**GIT**

**מה זה: Git?**

גיט היא מערכת לניהול גירסאות בקוד. היא פותחה בשנת 2005 בידי לינוס טורוולדס שהוביל בשעתו ומוביל עדיין פרויקט קוד פתוח ענק שהוא גרעין מערכת ההפעלה Linux. ללינוס נמאס ממערכת ניהול הגירסאות הקודמת בה השתמש וכמו מתכנת טוב הוא הלך לבנות משהו בעצמו.

כשלינוס בנה את גיט היה לו בראש צורך אחד מרכזי: היכולת לאפשר להמון מפתחים ברחבי העולם לעבוד במקביל על פרויקט ענק בלי שאף אחד ישבור את הקוד לאחרים. למרות שרוב האנשים שמשתמשים בגיט לא מנהלים פרויקטים בסדר גודל של ה Linux Kernel, מתכנתים עדיין בוחרים בגיט בגלל היציבות, המקביליות ואפקט העדר שהכלי מציע.

מבחינת יציבות גיט שומר את המידע בצורה מאוד חסינה לטעויות אנוש. מרגע שמידע נכנס למאגר גיט מאוד קשה למחוק אותו משם או לשבור את המבנה הפנימי של המאגר עצמו. כך גיט מבטיח שכל מי שעובד על הפרויקט יוכל להתקדם בצורה עצמאית ואף אחד לא צריך לפחד שמישהו ישבור או יעלים לו את הקוד.

מבחינת מקביליות גיט בנוי לעבודה מקבילית בה מאות אלפי מתכנתים ברחבי העולם עובדים על אותו פרויקט. הוא נכתב על ידי מתכנתים וכותביו התאמצו מאוד כדי שתוכלו לכתוב קוד מכל מקום ובכל זמן ועדיין יהיה לכם קל לשלב את הקוד שלכם בפרויקט הגדול. בפרויקטים קטנים תמיכה זו בעבודה מקבילית אומרת שיהיה לכם ממש קל לעבוד גם בלי שרת ואפילו בלי חיבור אינטרנט. ובעתיד כשהפרויקט יגדל יהיה קל להכניס מתכנתים נוספים לעזור.

מבחינת אפקט העדר יש לזכור שבדרך כלל כל המתכנתים שעובדים על פרויקט מסוים משתמשים באותו כלי לניהול גירסאות וכך גם כל המתכנתים בחברה מסוימת לרוב ירצו את אותו כלי לצורך נוחות תחזוקה. ככל שגיט הפך יותר פופולרי ונכנס בתור הבחירה הראשונה של מפתחי קוד פתוח הוא זחל משם גם לתעשיה הסגורה והיום גיט הוא אחד מכלי ניהול הגירסאות הפופולריים בעולם.

מה זה בכלל ניהול גירסאות?

מערכת ניהול גירסאות מכוונת לפתור מספר בעיות שמתכנתים נתקלים בהן בעבודה השוטפת. למתכנתים שעובדים בצוות קטן או לבד היתרונות כוללים:

1. מערכת ניהול גירסאות מספקת מעקב טוב אחר ההיסטוריה של הפרויקט- היא תאפשר לכם לשאול מאיפה קטע קוד מסוים הגיע והאם מספר קטעי קוד נכתבו באותו זמן וכחלק מאותו פיצ'ר. היא עוזר להיזכר בתקלות עבר ובאיך פתרתם אותם כדי ללמוד מזה לתקלות חדשות בהן אתם נתקלים.
2. מערכת ניהול גירסאות תתן לכם אפשרות לחזור אחורה בזמן לכל נקודה שהייתם בה ולהשוות את הפרויקט למצבו לפני שינוי גדול שנעשה או בתאריך מסוים.
3. מערכת ניהול גירסאות תאפשר לכם "לשמור" או לתייג גירסא מסוימת כך שתוכלו להיות בטוחים שאתם עובדים על אותו הקוד בדיוק גם אם יש לכם מספר מכונות (למשל השרת ומכונת הפיתוח).
4. מערכת ניהול גירסאות תאפשר לכם לעבוד במקביל על מספר פיצ'רים ולהעלות לשרת רק את אלה שמוכנים בלי שהפיצ'רים שנמצאים באמצע עבודה יידחפו ויגרמו לבעיות.

