**האוניברסיטה הפתוחה**

**המחלקה למתמטיקה ולמדעי המחשב**

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

תוכן עניינים

|  |  |  |
| --- | --- | --- |
| ‏1 | הקדמה | 4 |
| ‏2 | רקע | 4 |
| ‏2.1 | פייתון – Python | 4 |
| ‏2.1.1 | תכנות דינאמי | 4 |
| ‏2.2 | ממשק תכנות יישומים - API | 4 |
| ‏2.3 | תכנות דינאמי מול סטטי | 5 |
| ‏3 | שאלות מחקר | 5 |
| ‏3.1 | איך סיפריות מבצעות דפרקציה? | 5 |
| ‏3.2 | איזה השפעות יכולות להיות לדפרקציות? | 7 |
| ‏3.3 | מהו אופן השימוש? | 8 |
| ‏3.4 | מהו אופן הדוקומנטציה? | 9 |
| ‏3.5 | איך הקהילה מגיבה? | 11 |
| ‏3.6 | מהם השיטות לפתרון הבעיה? | 12 |
| ‏3.6.1 | החלפה לממשק תכנות יישומים אחר | 12 |
| ‏3.6.2 | לתפוס שגיאות מסוגים שונים | 12 |
| ‏3.6.3 | הוספה או שינוי מיקום או שם של פרמטרים | 13 |
| ‏3.6.4 | בדיקת גרסא של סיפריה | 13 |
| ‏4 | סיכום | 14 |
| ‏5 | ביביליוגרפיה | 15 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

רשימת איורים

[איור 1: תבניות שינויים של ממשקי תכנות יישומים 5](#_Toc145864882)

[איור 2: שינויים של חתימות לאורך גרסאות שונות ב-TensorFlow 6](#_Toc145864883)

[איור 3: דפרקציה של חתימה של פונקציה 6](#_Toc145864884)

[איור 4: קורלציה בין סימפטום לסיבה 7](#_Toc145864885)

[איור 5: ממשקי תכנות יישומים שהוצאו משימוש ללא דוקומנטציה 9](#_Toc145864886)

[איור 6: ממשקי תכנות יישומים שהוצאו משימוש ללא אלטרנטיבות 9](#_Toc145864887)

[איור 7: דוגמא - ממש תכנות יישומים מוגדר בצורה לא ברורה 10](#_Toc145864888)

[איור 8: התפלגות הסנטמנטליות של מפתחים בקרב חמשת השפות הפופולריות ביותר 11](#_Toc145864889)

[איור 9: דוגמא - מפתח שמתלונן על צורת הדוקומנטציה 11](#_Toc145864890)

[איור 10: תיקון בעיה על ידי החלפת ממשק 12](#_Toc145864891)

[איור 11: תפיסת שגיאות 12](#_Toc145864892)

[איור 12: שינוי לגישה של פרמטרים לפי keyword 13](#_Toc145864893)

[איור 13: שינוי לגישה של פרמטרים לפי מיקום 13](#_Toc145864894)

[איור 14: בדיקת גרסא של סיפריה 13](#_Toc145864895)

רשימת טבלאות

[טבלה 1: סיפריות למחקר 7](#_Toc145864896)

[טבלה 2: כמויות/אחוזים של שגיאות 8](#_Toc145864897)

# הקדמה

אחת השפות תכנות הכי פופולריות בשנים האחרונות היא Python [1], כאשר הסיפריות שלה משומשות עבור מכונות לומדות ומערכות לחישובים מדעיים. ממשקי תכנות יישומים (API) בסיפריות פייתון יוצאים מכלל שימוש בגלל שיפורי פיצ׳רים ותיקוני באגים בדיוק כמו בשפות אחרות.

שינוים אלה גורמות למפתחים לחוסר מוטיבציה להשתמש בפייתון עבור שימושים נוספים לפיתוחי תוכנה. בכל סיטואציה שבה מפתח צריך לזהות ולהחליף ממשקי תכנות יישומים (API), היא משימה מונוטונית וצורכת זמן יקר, כאשר בכל פרויקט קיים מספר רב של קריאות לממשקי תכנות יישומים (API). יתרה מזאת, התיעוד של ממשקים (במסמכים) גורם למשימות הללו להרבה יותר מסובכות ומאתגרות. הנושא הזה, לאורך שנים רבות היה בעיה רצינית עבור מפתחים.

