, בזמן ריצה, מבצעות הרבה פעולות "רגילות" (כגון, הרחבות של אוביקטים והגדרות) ששפות תכנות סטטיות מבעות בזמן קומפילציה. פייתון היא שפה דינאמית פופולרית. תוכניות בפייתון מורצות על ידי מפרש (interpreter), כאשר הוא מתרגם את קבצי קוד עם סיומת *py* לקבצי *pyc* אשר מכילים *byte-code*. מסיבה זו, בעיות שנובעות מהפרה של טיפוסי משתנים, קריאה לממשקי תכנות יישומים חסרים וגם שגיאות *syntax*, לא יהיו נראים לעין עד לזמן הריצה.

העברת משתנים לפונקציות בפייתון הוא דבר מאוד גמיש. כאשר קוראים לפונקציה, הארגומנטים שלה יכולים לעבור או לפי מיקום בהגדרה של הפונקציה, או לפי השם של הארגומנט (*keyword*) [10]. בנוסף, כל ארגומנט יכול להיות מוגדר עם ערך ברירת מחדל, מה שלא מחייב העברת ערך כלשהו לארגומנט הזה בעת הקריאה לפונקציה.

## ממשק תכנות יישומים - API

ממשק תכנות יישומים הוא ערכה של ספריות קוד, פקודות, פונקציות, חוקים ופרוצדורות מוכנות, בהן המתכנתים יכולים לעשות שימוש פשוט, וזאת מבלי להידרש לכתוב אותן בעצמם, תוך כדי שימוש בו לפרויקט שלהם.

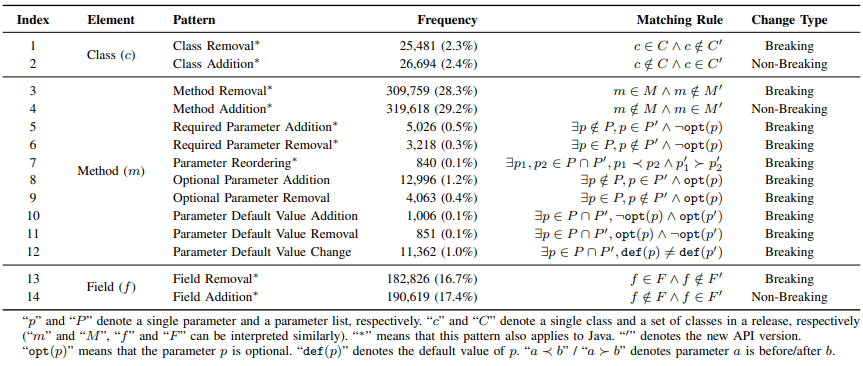
## תכנות דינאמי מול סטטי

* שפות תכנות סטטיות מבצעות בדיקות טיפוסי משתנים בזמן ריצת קומפילציה, כאשר שפות תכנות דינאמיות מבצעות בדיקות טיפוסי משתנים בזמן ריצה.
* שפות תכנות דינאמיות (כמו - Lisp, Perl, Python, Ruby) מתוכננות לייעל את עבודת המפתח ולבצע אופטימיזציות לקוד כך שאותה פונקציונליות תמומש בפחות שורות קוד. שפות תכנות סטטיות (כמו - Java, C, C++) מתוכננות לייעל את עבודת החומרה על מנת שהקוד ירוץ כמה שיותר מהר.
* שפות סטטיות דורשות הצהרה של טיפוסי המשתנים לפני שמשתמשים בהם, בזמן שבשפות דינאמיות אין צורך בהצהרה.

# שאלות מחקר

## איך ספריות מבצעות הוצאה משימוש?

כאשר מנתחים מאות גרסאות של ספריות, כל גירסא יכולה להוביל לממשקי תכנות יישומים אחרים. להתחלה, נלקחו תשעה חוקי התאמה בסיסיים של ממשקי תכנות יישומים [3], ורוב השינויים תורגמו לתבניות הבאות:

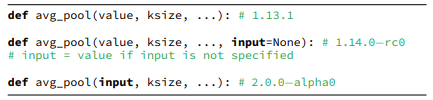


איור 1: תבניות שינויים של ממשקי תכנות יישומים [3]

בטבלה שבאיור 1, ניתן לראות רשימה של תבניות אשר מחולקות לשלוש קטגוריות: Class, Method, Field. התבניות מגדירות את סוג השינוי שנעשה לממשק (לדוגמא – מחיקת מחלקה, מחיקת פונקציה, הוספה או מחיקה של פרמטר מחתימה של פונקציה, וכו'). לאחר מכן, העמודה Frequency מגדירה את כמות המקרים שבהם התבנים קרתה. העמודה Matching Rule מגדירה את החוקיות של אותה התבנית. ובנוסף, עמודה Change Type מגדירה האם התבנית המדוברת שוברת (Breaking) או לא את הפרויקט בזמן ריצה אם תבנית כזו מתרחשת. בסך הכל נמצאו ארבע עשרה תבניות כאשר רק חמישה מהם לא נמצאו ב-Java.

ספריות פייתון מוצאות משימוש מסיבות שונות, אשר מסתכמות לשלוש קטגוריות [10]:

1. **מחלקה** או פונקציה שנמצאות בשימוש כאשר אינן נמצאות בספריה יותר, לדוגמא:

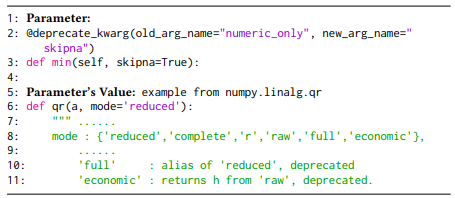


איור 2: שינויים של חתימות לאורך גרסאות שונות ב-TensorFlow [3]

מחלקה של פייתון שהוצאה משימוש נמצאת בקבוצה ביחד עם פונקציות מאחר והיא דומה מאוד לקריאה לפונקציה (כמו קריאה לבנאי (constructor) ב-Java). פונקציות, הן הגורם הפופולרי ביותר להוצאה משימוש של ממשקי תכנות יישומים על ידי מפתחי פייתון, בדומה להוצאה משימוש בשפות תכנות אחרות.

מחלקות ופונקציות נמצאות בקטגוריה אחת, מאחר ובפייתון ההתייחסות למחלקות היא זהה כמו לפונקציות (כמו בנאים ב-Java).

1. שינוי **חתימה של פונקציה**: מיקום או שם של פרמטר, אשר מוגדרים בממשק תכנות יישומים, יכולים לעבור הוצאה משימוש בפייתון. לדוגמא, הפרמטר “numeric\_only” מהפונקציה “pandas.DataFrame.min” כבר לא בשימוש, והפרמטר החדש נקרא “skipna”.



איור 3: הוצאה משימוש של חתימה של פונקציה [10]

שינויים בפרמטרים כמו באיור 2 אינם מתקיימים, בזמן שב-Java כן מתקיימים. שינויים כאלו קורים בפייתון מסיבה אחת ויחידה, פייתון תומכת בערכי ברירת מחדל לפרמטרים בהגדרות של פונקציות.

1. **ערך של פרמטר**: קטגוריה זו מתיחסת למקרים כאשר פרמטר של ממשק תכנות יישומים לא תומך יותר בערכים מסוימים עבור פרמטרים. לדוגמא, באיור 3, שורות 10-11, ערכים כמו “full” ו-“economic” כבר לא בשימוש עבור הפונקציה “qr”.

הוספה או מחיקה של ערך ברירת מחדל יגרום לפרמטר להיות חובה או בחירה לפונקציה.

מספר המקרים שבהם הוצאה משימוש של ממשקי תכנות יישומים בגלל ערך של פרמטר הוא נמוך מאוד. למרות זאת, ממשקי תכנות יישומים שהוצאו משימוש מהסיבה הזו, מהווים אתגר מאוד גדול עבור מפתחים.