למתכנתים שעובדים בצוות יש עוד מספר יתרונות:

1. מערכת ניהול גירסאות תאפשר לזהות מי הכניס שינוי מסוים למערכת ומה היקף השינוי (עבור כל קטע קוד בתוכנית).
2. מערכת ניהול גירסאות תעזור למספר מתכנתים לעבוד על חלקים שונים בקוד ותחבר את התוצרים של שניהם לגירסא אחת משותפת. מערכות ניהול גירסאות מסוימות (אבל לא גיט) יוכלו גם למנוע ממתכנתים מסוימים לעבוד על קטעים מסוימים בקוד.
3. מערכת ניהול גירסאות עוזרת לתקשורת בין מתכנתים שכן היא קובעת אמת אחת, מצב אחד של הקוד ונותנת שם למצב זה. כך כולם יודעים שבגירסא 4.2.11 של המוצר יש יכולות מסוימות ותקלות מסוימות וקל יותר לבדוק מי מריץ כל דבר.

התקנת גיט

לצורך ההתקנה נוריד את Git for Windows מהקישור:

<https://git-scm.com/download/win>

בתהליך ההתקנה כדאי להוסיף את ה V על האפשרות של התקנת אייקונים לשולחן העבודה כי כך יהיה יותר נוח למצוא ולהפעיל את גיט אחר כך.

2. גיט למק

במק התקנת גיט היא כמעט אוטומטית ובמקרים רבים כבר יש לכם אותה כחלק ממערכת ההפעלה. פתחו מסוף וכתבו בשורת הפקודה:

$ git --version

אם גיט כבר מותקן תקבלו פלט שמציין איזה גירסת גיט מותקנת, ואם לא מותקן יופיע דיאלוג בו תוכלו ללחוץ Next כדי להתקין גיט בצורה אוטומטית דרך מערכת ההפעלה.

3. גיט ללינוקס

גיט קיים כמעט בכל מערכת לינוקס דרך מנהל החבילות של ההפצה. לדוגמא כדי להתקין גיט על Ubuntu עליכם לפתוח מסוף ולכתוב:

$ sudo apt install git-all

איך git עובד

גיט שומרת את כל גירסאות הפרויקט שלכם ומידע אודות כל גירסא במערכת קבצים נפרדת שיושבת ליד הפרויקט בתוך תיקיה נסתרת. בשיעור זה נראה איך המבנה הזה עובד ואיך הוא עוזר לנו בתהליך הפיתוח.

מחסן הגירסאות

דרך טובה להסתכל על גיט היא להיזכר במשחקי מחשב ישנים (או חדשים) בהם הרבה דברים רעים יכלו לקרות לאורך המשחק: יכולת למות או יותר גרוע להיתקע בדרך ללא מוצא. כשדברים כאלה קרו רציתם לחזור לנקודה כלשהי שבה הכל עוד היה בסדר ולנסות שוב. ברוב המקרים משחקים איפשרו לכם "לשמור" את המשחק שלכם ברגע מסוים ולטעון משחק שמור כדי להמשיך ממנו.

גיט עובדת די באותו האופן, רק שבמקום משחק יש לנו קוד.

בכל נקודה בתהליך הפיתוח אנחנו יכולים לשמור את קוד המערכת כולו בצד בתיקייה נסתרת שגיט מנהלת עבורנו. פקודת גיט אחת תשמור את הפרויקט ופקודת גיט אחרת תטען פרויקט שמור. בכל פעם שאנחנו שומרים פרויקט נוכל להוסיף לנקודת השמירה תיאור מילולי קצר שיעזור לנו להיזכר מה היה מצבנו באותה נקודת זמן.