בפרויקט זה, אני אציג כלי שבעזרתו יהיה ניתן להמנע מהבעיות הנ״ל וגם לחסוך המון שעות של עבודה של מפתחים

# רקע

## פייתון – Python

פייתון היא שפה דינאמית, קוד פתוח (open-source) חינמי, ומפוענח (interpreted) אשר בדרך כלל משומש לטובת בניית תוכנות, להפוך משימות לאוטומטיות, לבצע 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. הברסטיליות שלה והקלות שלה עבוד מפתחים צעירים, הפכו אותה לאחת השפות הכי פופולריות כיום [1].

פייתון תומכת בנוסף בתכנות מונחה עצמים וגם תכנות פרוצדורלי.

### תכנות דינאמי

תכנות דינאמי הוא ברובו אופטימיזציה מאשר רקורסיה.

בדומה לאלגוריתם ״הפרד ומשול״ (Divide and Conquer), תכנות דינאמי פותר בעיות על ידי שילוב של פתרונות עם תת פתרונות. אלגוריתמי הפרד ומשול מחלקים את הבעיה לתתי בעיות על מנת לפתור אותם בצורה רקורסיבית ורק לאחר מכן למזג בין הפתרונות שלהם ולפתור את הבעיה המקורית.

הרעיון הוא פשוט לשמור את התוצאות של תתי הבעיות בצורה כזו שאין צורך בחישוב שלהם מחדש לפעם הבעה שיהיה בהם צורך. אופטימיזציה כזו מפחיתה סיבוכיות זמן ריצה מזמן אקספוננציונלי לפולינומי.

## ממשק תכנות יישומים - API

ממשק תכנות יישומים הוא ערכה של ספריות קוד, פקודות, פונקציות, חוקים ופרוצדורות מוכנות, בהן יכולים המתכנתים לעשות שימוש פשוט, בלי להידרש לכתוב אותן בעצמם כדי שיוכלו להשתמש במידע של היישום שבו הם רוצים להשתמש לטובת היישום שלהם.

## תכנות דינאמי מול סטטי

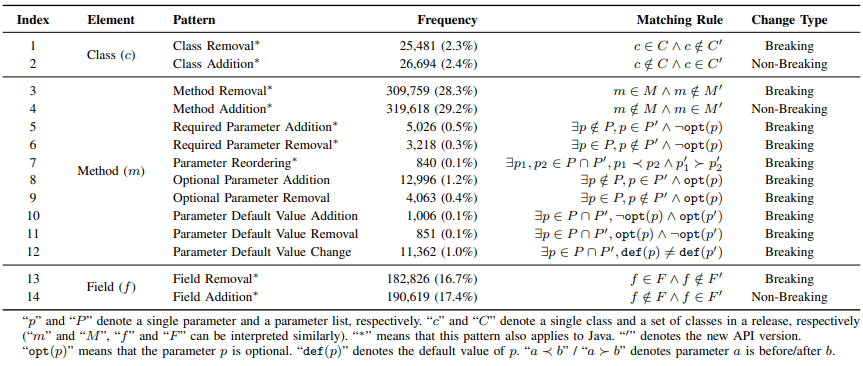
* שפות תכנות סטטיות מבצעות בדיקות טיפוסי משתנים בזמן ריצת קומפילציה, כאשר שפות תכנות דינאמיות מבצעות בדיקות טיפוסי משתנים בזמן ריצה.
* שפות תכנות דינאמיות (כמו - Lisp, Perl, Python, Ruby) מתוכננות לייעל את עבודת המפתח ולבצע אופטימיזציות לקוד כך שאותה פונקציונליות תמומש בפחות קוד. שפות תכנות סטטיות (כמו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

# שאלות מחקר

הדרך הטובה ביותר להבין התנהגות של שפת תכנות היא להשוות אותה לשפה מוכרת אחרת. מפה והלאה אני אשתמש ב-Java על מנת לעשות את ההשוואה ולהראות את דפוסי ההתנהגות והתנהלות של פייתון.