## איזה השפעות יכולות להיות להוצאה משימוש?

על מנת לענות על השאלה הזו, נלקחו בחשבון הספריות הבאות:

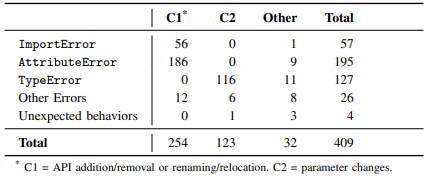
|  |  |
| --- | --- |
| **Library** | **Category** |
| TensorFlow | Deep Learning |
| Keras |
| Scikit-Learn | Data Analytics |
| Pandas |
| Flask | Web Development |
| Django |

טבלה 1: ספריות למחקר

הספריות הללו נבחרו על ידי חיפוש באתר GitHub, באמצעות שימוש של החיפוש המתקדם – language:Python topic:<framework>

מתוך שלל התוצאות נלקחו 1000 פרויקטים ב-GitHub בעלי הדירוג הגבוה ביותר. בתוך אותם פרויקטים נעשה מעבר איטרטיבי על מנת לעבור על כל ה-commit שנעשו ולמצוא בתוכם את המילים compatibility או version. לאחר כמה איטרציות, ניתן היה להבחין ברשימה של מילים שחוזרות על עצמן כאשר מפתחים פותרים בעיות של ספריות [3]:

1. Compatibility
2. Version
3. Evolution
4. Exception
5. TypeError
6. AttributeError
7. ImportError



איור 4: קורלציה בין סימפטום לסיבה [3]

ניתן לראות שרוב מוחלט (405/409) מהמקרים גורמים לקריסות של המערכות בהם יש שימוש בספריות הנ״ל. היחס הוא גבוה מאוד, בהשוואה לפרויקטים אחרים, לדוגמא במחקר שנעשה על התפתחות של אפליקציות אנדרואיד, נמצא שרק 15 מתוך 67 בעיות תאימות של ממשקי תכנות יישומים גרמו לקריסות של האפליקציות [11]. אחת הסיבות המרכזיות שיכולות לגרום לשוני כזה היא הסיבה שפייתון לא עובר תהליך קומפילציה לפני ריצה, מה שמאפשר לאנדרואיד (Java) להיות הרבה יותר עמיד בפני בעיות כאלה.

אם נוסיף אחוזים לטבלה הנ״ל ונבודד את בעיות הקריסה נקבל:

|  |  |  |
| --- | --- | --- |
| שגיאות | כמות | אחוז |
| ImportError | 57 | 14.1% |
| AttributeError | 195 | 48.1% |
| TypeError | 127 | 31.4% |
| Other Errors | 26 | 6.4% |

טבלה 2: כמויות/אחוזים של שגיאות

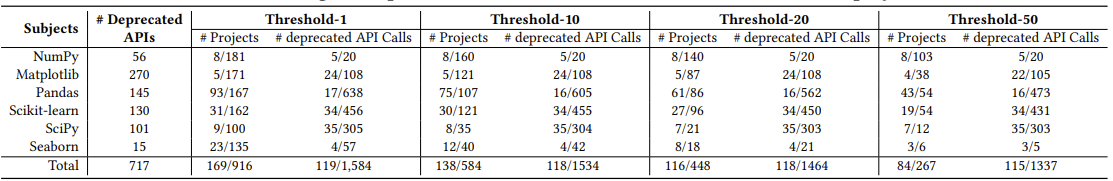
## מהו אופן השימוש?

משתי שאלות המחקר הקודמות, ניתן לראות שיש חוסר סדר בהגדרות של ממשקי תכנות יישומים. לכן, ניתן להניח שחוסר סדר זה עובר ישירות אל המפתחים שמשתמשים בממשקים אלו. בשלב זה, נבחן איך ממשקים מסוג זה, שהוצאו משימוש, באים לידי ביטוי בפרויקטי פייתון ובאיזה רמה מפתחים ניגשים לממשקים כאלה.

על מנת לגשת לבחינה הנ"ל, נלקחו 200 פרויקטים לאחר חיפוש ב-GitHub שמכילים בתוכם שימוש של ששת הספריות 1. מערכת החיפוש של GitHub מבוססת על כלי חיפוש טקסט בסיסי שמקבל רשימה של מילים כקלט, לכן תוצאות החיפוש לא לגמרי קשורות לתנאי החיפוש [12]. על מנת להנמיך את הרעש, הפרויקטים עברו טיהור על ידי ווידוא שהם באמת עשו שימוש בממשקי תכנות יישומים מתוך הספריות שנבדקו. לבסוף, נמצאו בסך הכל 916 פרויקטי פייתון שהשתמשו ביותר מ-200 אלף קריאות לממשקי תכנות יישומים, מתוך ששת הספריות שנבדקו.

מתוך כלל הפרויקטים, נלקחו כל הודעות ה-commit, ונבדק כמה מהם מכילות שילוב של המילים “*fix”* וגם *“deprecat\*”*. עבור כל שינויי הקוד בפרויקטי פייתון, ההודעות של שינויי הקוד הללו הכילו דפרקציה לעתים מאוד נדירות (פחות מ-0.1%) [10].

מאחר והמספר של שינויי הקוד הוא מאוד קטן, צריך לבחון שלב אחד עמוק יותר כי הודעות ה-commit לא בהכרח מציגות את התמונה המדויקת בגלל מגוון רחב של ניסוחים (כמו בחירת מילים או סגנון הסבר). נעשה שימוש בכלי שעובר על היסטורית ה-Git של פרויקט, ומוצא את כל ממשקי תכנות היישומים שהוצאו משימוש שכבר לא בשימוש בפרויקט, מה שמאפשר לו להבין שנעשה תיקון לממשקי תכנות יישומים אלה.

בסך הכל, 119/717 ממשקי תכנות יישומים שהוצאו משימוש מתוך ששת הספריות, עדיין נמצאות בשימוש ב-18% (169/916) מהפרויקטים שנבחנו.

איור 5: שימוש בממשקי תכנות יישומים שהוצאו משימוש בפרויקטי משתמשים [10]

מהתוצאות הנ"ל, ניתן לראות שפרויקטים שמשתמשים בספריה Pandas, יש להם הכי הרבה ממשקי תכנות יישומים שהוצאו משימוש (55%=93/167). בנוסף, גם משתמשים בספריה הזו הכי הרבה (40%=638/1584).

מפתחים בדרך כלל משתמשים באסטרטגיות הבאות על מנת לתקן את השגיאות (ראה איור 4). תבניות אלו מכסות רק 93% מהתיקונים, כל היתר כוללות תיקונים ספציפיים למקרה, כך שלא ניתן להפוך אותם לגנריים [13].

1. התאמת מגבלות הגרסה של תלויות ישירות. הקונפליקטים בין התלויות נקבעו, בדרך כלל, על ידי כוונון התלויות הישירות, כך שיהיו תואמות לאלו של התלויות הטרנזיטיביות.
2. עדכון גרסאות של התלויות הישירות שנחוצות בספריות שבהן ניתן לראות את השגיאה.
3. מחיקה של תלויות ישירות בעלות שגיאות, תוך שמירה על אלה שטרנזיטיביות. יש צורך לישר קו בין התלויות הישירות לאלה הטרנזיטיביות. כאשר זה לא מתאפשר, מפתחים לרוב יבחרו למחוק את התלויות הישירות שיוצרות קונפליקטים.
4. שדרוג, או שנמוך, של סביבות וכלי פיתוח. תלויות עם קונפליקטים בין הסביבה המקומית למרוחקת, בדרך כלל נפתרות על ידי שדרוג, או שנמוך, גרסאות של כלי הפיתוח.
5. יצירה של סביבה מבודדת. זה פתרון אפשרי למקרה של חוסר התאמה בין סביבה מקומית לסביבה מרוחקת.

## מהו אופן התיעוד?

ספריות פייתון משתמשות בשיטות ספציפיות על מנת להוציא משימוש ממשק תכנות יישומים מסוים, מה שגורם לצורך הרבה יותר גדול לתיעוד הרבה יותר מדויק ומסודר של שינויים.