ליד הפרויקט שלנו יש תיקייה בשם **.git** בה גיט שומר את העותקים השמורים של הפרויקט. בצד שלנו, בכל פעם שאנחנו עושים שינוי בקוד נרצה לשמור את הקוד בתוך אותה תיקיית **.git**. לאורך זמן יצטברו הרבה גירסאות של הפרויקט ומגירסאות אלה נוכל לבנות סיפור של מה קרה בפרויקט ואיך הגענו לנקודה בה אנחנו נמצאים היום.

גיט מספק יכולות מתקדמות של צפיה וחיפוש בגירסאות ישנות: עבור כל גירסא נוכל לראות איזה קבצים השתנו בה לעומת הגירסא שלפניה (או גירסא אחרת שלשהי). או בהינתן שורת קוד מסוימת נוכל לדעת מאיזה גירסא היא הגיעה ואיזה עוד שורות קוד נכתבו בסמוך אליה.

2. שיתוף פעולה

גיט היא מערכת מבוזרת המאפשרת שיתוף פעולה בהיקף עצום בין מתכנתים המפוזרים ברחבי העולם. היא מצליחה לעשות את זה בזכות המבנה המבוזר של המערכת: לכל מתכנת שלוקח חלק בפרויקט יהיה עותק משלו של "מחסן הגירסאות" של גיט, כלומר של התיקיה **.git**. בנוסף, יהיה עותק אחד משותף של התיקיה אליו מתכנתים יכולים לשלוח שינויים וממנו הם יכולים למשוך את השינויים של האחרים.

בדרך כלל לפני תחילת עבודה מתכנתים ימשכו את השינויים של חבריהם לפרויקט ובסיום העבודה ידחפו את השינויים שהם ביצעו לאותו מאגר משותף.

השילוב בין מאגר מקומי למאגר משותף נותן המון כח לגיט: מצד אחד אנחנו יכולים להתקדם בפרויקט ולעבוד רגיל לגמרי גם כשאנחנו לא מחוברים לרשת, או לעבוד על כיוונים שלנו שאולי אחרים לא יודעים מהם. מצד שני כשצריך לשתף פעולה יש לנו מנגנון פשוט בעזרתו אנחנו שולחים קוד לאחרים והם משלבים אותו בקוד שלהם.

יצירת פרויקט בתוך סביבת פיתוח

לצורך הדוגמא ארצה להראות איך גיט נראה מתוך סביבת הפיתוח ומכלי שורת הפקודה וקצת על היכולות של פיתוח פרויקט המנוהל בגיט.

אני כאן אשתמש בסביבת פיתוח שנקראת PyCharm שהיא סביבה לפיתוח בשפת Python. מפתחים בשפות אחרות יכולים להשתמש בסביבות דומות של אותה חברה כדי לקבל ממשק שנראה כמעט אותו דבר (החברה נקראת Jetbrains ויש להם גם את IntelliJ ל Java ואת Webstorm לעולם ה Web). סביבות פיתוח אחרות תומכות ברצפי עבודה מאוד דומים אז גם אם אתם רגילים ל IDE אחר אני בטוח שתמצאו את הכפתורים.

נתחיל בהתקנת Python מהקישור:

<https://www.python.org/downloads/windows/>

נמשיך בהתקנת PyCharm מהקישור:

<https://www.jetbrains.com/pycharm/download/>

וכעת ניצור פרויקט חדש ונוסיף אליו מספר קבצים, לדוגמא נכתוב את הקובץ main.py עם התוכן הבא:

print("Hello World")

בנוסף אליו PyCharm יוצרת ומוסיפה אוטומטית תיקיה בשם venv. נשאיר אותה כאן לבינתיים.

באופן רגיל ובלי מערכת ניהול גירסאות אפשר בכיף לשנות את הקבצים מתוך סביבת הפיתוח אבל אין לנו תיעוד של מה שינינו ואין דרך לחזור למצב שהיה לפני השינוי. מפתחים רבים משתמשים ב Dropbox או מערכות גיבוי דומות כדי לנהל גירסאות ולגבות את הקוד וזה נותן פיתרון חלקי. בואו נוסיף את גיט לפרויקט ונתחיל לשלב אותו ולבנות פיתרון מקיף יותר.