## איך סיפריות מבצעות דפרקציה?

כאשר מנתחים מאות גרסאות של סיפריות, כל גרסא יכולה להוביל לממשקי תכנות יישומים אחרים. להתחלה, נלקחו תשעה חוקי התאמה בסיסיים של ממשקי תכנות יישומים [2], ורוב השינויים תורגמו לתבניות הבאות:

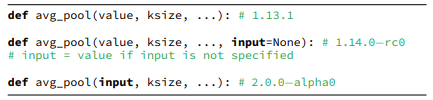


איור 1: תבניות שינויים של ממשקי תכנות יישומים

בסך הכל נמצאו 14 תבניות כאשר רק חמישה מהם לא נמצאו ב-Java [2].

ספריות פייתון מוצאות משימוש מסיבות שונות, אשר מסתכמות לשלוש קטגוריות [3]:

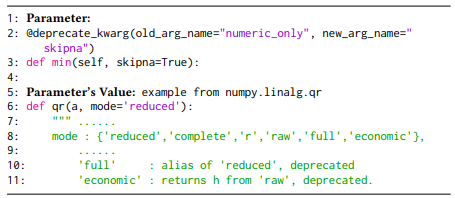
1. **מחלקה** או פונקציה שנמצאות בשימוש כאשר אינן נמצאות בסיפריה יותר, לדוגמא:



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

מחלקה של פייתון שהוצאה משימוש נמצאת בקבוצה ביחד עם פונקציות מאחר והיא דומה מאוד לקריאה לפונקציה (כמו קריאה לקונסטרקטור ב-Java).

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



איור 3: דפרקציה של חתימה של פונקציה

בזמן ששינויים בפרמטרים יכולים לקרות גם ב-Java, מקרים כמו באיור 2, לא יכולים לקרות. 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.

## איזה השפעות יכולות להיות לדפרקציות?

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

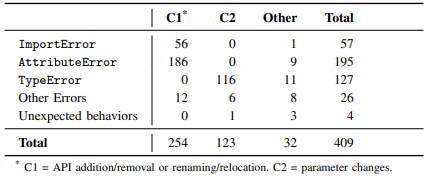
|  |  |
| --- | --- |
| **Library** | **Category** |
| TensorFlow | Deep Learning |
| Keras |
| Scikit-Learn | Data Analytics |
| Pandas |
| Flask | Web Development |
| Django |

טבלה 1: סיפריות למחקר

הסיפריות 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 ברישמה של מילים שחוזרים על עצמם כאשר מפתחים פותרים בעיות של סיפריות [4]:

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



איור 4: קורלציה בין סימפטום לסיבה

ניתן לראות שרוב מוחלט (405/409) מהמקרים גורמים לקריסות של המערכות בהם יש שימוש בסיפריות 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 [5]. 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 הללו. בשלב זה, מעניין לבחון איך ממשקים כאלו שהוצאו משימוש, באים לידי ביטוי בפרויקטי פייתון ובאיזה רמה מפתחים ניגשים לממשקים כאלה. [[1]](#footnote-1)

עבור כל שינויי הקוד בפרויקטי פייתון, ההודעות של שינויי הקוד הללו הכילו דפרקציה לעתים מאוד נדירות (פחות מ-0.1%) [3]. מפתחים בדרך כלל משתמשים באסטרטגיות הבאות בשביל לתקן את השגיאות הנ״ל (ראה איור 4). תבניות אלו מכסות רק 93% מהתיקונים, כל היתר כוללות תיקונים ספציפיים למקרה, כך שלא ניתן להפוך אותם לגנריים [6].

1. התאמת מגבלות הגרסה של תלויות ישירות. הקונפליקטים בין התלויות נקבעו, בדרך כלל, על ידי כוונון התלויות הישירות, כך שיהיו תואמות לאלו של התלויות הטרנזיטיבית.
2. שדרוג, או שנמוך, גרסאות של התלויות הישירות שנחוצות בספריות שתחת קונפליקט.
3. מחיקה של תלויות ישירות שתחת קונפליקט, תוך שמירה על אלה שטרנזיטיביות. יש צורך לישר קו בין התלויות הישירות לאלה הטרזיטיביות. 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 המקומים 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 מקומים לסביבה מרוחקת.

## מהו אופן הדוקומנטציה?

סיפריות פייתון משתמשות בשיטות ספציפיות על מנת להוציא משימוש ממשק תכנות יישומים מסוים, מה שגורם לצורך הרבה יותר גדול לדוקומנטציה הרבה יותר מדויקת ומסודרת 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 דוקומנטציה.