מימושים של ממשקי תכנות יישומים בפייתון, שהוצאו משימוש, אינם גנריים אך מתווספים מתוך הצורך לשימוש של כלל המפתחים שמשתמשים בממשקים הללו. נהייה קשה יותר ויותר להמציא מנגנון אוטומטי לניהול ממשקים שהוצאו משימוש, כולל תיעוד.

לכן, מנהלי ספריות בפייתון תלויים בפתרונות ad-hoc, ובנוסף, לכתיבת תיעוד של אותם ממשקים. דבר זה בהרבה מקרים מוביל לטעויות.

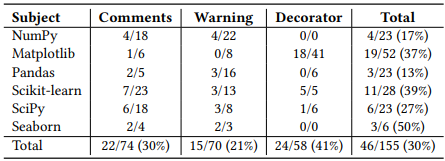
קיימים שלושה סוגים של הוצאה משימוש – comments, הזהרות hard-coded, decorators. באיור הבא ניתן לראות שכל אחד מהם נמצא ביחסית אחוז מאוד נמוך מהספריות [10].

Table

Description automatically generated

איור 6: ממשקי תכנות יישומים שהוצאו משימוש ללא תיעוד [10]

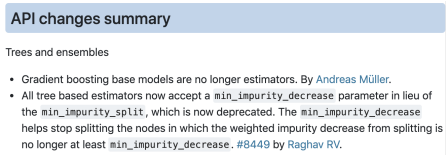
בסך הכל, קרוב ל-30% מתוך כלל הממשקים, שיש להם תיעוד, לא מספקים אלטרנטיבות למפתחים, מה שניתן לראות בתוצאות המאמר לגבי ממשקי תכנות יישומים של Java [5].



איור 7: ממשקי תכנות יישומים שהוצאו משימוש ללא אלטרנטיבות [5]

באמצעות ההבנה של הסיבות שממשק הוצא משימוש, עוזר למפתחים להגיע להחלטות מושכלות. 41% מממשקים כאלה שמוגדרים בעזרת decorator, לא מביאים אלטרנטיבות, שזה הרבה יותר משני סוגי ההוצאה משימוש האחרים (comments, הזהרות hard-coded) [10].

במקרים מסוימים, יכול להיות מדובר בחוסר פירוט של ההוצאה משימוש.



איור 8: דוגמא - ממשק תכנות יישומים מוגדר בצורה לא ברורה [14]

לאורך התיעוד של ממשקים שהוצאו משימוש, חלקם לא ברורים, ואפילו לא מתוארים בצורה נכונה. לדוגמא, איור 8 מציג תיעוד לא ברור של ממשק תכנות יישומים שהוצא משימוש, כאשר הפרמטר “min\_impurity\_split” הוצא משימוש, מהסיבה של “All tree based estimators are deprecated”, כאשר הסיבה הזו מתייחסת לשבעה ממשקים שונים ולא רק לאחד.

במקרה הזה, ייקח למפתחים הרבה יותר זמן להבין באיזה שינויים מדובר, ואיזה שינויים עליהם לעשות בפרויקט שלהם, על מנת לעבוד בצורה נכונה עם השינויים הללו.

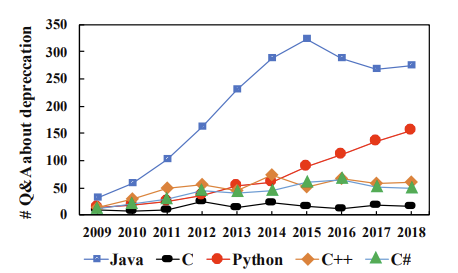
קיים מספר גדול מאוד של ממשקים שהוצאו משימוש שלא נמצאים וגם לא מיוחסים בתיעוד הרשמי של הספריה. מבחינת אלה שיש להם תיעוד, לצערינו, לא לכולם יש אלטרנטיבה שתעזור למפתחים להימנע משימוש של ממשקים שכבר אינם נתמכים.

היוצרים של ממשקי תכנות יישומים, לא משתמשים בהודעות הוצאה משימוש בצורה מספיק טובה על מנת לעזור לכל המפתחים שמשתמשים באותם ממשקים, להימנע מאותן בעיות שימוש. פחות מ-25% מהממשקים הגדירו אלטרנטיבה ראויה לממשק שלהם. לא רק שרוב היוצרים לא מסבירים את ההוצאה משימוש, הם גם לא מגדירים בדיוק מתי זה נכנס לפועל.

## איך הקהילה מגיבה?

בכדי להבין איך באים לידי ביטוי שיחות, של מפתחים, בנוגע להוצאה משימוש של ממשקי תכנות יישומים, נבדקו שאלות שנשאלו על ידי מפתחים באתר הכי מפורסם לדבר הזה – Stack Overflow [15]. נעשה חיפוש של שילוב של מילות מפתח שכוללות – שפות תכנות (Java, Python, C, C++) וגם מילים שקשורות להוצאה משימוש (כגון: deprecate, deprecated, deprecation), ביחד עם מאגר רשמי של Stack Overflow (Sotorrent) [16]. נלקחו דוגמאות של 10 שנים, מה שיצר מאגר של 1,063,837 פוסטים.

מתוך כל הפוסטים הנ"ל חושבה הכמות לפי שפת תכנות –

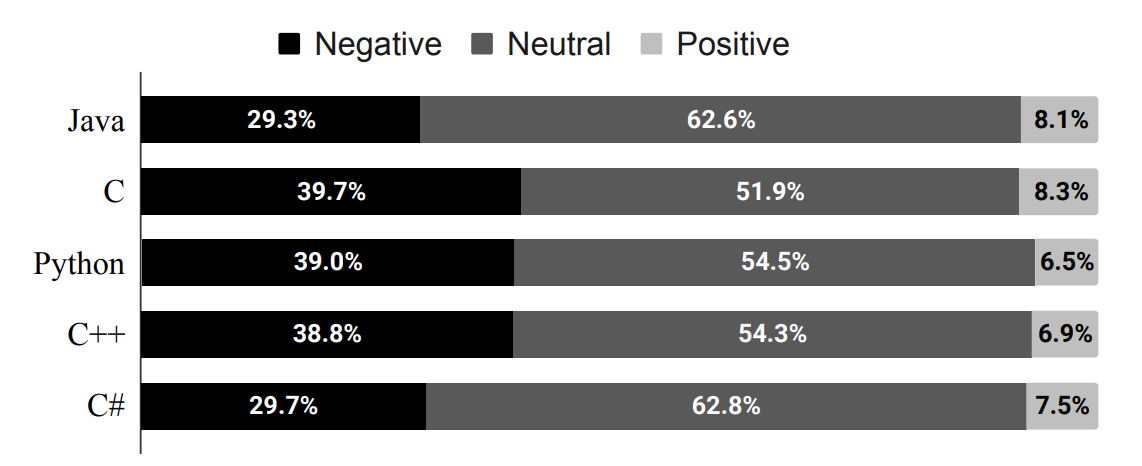


איור 9: התפלגות של פוסטים שקשורים לממשקי תכנות יישומים שהוצאו משימוש לפי שפת תכנות [10]

ניתן לראות שהכי הרבה פוסטים שמדברים על הוצאה משימוש הם בשפת תכנות Java. למרות זאת, לאחר שהגיעה לשיא בשנת 2015, כמות הפוסטים החלה לרדת עם השנים. בנוסף, ניתן לראות שרק פוסטים שקשורים לשפת התכנות פייתון, בעלייה מתמדת. לכן, הוכחה זו מראה לנו שפוסטים שנוגעים להוצאה משימוש של ממשקי תכנות יישומים בפייתון מושכים הרבה יותר יחס בקרב מפתחי פייתון בשנים האחרונות.