2. הוספת גיט לפרויקט משורת הפקודה

נפתח מסוף לשורת הפקודה ונכנס לתיקיית הפרויקט. בתוך התיקיה נכתוב:

$ git init

הפקודה יוצרת תיקיה חדשה בשם **.git** בתוך תיקיית הפרויקט שלכם. גיט יענה לנו בפלט:

Initialized empty Git repository in C:/Users/IEUser/PycharmProjects/helloworld/.git/

מכלי שורת הפקודה כתבו את הפקודה:

$ ls -a

ותוכלו לראות את כל הקבצים והתיקיות בתיקיה הנוכחית ושם התווספה התיקיה החדשה **.git**. כך זה נראה אצלי:

$ ls -a

./ ../ .git/ .idea/ main.py venv/

בתוך התיקיה **.git** יישמרו כל הגירסאות של כל הקבצים שלכם לאורך חיי הפרויקט. זו בעצם תיקיית ההיסטוריה של הפרויקט שלכם.

3. הקומיט הראשון שלי

עכשיו בשביל הניסוי בואו נשמור את הגירסא הנוכחית של הפרויקט בצד. בגיט שמירה בצד כזו מכונה commit ולכן ניצור את הקומיט הראשון שלנו.

קומיט בגיט מורכב מאוסף של קבצים ותיקיות שיכולים להיות כל קבצי הפרויקט או רק חלק ממנו. הפקודות הבאות יצרפו את כל קבצי הפרויקט לקומיט הראשון ואז ישמרו את הקומיט הזה בתיקיית ההיסטוריה:

$ git add .

הפקודה git add גם מדפיסה מספר הודעות שגיאה על המסך:

warning: LF will be replaced by CRLF in venv/Lib/tcl8.6/init.tcl.

The file will have its original line endings in your working directory.

הודעות שגיאה אלה מתיחסות לקבצים בתוך תיקיית venv ומספרות לנו שקבצים מסוימים מסתיימים בתו סיום שורה של יוניקס למרות שאנחנו על מערכת Windows. גיט ערוך להתמודד עם מתכנתים שונים שעובדים על אותו פרויקט ממערכות הפעלה שונות וכדי שהמנגנון הזה יעבוד הוא משנה את כל תווי סיום השורה לפורמט אחיד. בהמשך הקורס נראה כיצד לשלוט בהתנהגות זו.

בינתיים אפשר להתעלם מהאזהרה ולהמשיך לפקודה הבאה:

$ git commit -m 'initial commit'

שימו לב לטקסט initial commit: טקסט זה מתאר את הגירסא וילווה אותה בתוך מאגר ההיסטוריה לנצח נצחים (או לפחות עד שתמחקו את מאגר ההיסטוריה).

גם כאן גיט עונה לנו באזהרה והפעם אזהרה שאי אפשר להתעלם ממנה:

\*\*\* Please tell me who you are.

Run

git config --global user.email "shay@example.com"

git config --global user.name "Your Name"

to set your account's default identity.

Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'IEUser@IE11WIN8\_1.(none)')

בשביל לשמור קבצים בהיסטוריה עלינו תחילה להזדהות. אל תדאגו לא צריך כאן דרכון או שום דבר פשוט להגיד מי אתם ומה המייל שלכם כדי שיישמר יחד עם השינוי. נעקוב אחר ההוראות של גיט ונכתוב (החליפו את הפרטים באלו שלכם):

$ git config --global user.email "shay@tocode.co.il"

$ git config --global user.name "shay"

וכעת הפעילו שוב את פקודה השמירה:

$ git commit -m 'initial commit'

עכשיו הכל עבד והפלט שקיבלנו מציג את כל הקבצים שנשמרו בצד. בפעמים הבאות גיט כבר יזכור את השם והאימייל שהכנסתם ובשביל לשמור גירסא נוספת תצטרכו להשתמש רק ב add ו commit. בואו נראה את זה רגע.

4. הקומיט השני שלי

נחזור ל PyCharm כדי לגלות ש PyCharm מספיק חכם לזהות שהוספנו את גיט לפרויקט. הוא שואל אותנו אם להוסיף באופן אוטומטי קבצים מסוימים לגיט. אנחנו נגיד שלא וגם נסמן את התיבה Remember, don't check again. אני מעדיף להוסיף את כל הקבצים בצורה ידנית.