לכן, מנהלי סיפריות בפייתון תלויים בפתרונות ad-hoc, ובנוסף, לכתוב דוקומנטציה של אותם ממשקים. דבר זה בהרבה מקרים מוביל לטעויות.

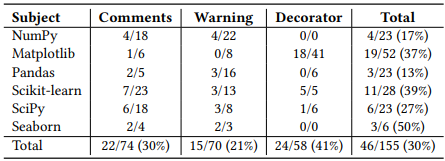
קיימים שלושה סוגים של דפרקציה – comments, הזהרות hard-coded, decorators. באיור הבא ניתן לראות שכל אחד מהם נמצא ביחסית אחוז מאוד קטן מהסיפריות [3].

Table

Description automatically generated

איור 5: ממשקי תכנות יישומים שהוצאו משימוש ללא דוקומנטציה

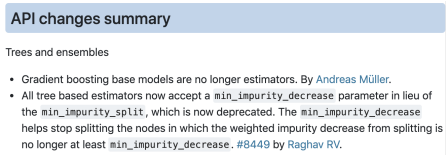
בסך הכל, קרוב ל-30% מתוך כל הממשקים, שיש להם דוקומנטציה, לא מספקים אלטרנטיבות למפתחים, מה שניתן לראות בתוצאות המאמר לגבי ממשקי תכנות יישומים של Java [7].



איור 6: ממשקי תכנות יישומים שהוצאו משימוש ללא אלטרנטיבות

להבין את הסיבה שממשק הוצא משימוש, עוזר למפתחים להגיע להחלטות מושכלות. 41% מממשקים כאלה שמוגדרים בעזרת decorator, לא מביאים אלטרנטיבות, שזה הרבה יותר משני סוגי הדפרקציה האחרים (comments, הזהרות hard-coded) [3].

במקרים מסוימים, יכול להיות מדובר בחוסר פירוט של הדפרקציה.



איור 7: דוגמא - ממש תכנות יישומים מוגדר בצורה לא ברורה

לאורך הדוקומנטציה של ממשקים שהוצאו משימוש, חלקם לא ברורים, ואפילו לא מתוארים בצורה נכונה. לדוגמא, איור 7 מציג דוקומנטציה לא ברורה של ממשק תכנות יישומים שהוצא משימוש, כאשר הפרמטר “min\_impurity\_split” הוצא משימוש, מהסיבה של “All tree based estimators are deprecated”, כאשר זה מתייחס לשבעה ממשקים שונים ולא רק לאחד.

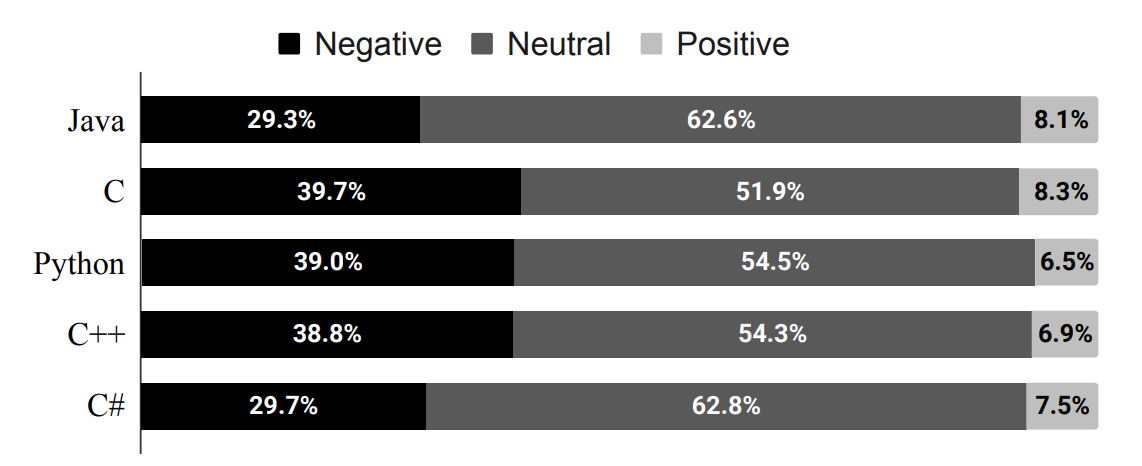
במקרה הזה, ייקח למפתחים הרבה יותר זמן להבין באיזה שינויים מדובר, ואיזה שינויים עליהם לעשות בפרויקט שלהם, על מנת לעבוד בצורה נכונה עם השינויים הללו.