לאחר מכן, על מנת לחקור כיצד מפתחים מרגישים כלפי הדיונים על הוצאה משימוש, נעשה שימוש בכלי שנקרא Senti4D. מדובר בכלי שמנתח סנטימנטליות בתחום הפיתוח תוכנה [17]. ניתוח סנטימנטלי, הוא מחקר מאוד סוביקטיבי (נייטרלי לעומת אמוציונלי) וגם קוטביות (חיובי לעומת שלילי) של טקסט [18]. זה מתבסס על לקסיקון של סנטימנטליות, שהוא אוסף גדול של מילים, כל אחת ממופה עם אוריינטציה חיובית ושלילית. הסנטימנט הכללי של טקסט מחושב לפי אותו מיפוי של המילים המוכלות בטקסט. על הבסיס הזה נבנה הכלי Senti4D.

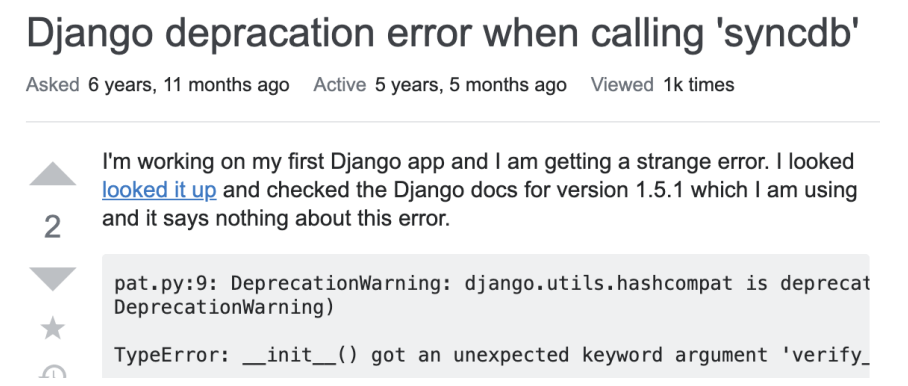
תגובות של מפתחים מתחלקים לשלוש קטגוריות (למשל: שלילי, נייטרלי וחיובי). התוצאות של הניסוי הזה מוצגות באיור הבא.



איור 10: התפלגות קוטביות של מפתחים בקרב חמשת השפות הפופולריות ביותר [10]

ניתן לראות כי לפייתון יש את השיח החיובי הנמוך ביותר, בקרב השפות הפופולריות ביותר, בנוגע להוצאה משימוש של ממשקי תכנות יישומית. בנוסף, בנוגע לקוטביות השלילית, לפייתון יש את הכמות כמעט הכי גבוהה מבין החמש שפות. נתונים אלה מרמזים, שמפתחי פייתון, לא מרוצים עם צורת ההוצאה משימוש של הממשקים הנוכחית, בקרב הקהילה של פייתון [10].

האיור הבא מציג דוגמא, בה מפתח מתלונן לגבי הוצאה משימוש של ממשק תכנות יישומים, כמו כן, החוסר פירוט של התיעוד עבור אותה הוצאה משימוש.



איור 11: דוגמא - מפתח שמתלונן על צורת התיעוד [19]

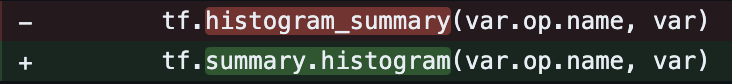
ככל שפייתון נהיית יותר פופולרית, נושאים כמו הוצאה משימוש של ממשקי תכנות יישומים מוזכרים הרבה יותר בדיונים באתרים כמו StackOverflow. כמו התגובות שניתן לראות בשפות תכנות אחרות, כך גם מפתחי פייתון מגיבים בצורה שלילית כאשר עולה הנושא של הוצאה משימוש של ממשקים. לכן, יש צורך בלהבין את התנהגותן בפייתון, ולתת פתרון שיעזור למפתחים להתמודד עם בעיות מסוג זה.

## מהם השיטות לפתרון הבעיה?

שינויים בממשקי תכנות יישומים בפייתון יכולים לגרום לקריסות או להתנהגויות לא צפויות באפלקיציות, כולל יותר מ-10 סוגים שונים של שגיאות זמן ריצה, כאשר רובם ייחודיים לפייתון. על ידי מחקר של מגוון רחב של התנהגויות ותבניות של ספריות, ניתן לראות שמפתחים לרוב מיישמים את ארבעת השיטות הבאות כאשר מתקנים בעיות של ספריות שהוצאו משימוש:

### החלפה לממשק תכנות יישומים אחר

זה אולי התיקון הכי נפוץ, כאשר מפתחים מחליפים את הממשק עם הספריה לממשק אחר (לרוב מאותה ספריה), לדוגמא [20]:

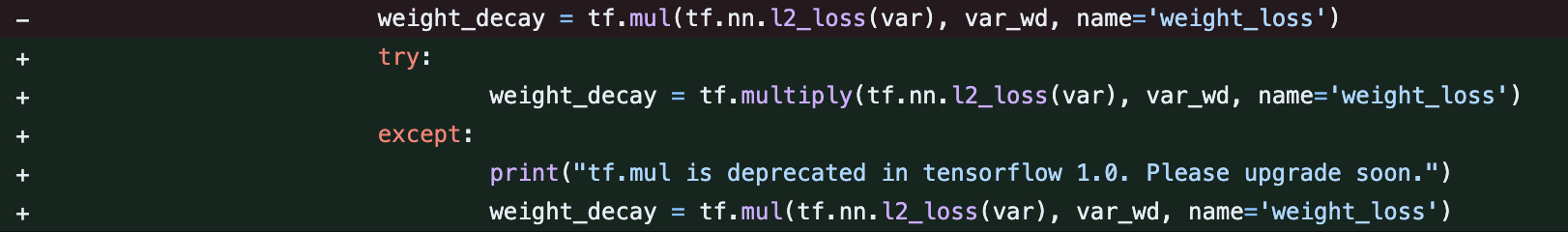


איור 12: תיקון בעיה על ידי החלפת ממשק

המקרה הנ״ל קרה בגלל שהממשק הועבר למקום אחר והשם שלו השתנה בזמן השינויים שנעשו בספריה TensorFlow. בזמן ששינויים כאלה הם יחסית קלים לביצוע, לא ניתן להבטיח שהאפליקציה תהיה מותאמת לגרסאות ישנות יותר של הספריה.

### לתפוס שגיאות מסוגים שונים

כמו שראינו, שינויים בממשקי תכנות יישומים יכולים לגרום לקריסות של אפליקציות עם שגיאות כמו ImportErorr. על מנת לתקן מקרים כאלה, מפתחים לעתים מוצאים בשימוש של try-catch כמאוד שימושי לטובת תפיסה של שגיאות ואפילו לעבור לאלטרנטיבות אחרות (בתוך הבלוק של catch) [21].



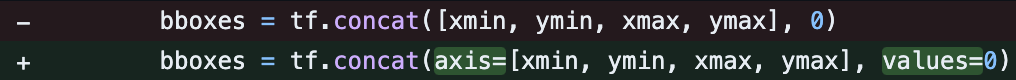
איור 13: תפיסת שגיאות

כאן ניתן לראות שהמפתחים ניסו להשתמש בפונקציה multiply של Tensorflow, אך היה שינוי של שם הפונקציה בין הגרסאות (ככל הנראה מגרסא 1.0). במקרה והפונקציה לא קיימת, תהיה כאן שגיאה והפונקציה mul תיקרא במקום (מאחר והיא קיימת בגרסאות ישנות יותר).

מקרה מעניין ניתן לראות כאן (איור 13), שימוש בפונקציה מגרסאות חדשות יותר בבלוק try ובאלטרנטיבות בבלוק catch. שימוש מסוג כזה מונע שימוש ״יקר״ מבחינת חישובים וזמן ריצה מאחר וזריקת שגיאות הוא תהליך ״יקר״. כמו כן, גישה כזו מאפשרת תאימות לאחור. אך רק במקרים של חוסר התאמה אחורה ולא במקרים של אי התאמות חדשות.