בחזרה לקובץ main.py ובואו נעדכן אותו ונוסיף שם פונקציה כך שתוכן הקובץ נראה כך:

def hello\_text():

return "Hello World"

print(hello\_text())

עכשיו קרה דבר מעניין: כבר בתוך PyCharm אנחנו רואים ששם הקובץ main.py שינה את הצבע לתכלת כדי לסמן לנו שהקובץ עודכן מאז הגירסא האחרונה ששמורה בגיט.

נחזור לשורת הפקודה והפעם מספיק להשתמש בשתי הפקודות הבאות כדי לשמור את הגירסא העדכנית:

$ git add .

$ git commit -m 'ADD: move hello text to its own function'

עכשיו הצבע ב PyCharm חזר ללבן וכך אנחנו יודעים שכל הנתונים נשמרו בצד. לחצו כפתור ימני על הפרויקט ובחרו Git ואז Show History. בתחתית המסך ייפתח חלון שמראה את שני הקומיטים ששמרנו בצד עם הטקסט שכתבנו ליד כל קומיט. עכשיו אתם יכולים ללחוץ על הקומיט ולבחור Compare With Local ואז ייפתח לכם חלון שמראה את כל הקבצים שהשתנו בין הגירסא שבחרתם לגירסא שעכשיו יש לכם בתיקיה. לחיצה כפולה על אחד הקבצים תראה ממש את ההבדלים.

בשיעורים הבאים נראה איך לראות את כל המידע הזה גם לא מתוך הכלי PyCharm כך שתוכלו להגיע אליו לא משנה באיזה IDE אתם עובדים.

5. קומיט שלישי ואחרון

כמעט כל סביבות הפיתוח היום תומכות בגיט תמיכה מלאה וכמובן גם PyCharm. היכולת לראות את ההיסטוריה היא מאוד נחמדה אבל אם אנחנו כבר עובדים בתוך IDE מסוים כנראה שלא נרצה לעזוב אותו רק בשביל לבצע commit. עדכנו את הקובץ main.py פעם שלישית והפעם לחצו כפתור ימני על שם הקובץ, בחרו git ואז commit file. בתוך PyCharm יהיה לכם מקום לכתוב את ההודעה שתלווה את הקומיט ועוד מספר אופציות עליהן נדבר בהמשך.

כתבו איזה טקסט שאתם רוצים וסיימו את הקומיט. וודאו באמצעות צפיה בהיסטוריה שאכן יש לכם שלושה קומיטים שמורים בצד .

Staging Area

נדבר יותר לעומק על ההבדל בין add ל commit ונראה מה באמת עושה הפקודה git add ולמה צריך את שתי הפקודות.

ה Staging Area של גיט

נחזור לפרויקט שאנחנו עובדים עליו ונבצע שני שינויים: נעדכן את הקובץ main.py באמצעות שינוי הטקסט ונוסיף קובץ חדש בשם utils.py. בקובץ main.py נכתוב:

def hello\_text():

return "Hello World! -- v2"

print(hello\_text())

ובקובץ החדש utils.py נכתוב:

def twice(x):

return x \* 2

משורת הפקודה כתבו את הפקודה:

$ git status

כך נראה הפלט אצלי:

$ git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: .idea/workspace.xml

modified: main.py

Untracked files:

(use "git add <file>..." to include in what will be committed)

utils.py

no changes added to commit (use "git add" and/or "git commit -a")

הפקודה **git status** מציגה לנו בשורת הפקודה את מצב העניינים מבחינת איזה קבצים גיט מכיר ואיזה קובץ נמצא באיזה מקום. ראינו שגיט שומר את כל הקומיטים בתוך תיקיית **.git** כך שכל קומיט מייצג את מצב הפרויקט ברגע נתון. בנוסף לאזור זה גיט שומר גם את כל הדברים שאתם מתלבטים אם לעשות להם קומיט או מתכננים לעשות להם קומיט אבל עדיין לא הספקתם באזור שנקרא Staging.