קיים מספר גדול מאוד של ממשקים שהוצאו משימוש שלא נמצאים וגם לא מיוחסים בדוקומנטציה הרשמית של הסיפריות שלהם. מבחינת אלה שיש להם דוקומנטציה, לצערינו, לא לכולם יש אלטרנטיבה כלשהי שתעזור למפתחים להמנע משימוש של ממשקים שהוצאו משימוש.

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

## איך הקהילה מגיבה?

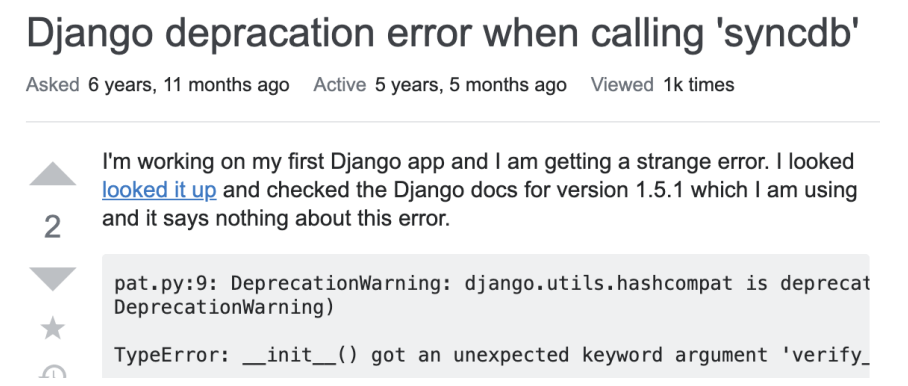
על מנת למצוא אמוציות של מפתחים כלפי הדיונים על דפרקציות, נעשה שימוש בכלי שנקרא Senti4D. מדובר בכלי שמנתח סנטימנטליות בתחום הפיתוח תוכנה [8]. תגובות של מפתחים מתחלקים לשלוש קטגוריות (למשל: שלילי, נייטרלי וחיובי). התוצאות של הניסוי הזה מציג את האיור הבא.



איור 8: התפלגות הסנטמנטליות של מפתחים בקרב חמשת השפות הפופולריות ביותר

ניתן לראות כי לפייתון יש את השיח החיובי הנמוך ביותר, בקרב השפות הפופולריות ביותר, בנוגע להוצאה משימוש של ממשקי תכנות יישומית. בנוסף, בנוגע לסנטימנט השלילי, לפייתון יש את הכמות כמעט הכי גבוהה מבין החמש. נתונים אלה מרמזים, שמפתחי פייתון, לא מרוצים עם צורת ההוצאה משימוש של הממשקים הנוכחית, בקרב הקהילה של פייתון [3].

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



איור 9: דוגמא - מפתח שמתלונן על צורת הדוקומנטציה

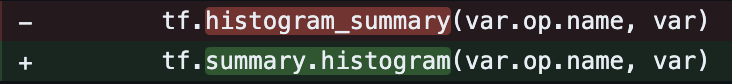
ככל שפייתון נהיית יותר פופולרית, נושאים כמו הוצאה משימוש של ממשקי תכנות יישומים מוזכרים הרבה יותר בדיונים באתרים כמו StackOverflow. כמו התגובות שניתן לראות בשפות תכנות אחרות, כך גם מפתחי פייתון מגיבים בצורה שלילית כאשר עולה הנושא של דפרקציה של ממשקים. לכן, יש צורך בלהבין את ההתנהגות של דפרקציות בעולם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

## 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 סיפריות, ניתן לראות שמפתחים לרוב מיישמים את ארבעת השיטות הבאות כאשר מתקנים בעיות של סיפריות שהוצאו משימוש:

### החלפה לממשק תכנות יישומים אחר

זה אולי התיקון הכי נפוץ, כאשר מפתחים מחליפים את הממשק עם הסיפרייה לממשק אחר (לרוב מאותה סיפריה), לדוגמא [9]:

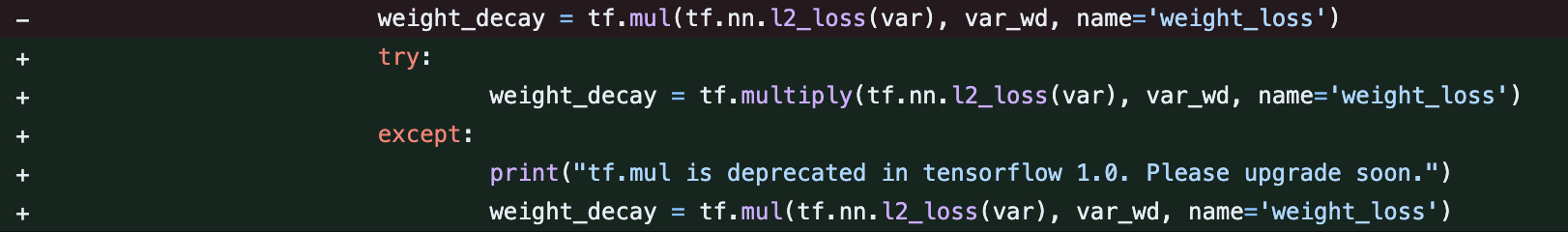


איור 10: תיקון בעיה על ידי החלפת ממשק

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

### 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) [10].



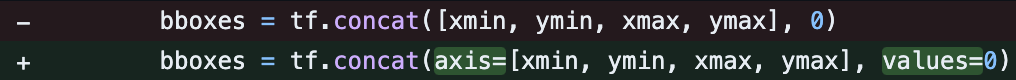
איור 11: תפיסת שגיאות

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 11), 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 באמת קורים אי התאמות חדשות.

### הוספה או שינוי מיקום או שם של פרמטרים

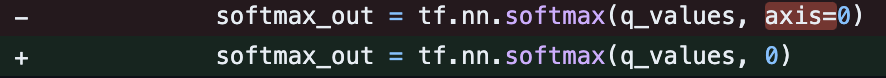
שינוים בחתימה של הפונקציה גורמים לשבירה של הפרויקט או להתנהגות לא צפויה, לכן, לעתים מפתחים בוחרים לשנות את הגישה מפרמטרים לפי מיקום לגישה של פרמטרים לפי keyword. לדוגמא [11]:



איור 12: שינוי לגישה של פרמטרים לפי keyword

הבעיה שגרמה לבעיה עם הממשק היא שינוי של המיקום של הפרמטרים, לכן, על ידי הוספה של ה-keyword של הפרמטרים לפני כל ערך, שומר על אחידות גם אם יהיו עוד שינויים דומים לזה בעתיד.

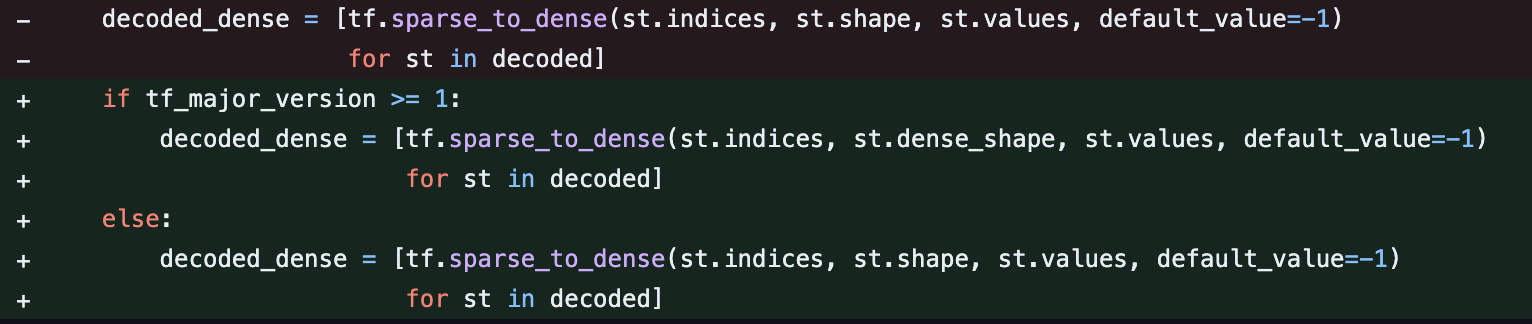
מה שפתרון זה לא פותר, הוא שינוי של שמות הפרמטרים. במידה ויש שינוי של שמות, הפתרון הקודם לא עוזר. לכן, במקרים מסוימים מפתחים בוחרים דווקא כן להשתמש בגישה של ערכים לפי מיקום בחתימה של הפונקציה. גישה כזו מאפשר תאימות לאחור [12].