### הוספה או שינוי מיקום או שם של פרמטרים

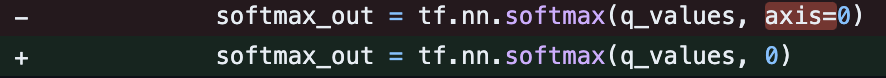
שינוים בחתימה של הפונקציה גורמים לשבירה של הפרויקט או להתנהגות לא צפויה. לכן, לעתים מפתחים בוחרים לשנות את הגישה מפרמטרים לפי מיקום לגישה של פרמטרים לפי keyword. לדוגמא [22]:



איור 14: שינוי לגישה של פרמטרים לפי keyword

הגורם לבעיה עם הממשק הוא שינוי של מיקום הפרמטרים. לכן, על ידי הוספת ה-keyword של הפרמטרים לפני כל ערך, שומר על אחידות גם אם יהיו עוד שינויים דומים בעתיד.

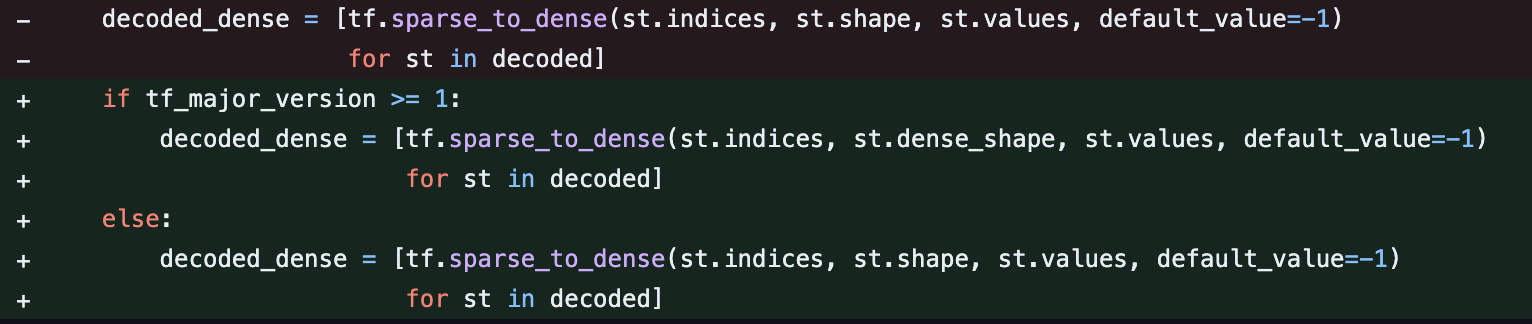
מה שפתרון זה לא פותר, הוא שינוי של שמות הפרמטרים. במידה ויש שינוי של שמות, הפתרון הקודם לא עוזר. לכן, במקרים מסוימים מפתחים בוחרים דווקא כן להשתמש בגישה של ערכים לפי מיקום בחתימה של הפונקציה. גישה כזו מאפשרת תאימות לאחור [23].



איור 15: שינוי לגישה של פרמטרים לפי מיקום

### בדיקת גרסא של ספריה

לפני קריאה לממשק מסוים, זה נפוץ בקרב מפתחים לבדוק את גרסת הספריה.



איור 16: בדיקת גרסאת הספריה

בדוגמא הנ״ל ניתן לראות איך מפתחים בודקים אם הממשק שבו הם מתכוונים להשתמש קיים בעזרת בדיקה של הגרסא של הספריה. בדיקות מסוג כזה דורשות מהמפתחים להיות מודעים לשינויים שנעשים בספריות בהן הם משתמשים, כמו כן, לדעת את הגרסא המדויקת של אותה ספריה בה היה שינוי בממשק תכנות יישומים הספציפי הזה.

בזמן שלפתרון זה יש תאימות לאחור, הוא לא מתאים לכל מפתח. צריך שיהיה מפתח מנוסה על מנת לנתח ולזהות את ההגדרה המדויקת של הספריה המדוברת.

# החלטות תיכון בפיתוח הכלי

הכלי שאותו בניתי [24], הוא כלי שמתפקד כסקריפט חיצוני לכל דבר. משתמש שרוצה לבדוק את הפרויקט שלו יהיה צריך להריץ את הסקריפט ולתת, כקלט, את המיקום האבסולוטי של הפרויקט שלו (שאותו הוא מעונין לבחון).

## בסיס

ברמה העליונה של הכלי נעשה שימוש של כמה ספריות בסיסיות על מנת לתת את היכולת להזין פרמטרים וגם לגרום לכלי לרוץ בצורה מקבילית.

1. *argparse* [25]– בשימוש לטובת הזנת פרמטרים לכלי. ניתן להזין פרמטרים כגון: (1) מיקום אבסולוטי (*--dir*) של הפרויקט אותו צריך לנתח (פרמטר שהוא חובה). (2) כמות החוטים (*--threads*) שהמשתמש רוצה שירוצו (ברירת מחדל היא 20). (3) "ורבוס" (*--verbose*) – דגל שמסמן לכלי להדפיס את כלל ההדפסות לוג שקיימות במערכת ללא שום קשר לרמת הלוג שמוגדרת בפרמטרים (ברירת מחדל – *False*). (4) רמת לוג (*--logging\_level*) – יכולה להיות בטווח המספרים [1-4] (ברירת מחדל - 3), כאשר: *ERROR = 1, WARN = 2, INFO = 3, DEBUG = 4*
2. *multiprocessing* [26]– בשימוש לטובת ריצת הכלי בצורה מקבילית. הפרמטרים שהמשתמש מזין (או אלה שמוגדרים כברירת מחדל) מגדירים את כמות ה-threads שירוצו, מה שמזין את המחלקה Pool (מתוך הספריה הנ"ל).

## לוגים

על מנת לדווח למשתמש את מהלך הבדיקה, מה נמצא ומה התוצאות, בניתי מחלקה שמדפיסה את הפלטים בצבעים לפי רמת הלוג.

1. *DEBUG* – כחול
2. *INFO* – ברירת מחדל של מערכת ההפעלה
3. *WARN* – צהוב
4. *ERROR* – אדום

על מנת להגדיר את הרמות הנ"ל, נבנה *Enum* :

class VerboseLoggingLevel(Enum):

    def \_\_le\_\_(self, b):

        return self.value <= b

    def \_\_ge\_\_(self, b):

        return self.value >= b

    ERROR = 1

    WARN = 2

    INFO = 3

    DEBUG = 4

עבור ה-enum, הוגדרו גם שתי פונקציות על מנת לעשות השוואות עם ערכים אחרים (גדול מ, וקטן מ).

המחלקה שמנהלת את הלוגים, מכילה ארבע פונקציות (אחת עבור כל רמת לוג) כאשר כל פונקציה מדפיסה את הלוג שקיבלה בצבע שמוגדר לה.

class VerboseLogging:

    def \_\_init\_\_(self, verbose: bool, logging\_level: int) -> None:

        self.VERBOSE = verbose

        self.VERBOSE\_LEVEL = logging\_level

    def print\_debug(self, msg: str) -> None:

        if (self.VERBOSE or self.VERBOSE\_LEVEL >= VerboseLoggingLevel.DEBUG):

            print(colored("[debug] " + msg, "blue"))

    def print\_info(self, msg: str) -> None:

        if (self.VERBOSE or self.VERBOSE\_LEVEL >= VerboseLoggingLevel.INFO):

            print(colored("[info] " + msg))

    def print\_warn(self, msg: str) -> None:

        if (self.VERBOSE or self.VERBOSE\_LEVEL >= VerboseLoggingLevel.WARN):

            print(colored("[warn] " + msg, 'yellow'))

    def print\_error(self, msg: str) -> None:

        if (self.VERBOSE or self.VERBOSE\_LEVEL >= VerboseLoggingLevel.ERROR):

            print(colored("[error] " + msg, 'red'))

כמו שנאמר מעלה, בהגדרת פרמטרים, הפרמטר verbose מאפשר להדפיס את כל הלוגים ללא התייחסות לרמת הלוג המוגדרת. ניתן לראות את זה בכל פונקציה של הלוגים כאשר הפרמטר הזה נבדק.