הפקודה git add מוסיפה גירסא מסוימת של קובץ לאזור ה Staging והפקודה git reset מוציאה אותו משם. ניקח את השינוי שלנו בקובץ main.py ונוסיף את הקובץ main.py לאחר השינוי לאזור ה Staging:

$ git add main.py

אני יכול לוודא שהקובץ נוסף באמצעות הקלדת git status פעם נוספת:

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: main.py

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: .idea/workspace.xml

Untracked files:

(use "git add <file>..." to include in what will be committed)

utils.py

הקובץ main.py מופיע כעת ברשימה העליונה: Changes to be committed. יותר מזה אני יכול לראות ממש איזה גירסא של main.py שמורה באזור ה Staging שלי באמצעות הפקודה:

$ git ls-files --stage -- main.py

הפלט אצלי על המחשב נראה כך:

100644 1868d11237a54a6ed8140d68e7953ca52b303017 0 main.py

כאשר המספר הארוך בעמודה השניה הוא מזהה של הקובץ בגירסא שלו ששמורה כרגע ב Staging Area. אפשר לראות גם את תוכן הקובץ ששמור שם עם הפקודה:

$ git cat-file blob 1868d11237a54a6ed8140d68e7953ca52b303017

def hello\_text():

return "Hello World! -- v2"

print(hello\_text())

הפקודה **git add** מוסיפה קבצים לאזור "הכנה לשמירה". אזור זה שמור גם הוא בתיקיה **.git** אבל מטבעו הוא זמני: קבצים יכולים להיכנס ולצאת מאזור זה והשמירה הקבועה שלהם תקרה רק באמצעות commit.

נוסיף קובץ נוסף לרשימה הפעם את הקובץ החדש שיצרנו utils.py:

$ git add utils.py

ונראה מה מצב הפרויקט אחרי הוספת שני הקבצים:

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: main.py

new file: utils.py

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: .idea/workspace.xml

קיצור הדרך שראינו בשיעורים קודמים:

$ git add .

מוסיף לאזור ה Staging את כל הקבצים והתיקיות מהתיקיה הנוכחית בצורה רקורסיבית. הפעילו אותו ולאחר מכן שוב הציגו את הסטטוס:

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: .idea/workspace.xml

modified: main.py

new file: utils.py

2. שימוש נכון ב Staging Area

ראינו כבר שמתוך סביבת הפיתוח אפשר לראות את כל הקומיטים השמורים בתיקיית ההיסטוריה וגם להסתכל על ההבדלים ביניהם או בינם לבין הגירסא הנוכחית של הפרויקט. לכן יש חשיבות גדולה לשאלה איזה קבצים נמצאים בקומיט מסוים ולהפרדה נכונה בין קומיטים. ככל שקומיט מייצג פיצ'ר מסוים או התנהגות מסוימת סגורה יהיה קל יותר להסתכל עליו ולהבין את הקשר בין הקבצים השונים בו.

הבחירה של גיט ביצירת ה Staging Area היא להגדיר קומיט בתור רשימת שינויים שיכולה להתפרס על פני מספר קבצים. הטקסט שאנו מצרפים לקומיט אמור לעזור לנו להיזכר במה שינוי מסוים עוסק ולכן כדאי מאוד שקומיט יחיד יכיל רק את הקבצים שקשורים לאותו הנושא.

נעדכן שוב את הקבצים בפרויקט כדי ש main.py ישתמש בקובץ utils.py החדש שיצרנו. הזינו את התוכן הבא לקובץ main.py (זיכרו עדיין לא עשינו קומיט):

import utils

def hello\_text():

return "Hello World! -- v2"

print(hello\_text())

print(utils.twice(5))

עכשיו נרצה לבצע קומיט לקובץ main.py החדש ולקובץ utils.py מאחר ויש קשר ביניהם. ניזכר מה המצב בתיקיה כרגע:

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: .idea/workspace.xml

modified: main.py

new file: utils.py

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: .idea/workspace.xml

modified: main.py

Untracked files:

(use "git add <file>..." to include in what will be committed)