איור 13: שינוי לגישה של פרמטרים לפי מיקום

### בדיקת גרסא של סיפריה

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



איור 14: בדיקת גרסא של סיפריה

בדוגמא הנ״ל ניתן לראות איך מפתחים בודקים אם הממשק שבו הם מתכוונים להשתמש קיים בעזרת בדיקה של הגרסא של הסיפריה. בדיקות מסוג כזה דורשות מהמפתחים להיות מודעים לשינויים שנעשים בסיפריות בהן הם משתמשים, כמו כן, לדעת את הגרסא המדויקת של אותה סיפריה 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 הסיפריה המדוברת.

# סיכום

בעבודה זו, ניתחנו כמה מאמרים על מנת להבין את השאלות הבאות:

1. מה השיטות לפתרון הבעיה?
2. האם יש חסרון בדוקומנטציה משמעותית עבור ממשקי תכנות יישומים?
3. איך מפתחי פייתון מגיבים לממשקי תכנות יישומים שהוצאו משימוש?
4. מהם התבניות שבהם ממשקי תכנות יישומים מתפתחים בפייתון?

המאמרים התמקדו בסיפריות מפורסמות של פייתון על מנת לענות על השאלות הללו. 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, נחסף שמפתחים שמים לב לעתים רחוקות לדפרקציה של ממשקים. ממצאים אלה, מצביעים בצורה חד משמעית שהקהילה חייבת לקחת על עצמה יצירה של טכניקות על מנת לעזור ליוצרים של הסיפריות וגם למפתחים להתמודד עם הוצאה משימוש של ממשקי תכנות משתמש.

על סמך המחקרים של דפרקציות, חייב להגיע מימוש של 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.

ישנם מגוון רחב של פתרונות שהוצאו על ידי הקהילה, אך אף אחד מהפתרונות הללו הוא לא קרוב לשלמות וגם לא פותר את הבעיה בשלמותה. פתרון מסוג כזה צריך להיות נוח למשתמש על מנת לעזור, כמה שאפשר, למפתחים בקהילה.

# ביביליוגרפיה

|  |  |
| --- | --- |
| [1] | "The top programming languages," GitHub, 2022. [Online]. Available: https://octoverse.github.com/2022/top-programming-languages. |
| [2] | X. Laerte, A. Brito, A. Hora and M. Tulio Valente, "Historical and Impact Analysis of API Breaking Changes: A Large-Scale Study," *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER),* pp. 138-147, 2017. |
| [3] | 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. |
| [4] | Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu and Y. Xiong, "How Do Python Framework APIs Evolve? An Exploratory Study," *2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER),* pp. 81-92, 2020. |
| [5] | 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, ASE 2016*, Singapore, 2016. |
| [6] | 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. |
| [7] | G. Brito, A. Hora, M. Tulio Valente ו R. Robbes, “On the use of replacement messages in API deprecation: An empirical study,” *Journal of Systems and Software,* pp. 306-321, 2018. |
| [8] | F. Calefato, F. Lanubile, F. Maiorano ו N. Novielli, “Sentiment polarity detection for software development,” *n Proceedings of the 40th International Conference on Software Engineering,* pp. 128-128, 2018. |
| [9] | wolframteetz/colorization.tensorflow, "GitHub," 8 June 2017. [Online]. Available: https://github.com/shekkizh/Colorization.tensorflow/commit/bec97c520d62cb88d4c2969f90bb51437494b1cc?diff=unified. |
| [10] | jparkhill/TensorMol, 2017. [Online]. Available: https://github.com/jparkhill/TensorMol/commit/b8d07db49d037048d7e312083d70bf25798b8ac8. |
| [11] | visipedia/tf\_classification, 2017. [Online]. Available: https://github.com/visipedia/tf\_classification/commit/df29c8c1b56ceef4c9d56f131aa3a6bd16134307. |
| [12] | sadeepj/crfasrnn\_keras, 2018. [Online]. Available: https://github.com/sadeepj/crfasrnn\_keras/commit/6bfaee73e4388a5b71b041f6d5949a3cf8989a62. |

1. נלקחו בחשבון הסיפריות מטבלה 1: סיפריות למחקר [↑](#footnote-ref-1)