לטובת הדפסה ידידותית למשתמש של תוצאות הבדיקה, נוספה גם פונקציה לטובת התוצאות.

def print\_results(self, error\_msg: str, warn\_msg: str, rest\_msg: str):

    print(f"{colored(error\_msg, 'red')}, {colored(warn\_msg, 'yellow')} {colored(rest\_msg)}")

## פרמטרים לפונקציה

כאשר עוברים על קבצי פייתון, קיימות הרבה קריאות לפונקציות שונות. יש שלוש שיטות להעביר פרמטרים לפונקציה:

1. משתנה לוקאלי שהוגדר בקובץ.
2. ערך שמועבר לפונציה (כמו מספר קבוע או מחרוזת).
3. באמצעות keyword – כאשר מגדירים את שם הפרמטר שרוצים להעביר ומקנים לו ערך או משתנה לוקאלי.

לטובת הגדרה זו, נוסף Enum:

class ParameterType(Enum):

    LOCAL\_PARAM = 1

    LOCAL\_CONST = 2

    KEYWORD = 3

## איסוף מידע

לאחר שהכלי קיבל את כל הפרמטרים מהמשתמש, והריץ את ה-threads, כל thread מקבל קובץ מתוך הפרויקט ומבצע את האנליזה.



**איור 17: תרשים זרימה של קריאת קבצי הפרויקט**

לטובת קריאה של הקבצים ומעבר עליהם, נעשה שימוש בספריה *ast* [27]. ספריה זו מאפשרת לנתח קבצי קוד על ידי מעבר על חלקים שונים של הקוד. עבור המימוש של התרשים הנ"ל, נעשה שימוש בספריה בצורה הבאה:

tree = ast.parse(code, mode='exec')

visitor = AssignVisitor()

visitor.visit(tree)

func\_calls\_names = get\_func\_calls(tree)

המחלקה שעוברת על כלל ההשמות שיש בקוד יורשת מהמחלקה של *ast* [27] שנקראת NodeVisitor.

class AssignVisitor(ast.NodeVisitor):

    def \_\_init\_\_(self):

        self.class\_obj = {}

    def visit\_Assign(self, node):

        call\_name = get\_func\_calls(node.value)

        # for an assignment and its right side has a function call.

        # map this function call to its left

        if len(call\_name) > 0 and isinstance(node.targets[0], ast.Name):

            self.class\_obj[node.targets[0].id] = call\_name[-1][0]

        return node

על כל השמה שהפונקציה visit מזהה, המימוש החדש של הכלי מקבל את הקטע קוד שבו יש השמה. הפונקציה עצמה ממפה את המשתנה שקיבל את ההשמה לפונקציה שנקראה.

ניתן לשים לב שיש שימוש חוזר של הפונקציה get\_func\_calls.



**איור 18: תרשים זרימה - מעבר על קריאות לפונקציות**

הפונקציה הזו משתמשת בשתי מחלקות, שנוצרו עבור הכלי הזה, אחת אוספת את שמות הפונקציות והשניה את הפרמטרים שעוברים אליהן.

class FuncCallVisitor(ast.NodeVisitor):

    '''

    visit function call nodes

    '''

    def \_\_init\_\_(self):

        self.\_name = deque()

    @property

    def name(self):

        return ".".join(self.\_name)

    @name.deleter

    def name(self):

        self.\_name.clear()

    def visit\_Name(self, node):

        self.\_name.appendleft(node.id)

class KeyWordVisitor(ast.NodeVisitor):

    '''

    visit arguments (parameters passed to functions)

    '''

    def \_\_init\_\_(self):

        self.\_name = []

        self.skip = True

    @property

    def name(self):

        return self.\_name

    def visit\_Name(self, node):

        if not self.skip and node.id is not None:

            self.\_name.append(Parameter(node.id, ParameterType.LOCAL\_PARAM))

        self.skip = False

    def visit\_Constant(self, node):

        if node.value is not None:

            self.\_name.append(Parameter(node.value, ParameterType.LOCAL\_CONST))

    def visit\_keyword(self, node):

        if node.arg is not None:

            self.\_name.append(Parameter(node.arg, ParameterType.KEYWORD))

ניתן לראות שכל פרמטר מקבל השמה של הסוג שלו (כמו שהוגדר בפרמטרים לפונקציה).

## ניתוח

כעת, כאשר הושגו כל קריאות הפונקציות מהקבצים של הפרויקט, ניתן לעבור על כל אחד מהם ולהבין האם נעשה שימוש לא תקין באחד ממשקי תכנות יישומים של ספריות מסוימות.

עוברים בלולאה על כל הקריאות שנמצאו, ועל מנת שנוכל לבדוק את הספריה, צריך לעשות לה import תוך כדי ריצה של הכלי. על מנת שנוכל לבצע את זה, נעשה שימוש בספריה *importlib* [28].

importlib.import\_module(api\_module)

בנוסף, עלינו להשיג את החתימה של אותו ממשק שנעשה בו שימוש. בשביל זה נשתמש בספריה *inspect* [29].

inspect.signature(api)



**איור 19: תרשים זרימה של ניתוח הממשקים**

עבור כל ממשק, מבצעים שתי בדיקות:

1. בדיקה של הפרמטרים שמועברים לממשק. תקינות, סדר נכון, וכו'.
2. בדיקה האם כחלק מהמימוש של הממשק יש הזהרה כלשהי שנעשתה על ידי המפתח.

### בדיקת פרמטרים

קיימים כמה סוגים של פרמטרים שניתן להגדיר בחתימה של פונקציה:

1. פרמטר שמועבר רק לפי מיקום
2. פרמטר שמועבר או לפי מיקום או לפי keyword
3. פרמטר שמועבר רק לפי keyword
4. פרמטר שמועבר כמיפוי או רשימה של שדות – הוא מוגדר עם \*\* או \* לפניו (קוראים לו variadic)

תחילה תהיה בדיקה שכלל הפרמטרים שמועברים על ידי הפרויקט אכן קיימים בחתימה של הממשק. לאחר מכן נעבור על הפרמטרים מהחתימה של הממשק ונבצע בדיקה לפי הסוג של הפרמטר.

נשים לב שקיים פרמטר קונבנציונאלי שנוכח תמיד בכל הפונקציות שממומשות בתוך מחלקה – *self*. אנחנו לא נתייחס אליו מאחר והוא לא צריך להיות מועבר לממשק על ידי המשתמש.

for idx, param in enumerate(sig\_values):

    if param.kind == \_ParameterKind.VAR\_POSITIONAL \

            or param.kind == \_ParameterKind.VAR\_KEYWORD:

        # this param is a variadic

        continue

    if param.kind == \_ParameterKind.POSITIONAL\_ONLY \

            and not \_validate\_positional(idx, params):

        inner\_error\_count += 1

        logger.print\_error(f"The {param.kind.description} parameter [{param.name}] is not present in the API call - {api}")

    elif param.kind == \_ParameterKind.POSITIONAL\_OR\_KEYWORD \

        and not \_validate\_keyword(param, keyword\_names) \

            and not \_validate\_positional(idx, params):

        inner\_error\_count += 1

        logger.print\_error(f"The {param.kind.description} parameter [{param.name}] is not present in the API call - {api}")

    elif param.kind == \_ParameterKind.KEYWORD\_ONLY \

            and not \_validate\_keyword(param, keyword\_names):

        inner\_error\_count += 1

        logger.print\_error(f"The {param.kind.description} parameter [{param.name}] is not present in the API call - {api}")



**איור 20: תרשים זרימה של בדיקת הפרמטרים**

### בדיקת הזהרות

מפתחי ממשקי תכנות יישומים משתמשים בסוגים שונים של תיעוד (ניתן לראות בחלק ‏2.4 - מהו אופן התיעוד?). אחד מהשיטות הוא להכניס הזהרה בקוד באמצעות ספריה שקוראים לה *warnings* [30].