\_\_pycache\_\_/

הקבצים main.py ו workspace.xml נמצאים בשתי רשימות: גם ברשימת השינויים לשמירה בקומיט וגם ברשימת השינויים שאינם מיועדים לקומיט. הסיבה היא ששמרנו כבר ב Staging Area את הקבצים האלה ואחרי זה ביצענו בהם עוד שינויים (את השינויים ב workspace.xml הכלי PyCharm עשה אוטומטית). נעדכן תחילה את הגירסא ב Staging כך שתהיה העדכנית ביותר של הקבצים שמעניינים אותנו:

$ git add main.py utils.py

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: .idea/workspace.xml

modified: main.py

new file: utils.py

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: .idea/workspace.xml

Untracked files:

(use "git add <file>..." to include in what will be committed)

\_\_pycache\_\_/

עכשיו מתוך הרשימה הראשונה הקבצים main.py ו utils.py הם אכן הקבצים שאני רוצה לשמור, אבל הקובץ workspace.xml לא ממש רלוונטי לשינוי הקוד שלי ונכנס לשם בטעות בגלל שביצעתי **git add .**. בואו נוציא אותו מה Staging Area:

$ git reset .idea/workspace.xml

Unstaged changes after reset:

M .idea/workspace.xml

הפקודה **git reset** עושה הרבה מאוד דברים בגיט, אבל בשימוש הפשוט שלה היא מוציאה קובץ מה Staging Area כך שהקומיט הבא לא יכלול אותו.

צפיה ב **git status** מראה לנו שאנחנו עכשיו בכיוון הנכון:

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: main.py

new file: utils.py

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: .idea/workspace.xml

Untracked files:

(use "git add <file>..." to include in what will be committed)

\_\_pycache\_\_/

עכשיו אפשר להמשיך ולשמור את ה Staging Area בתור קומיט חדש עם:

$ git commit -m 'ADD utils.py and using it from main.py'

אני משאיר השינויים בקבצים האחרים מחוץ לרשימת השינויים שלי. האמת שהייתי מעדיף שהתיקיה **.idea** בכלל לא תישמר במערכת ההיסטוריה שלי כי ממילא יש בה כל הזמן שינויים אוטומטיים. בשיעור הבא נדבר על מחיקות ונראה איך להוציא תיקייה זו מניהול הגירסאות.

קומיט מחיקה

הפקודה **git rm** מבצעת שני דברים:

1. היא מוחקת קובץ מהתיקיה הנוכחית
2. מעדכנת את האינדקס ב Staging Area שהקובץ הוא כבר לא חלק מהפרויקט

בואו נראה את זה בפעולה בדוגמא על הפרויקט שלנו. נזכיר שהסטטוס בו עצרנו אחרי השיעור הקודם נראה כך:

$ git status

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: .idea/workspace.xml

Untracked files:

(use "git add <file>..." to include in what will be committed)

\_\_pycache\_\_/

no changes added to commit (use "git add" and/or "git commit -a")

ובתיקיה יש לנו שני קבצי פייתון בשמות **main.py** ו **utils.py**.

נתחיל במחיקת הקובץ **utils.py** והעברת הפונקציה שמוגדרת בו ל main. עדכנו את תוכן הקובץ main.py שיכלול את הקוד הבא:

def twice(x):

return x \* 2

def hello\_text():

return "Hello World! -- v2"

print(hello\_text())

print(twice(5))

עכשיו main.py לא משתמש יותר ב utils.py ולכן אפשר למחוק את utils. אבל תחילה בואו נעיף מבט מה גיט שומר כשאנחנו מייצרים קומיט.