על מנת שנוכל לבדוק האם יש הזהרות כלשהם במימוש של הממשק תכנות יישומים שנעשה בו שימוש, נצטרך להשיג את הקוד בזמן ריצה. נעשה זאת בעזרת הספריה *inspect* [29].

inspect.getsource(api)

על מנת לבדוק האם קיימת קריאה לפונקציה warn של הספריה *warnings* [30], נשתמש בפונקציה get\_api\_calls שמימשנו (ניתן לראות בחלק ‏3.4 - איסוף מידע).



**איור 21: תרשים זרימה של בדיקת הזהרות**

# סיכום

בעבודה זו, נעשה מחקר של ממשקי תכנות יישומים שהוצאו משימוש על ידי ניתוח של פרויקטים, תיעודים, תגובות קהילה ושיטות לפתרון. וזאת על מנת להבין את:

1. תדירות ההוצאה משימוש של ממשקי תכנות יישומים.
2. ההתנהגות של השינויים הללו לאורך זמן.
3. ההשפעה על אפליקציות ופרויקטים של מפתחים שמשתמשים בממשקים האלה.
4. המאפיינים של הספריות עם תדירות גבוהה ונמוכה של שינויים.

נותחו מאגרים רבים על מנת להבין את השאלות הבאות:

1. *איך ספריות מבצעות הוצאה משימוש?*

נמצאו ארבע עשרה תבניות כאשר רק חמישה מהם לא נמצאו ב-Java. כאשר חמשת התבניות הללו מתחלקות לשלוש קטגוריות: (i) מחיקה של מחלקה או פונקציה. (ii) שינוי חתימה של הפונקציה. (iii) ערכים שפרמטרים יכולים לקבל כקלט השתנו.

1. *איזה השפעות יכולות להיות להוצאות משימוש?*

מעל ל-98% ממקרי ההוצאה משימוש גורמים לקריסות של המערכות. לעומת מחקר אחר שמראה שעבור פרויקטים של אנדרואיד יש רק 22% מקרים כאלה.

1. *מהו אופן השימוש?*

בסך הכל, 119/717 ממשקי תכנות יישומים שהוצאו משימוש מתוך ששת הספריות, עדיין נמצאים בשימוש ב-18% (169/916) מהפרויקטים שנבחנו.

1. *מהו אופן התיעוד?*

היוצרים של ממשקי תכנות יישומים, לא משתמשים בהודעות הוצאה משימוש בצורה טובה מספיק על מנת לעזור לכל המפתחים שמשתמשים באותם ממשקים להמנע מבעיות שימוש. פחות מ-25% מהממשקים, שהוצאו משימוש, הגדירו אלטרנטיבה ראויה לממשק שלהם. לא רק שרוב היוצרים לא מסבירים את ההוצאה משימוש, הם גם לא מגדירים מתי (באיזה גרסא) הממשק יוצא משימוש.

1. *איך הקהילה מגיבה?*

לפייתון יש את השיח החיובי הנמוך ביותר, בקרב השפות הפופולריות ביותר, בנוגע להוצאה משימוש של ממשקי תכנות יישומים. בנוסף, בנוגע לקוטביות השלילית, לפייתון יש את הכמות כמעט הכי גבוהה מבין החמש. נתונים אלה מרמזים, שמפתחי פייתון, לא מרוצים עם צורת ההוצאה משימוש של הממשקים הנוכחית, בקרב קהילת משתמשי פייתון.

1. *מה השיטות לפתרון הבעיה?*

שינויים בממשקי תכנות יישומים בפייתון יכולים לגרום לקריסות או להתנהגויות לא צפויות באפלקיציות, כולל יותר מ-10 סוגים שונים של שגיאות זמן ריצה, כאשר רובם ייחודיים לפייתון. על ידי מחקר של מגוון רחב של התנהגויות ותבניות של ספריות, ניתן לראות שמפתחים לרוב מיישמים את ארבעת השיטות הבאות כאשר מתקנים בעיות של ספריות שהוצאו משימוש: (i) החלפה לממשק תכנות יישומים אחר. (ii) תפיסת שגיאות מסוגים שונים. (iii) הוספה או שינוי מיקום או שם של פרמטרים. (iv) בדיקת גרסא של ספריה.

התמקדנו בספריות מפורסמות של פייתון על מנת לענות על שאלות אלו. הממצאים קריטיים, על מנת להבין שההוצאות משימוש של הממשקים, מטופלים בצורה לא תקנית ולא מסודרת.

תוך כדי חקירה מעמיקה של השימושים השונים של אותם ממשקי תכנות יישומים, בפרויקטים נבחרים של פייתון, שהוצאו משימוש, ניתן לראות שמפתחים שמים לב לעתים רחוקות להוצאה משימוש של אותם ממשקים. ממצאים אלה, מצביעים בצורה חד משמעית שהקהילה חייבת לקחת על עצמה יצירה של טכניקות, על מנת לעזור ליוצרים של הספריות וגם למפתחים להתמודד עם הוצאה משימוש של ממשקי תכנות משתמש.

על סמך המחקר בנושא של הוצאה משימוש, בניתי כלי שיעזור בסריקה של הפרויקטים בפייתון, ויזהה האם יש סכנה או חשש להוצאה משימוש של ממשקי תכנות יישומים כלשהם בפרויקט. הכלי הזה יוכל לזהות בצורה עצמאית את הבעיות הללו.

הכלי הזה עובר על כלל הקריאות, שקיימות בפרויקט, לממשקי תכנות יישומים שונים, ומוודא שהשימוש בממשק הוא תקין ואין שום הזהרות מובנות בתוך הקוד שהמפתחים השאירו. השימוש בכלי הנ"ל הוא קל ופשוט לכל משתמש.

עבודות המשך יכולות לשכלל את הכלי הזה ולהפוך אותו לחלק אינטגרטיבי (למשל plugin) של IDE כמו Visual Studio Code.

# ביביליוגרפיה

|  |  |
| --- | --- |
| [1] | "The top programming languages," GitHub, 2022. [Online]. Available: https://octoverse.github.com/2022/top-programming-languages. [Accessed September 2023]. |
| [2] | D. Dig and R. Johnson, "The role of refactorings in API evolution," *21st IEEE International Conference on Software Maintenance (ICSM'05),* pp. 389-398, 2005. |
| [3] | Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu and Y. Xiong, "How Do Python Framework APIs Evolve? An Exploratory Study," *IEEE 27th international conference on software analysis, evolution and reengineering (SANER),* pp. 81-92, 2020. |
| [4] | L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia and J. Klein, "Characterising deprecated android APIs.," *Proceedings of the 15th International Conference on Mining Software Repositories,* pp. 254-264, 2018. |
| [5] | G. Brito, A. Hora, M. Tulio Valente and R. Robbes, "On the use of replacement messages in API deprecation: An empirical study," *Journal of Systems and Software,* pp. 306-321, 2018. |
| [6] | R. Robbes, M. Lungu and D. Röthlisberger, "How do developers react to API deprecation? The case of a Smalltalk ecosystem," *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering,* pp. 1-11, 2012. |
| [7] | A. Hora, R. Robbes, N. Anquetil, A. Etien, S. Ducasse and M. T. Valente, "How do developers react to API evolution? the pharo ecosystem case.," *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME),* pp. 251-260, 2015. |
| [8] | S. A. Haryono, F. Thung, H. J. Kang, L. Serrano, G. Muller, J. Lawall, D. Lo and L. Jiang, "Automatic Android deprecated-API usage update by learning from single updated example.," *Proceedings of the 28th international conference on program comprehension,* pp. 401-405, 2020. |
| [9] | M. Fazzini, Q. Xin and A. Orso, "Automated API-usage update for Android apps," *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis,* pp. 204-215, 2019. |
| [10] | J. Wang, L. Li, K. Liu and H. Cai, "Exploring How Deprecated Python Library APIs are (Not) Handled," *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering,* pp. 233-244, 2020. |
| [11] | L. Wei, Y. Liu and S. Cheung, "Taming android fragmentation: characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, Singapore, 2016. |
| [12] | Y. Zhang, D. Lo, P. S. Kochhar, X. Xia, Q. Li and J. Sun, "Detecting similar repositories on GitHub.," *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER),* no. IEEE, pp. 13-23, 2017. |
| [13] | W. Ying, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu and Z. Zhu, "Watchman: Monitoring dependency conflicts for python library ecosystem," *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering,* pp. 125-135, 2020. |
| [14] | scikit-learn, "scikit-learn - Version 0.19," [Online]. Available: https://scikit-learn.org/stable/whats\_new/v0.19.html#api-changes-summarys. [Accessed November 2023]. |
| [15] | "Stack Overflow," Where Developers Learn, Share, & Build Careers, [Online]. Available: https://stackoverflow.com/. [Accessed October 2023]. |
| [16] | S. Baltes, L. Dumani, C. Treude and S. Diehl, "Sotorrent: reconstructing and analyzing the evolution of stack overflow posts," *Proceedings of the 15th international conference on mining software repositories,* pp. 319-330, 2018. |
| [17] | F. Calefato, F. Lanubile, F. Maiorano and N. Novielli, "Sentiment polarity detection for software development," *Proceedings of the 40th International Conference on Software Engineering,* pp. 128-128, 2018. |
| [18] | B. Pang and L. Lee, "Opinion mining and sentiment analysis.," *Foundations and Trends® in information retrieval 2,* no. 1-2, pp. 1-135, 2008. |
| [19] | N. Villaescusa, "StackOverflow," StackOverflow, [Online]. Available: https://stackoverflow.com/questions/15847931/django-depracation-error-when-calling-syncdb. [Accessed December 2023]. |
| [20] | wolframteetz/colorization.tensorflow, "GitHub," 8 June 2017. [Online]. Available: https://github.com/shekkizh/Colorization.tensorflow/commit/bec97c520d62cb88d4c2969f90bb51437494b1cc?diff=unified. [Accessed November 2023]. |
| [21] | jparkhill/TensorMol, 2017. [Online]. Available: https://github.com/jparkhill/TensorMol/commit/b8d07db49d037048d7e312083d70bf25798b8ac8. [Accessed October 2023]. |
| [22] | visipedia/tf\_classification, 2017. [Online]. Available: https://github.com/visipedia/tf\_classification/commit/df29c8c1b56ceef4c9d56f131aa3a6bd16134307. [Accessed October 2023]. |
| [23] | sadeepj/crfasrnn\_keras, 2018. [Online]. Available: https://github.com/sadeepj/crfasrnn\_keras/commit/6bfaee73e4388a5b71b041f6d5949a3cf8989a62. [Accessed October 2023]. |
| [24] | Y. Shpund, "Project Dependencies Deprecations Analyzer," 2023. [Online]. Available: https://github.com/YoniShpund/OU-Final-Project. |
| [25] | Python, "argparse — Parser for command-line options, arguments and sub-commands," [Online]. Available: https://docs.python.org/3/library/argparse.html. [Accessed January 2024]. |
| [26] | Python, "multiprocessing — Process-based parallelism," [Online]. Available: https://docs.python.org/3/library/multiprocessing.html. [Accessed January 2024]. |
| [27] | Python, "ast — Abstract Syntax Trees," [Online]. Available: https://docs.python.org/3/library/ast.html. [Accessed January 2024]. |
| [28] | Python, "importlib — The implementation of import," [Online]. Available: https://docs.python.org/3/library/importlib.html. [Accessed January 2024]. |
| [29] | Python, "inspect — Inspect live objects," [Online]. Available: https://docs.python.org/3/library/inspect.html. [Accessed February 2024]. |
| [30] | Python, "warnings — Warning control," [Online]. Available: https://docs.python.org/3/library/warnings.html. [Accessed February 2024]. |
| [31] | X. Laerte, A. Brito, A. Hora and M. Tulio Valente, "Historical and Impact Analysis of API Breaking Changes: A Large-Scale Study," *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER),* pp. 138-147, 2017. |

# Contents

|  |  |  |
| --- | --- | --- |
| 4 | Introduction | ‏1 |
| 5 | Python | ‏1.1 |
| 5 | Application Programming Interface | ‏1.2 |
| 5 | Dynamic Programming vs. Static Programming | ‏1.3 |
| 5 | Research Questions | ‏2 |
| 6 | How is Depracation Done in Libraries? | ‏2.1 |
| 8 | How Can Depracation Impact a Project? | ‏2.2 |
| 9 | What are the Usage Patterns? | ‏2.3 |
| 11 | How do Depracations Documented? | ‏2.4 |
| 13 | How does the Community React? | ‏2.5 |
| 15 | What are the Ways to Solve this Problem? | ‏2.6 |
| 15 | Moving to a Different API | ‏2.6.1 |
| 15 | Catch Exceptions of Different Kinds | ‏2.6.2 |
| 16 | Changing Position or Name of a Parameter | ‏2.6.3 |
| 16 | Validating the Version of the Library | ‏2.6.4 |
| 17 | Software Design | ‏3 |
| 17 | Basics | ‏3.1 |
| 17 | Logs | ‏3.2 |
| 18 | Function Parameters | ‏3.3 |
| 19 | Information Collection | ‏3.4 |
| 22 | Analysis | ‏3.5 |
| 23 | Parameters Validation | ‏3.5.1 |
| 25 | Warnings Validation | ‏3.5.2 |
| 27 | Summary | ‏4 |
| 29 | Bibliography | ‏5 |
|  |  |  |

# Abstract

One of the most popular programming languages in the past years is Python, and its libraries are used for data science, machine learning, and web development. This programming language became popular thanks to the wide selection of libraries including data science and many scientific calculation libraries.

Application Programming Interfaces (APIs) in Python libraries get deprecated due to version promotions after feature enhancements and bug fixes, in the same way as in other programming languages. Most of the API changes include movement or changes of functions' prototypes, whether it is a parameter name change or its location.

Similar to frameworks in other languages, the APIs provided by Python frameworks often grow, which would yield compatibility issues in client applications. Libraries are commonly used to support code reuse and increase productivity.

Those kinds of changes make developers less motivated to use Python for software development, as well as compatibility issues with older versions. API elements should always get deprecated with information to help the client. However, in practice, there is evidence that those APIs are deprecated without any information to help the community developers adapt their applications to the changes.

It is preferred to use the deprecate-replace-remove technique to allow the developers to adapt to these changes smoothly. In this method, APIs that are not supported anymore will be primarily defined as deprecated, an informative message is added with replacements to help the developers transition to new APIs. With every version promotion, deprecated APIs are removed. Unfortunately, this procedure is not how the deprecation process is done.

Studies have shown suggestions for techniques that automatically update the deprecated APIs. However, most of them are meant for static programming languages like C# and Java, while Python is a dynamic programming language that presents different evolution patterns of its API in comparison to Java, for example.

It gives motivation for the need for new tools and techniques to detect deprecated APIs. There was evidence that developers who maintain the Python libraries, usually use more than one method to mark deprecation. It causes an inconsistency in the client application development. To avoid applying deprecated APIs, developers need to be aware of the APIs they use in their projects.

The above gives motivation for this study. I aim to apprehend how deprecated library APIs are declared and documented in practice by their owners and how library users use them and react to the deprecations.

By thoroughly looking into six decent Python libraries and over 1,000 GitHub projects, I find that API deprecations are poorly handled by library owners, which later introduces difficulties for Python developers in resolving the usage of deprecated library APIs. This proof indicates that our development community should take immediate action to handle the deprecation of Python library APIs. In this paper I introduce a tool, that does exectly that, and can make developers’ lives easier. This tool summarizes all the potential deprecation places and also marks APIs that have hard-coded warnings embedded in them.

**The Open University**

**Department of Math and Computer Science**

|  |
| --- |
|  |
| **Handling Deprecated Python Library APIs** |
| Paper submitted as fullfilment of the requirements  towards a Master of Science in Computer Science  By  **Yoni Shpund**  Prepared under the supervision of: Prof. Shmuel Tyszberowicz  **2023** |