חזרו לשורת הפקודה וכתבו **git log**. פקודה זו מציגה את כל ההיסטוריה של הפרויקט:

$ git log

$ git log

commit cc62d6140975e1ab806c3597b47e410b2daed37e (HEAD -> master)

Author: shay <shay@tocode.co.il>

Date: Thu Aug 16 03:36:31 2018 -0700

ADD utils.py and using it from main.py

commit 611ef8ae1927f51dfb03ef2a4fef0a135bf76023

Author: shay <shay@tocode.co.il>

Date: Thu Aug 16 02:56:13 2018 -0700

Yay I saved the world

commit b656e9ffe33792e4ea1cfb8e4bc0d44c3f5f6f07

Author: shay <shay@tocode.co.il>

Date: Thu Aug 16 02:48:06 2018 -0700

ADD: move hello text to its own function

commit 10966a66b9a80f9fc3b55fca4bcfe172b40bd7bd

Author: ynon <shay@tocode.co.il>

Date: Thu Aug 16 02:43:28 2018 -0700

initial commit

כל קומיט בגיט מכיל מזהה. המזהה של הקומיט הישן ביותר הוא: 10966a66b9a80f9fc3b55fca4bcfe172b40bd7bd ושל החדש ביותר הוא: cc62d6140975e1ab806c3597b47e410b2daed37e. אצלכם על המחשב מזהים אלה כנראה יהיו קצת שונים. המזהה הוא ערך SHA1 של כל הקבצים והתיקיות בפרויקט בנקודת הזמן שהקומיט נלקח יחד עם מידע מנהלתי שכולל מי עשה קומיט ומתי. לכן מזהה זה הוא ייחודי לכל קומיט.

באמצעות המזהה אפשר להציג את תוכן הקומיט עם הפקודה **git cat-file**. נתבונן בקומיט הישן ביותר בפרויקט (זכרו בשביל להריץ אצלכם תצטרכו למצוא את מזהה הקומיט הנכון בפרויקט שלכם):

$ git cat-file commit 10966a66b9a80f9fc3b55fca4bcfe172b40bd7bd

והאמת שבדרך כלל גיט יודע להשלים לבד את ה SHA1 המלא ולכן מספיק לכתוב רק את ה 5 אותיות הראשונות של המזהה:

$ git cat-file commit 10966

tree 492fe5710b7d5b740a67f1af9932dc345d322d45

author ynon <ynon@tocode.co.il> 1534412608 -0700

committer ynon <ynon@tocode.co.il> 1534412608 -0700

initial commit

השורה הראשונה של הקומיט מציגה את מזהה עץ הקבצים שקומיט זה מכיל. זכרו שכל קומיט מכיל את תוכן הפרויקט המלא בנקודת זמן נתונה. נתבונן בעץ הקבצים באמצעות פקודה נוספת **git ls-tree**:

$ git ls-tree 492fe

040000 tree b307e3dcbe6ce405fffac6aedec2b8f21390f8c4 .idea

100644 blob 8e235769c023ae8b5dad489734aa4ac26ad526de main.py

040000 tree 1df55c1617b4b51498e8a12ec07faaa646e7edda venv

מסתבר שבקומיט הראשון שביצענו במערכת שמרנו בצד 3 דברים: הקובץ **main.py** ושני עצי קבצים (כלומר תיקיות) בשמות **.idea** ו **venv**. אגב בכל עץ קבצים כזה יש קבצים משלו ואפשר להציג אותם גם באמצעות **git ls-tree** עם המזהה של העץ המתאים.

לכן כל קומיט שומר בצד את כל הקבצים ששייכים לקומיט זה ובנוסף את העצים שמתארים את מבנה הפרויקט ברגע שבוצע הקומיט. קומיט של הוספת קובץ כולל גם את הקובץ וגם את עץ הקבצים עם שם הקובץ והמזהה המתאים. קומיט שמוחק קובץ כולל גם הוא עץ קבצים אבל הפעם הקובץ שאנחנו מוחקים פשוט לא יופיע בעץ.

בואו נראה את זה בעת מחיקת הקובץ utils.py שלנו. כתבו את הפקודות הבאות כדי לשמור ב Staging Area את הקובץ main.py החדש ואת עץ הקבצים שלא כולל את utils.py ואז ולאחר מכן לבצע קומיט:

$ git add main.py

$ git rm utils.py

$ git commit -m 'moved twice back to main'

נתבונן בשני הקומיטים האחרונים כדי לראות את רשימות הקבצים בכל אחד. ראינו כבר שאפשר לזהות קומיט לפי ה Hash שלו אבל מסתבר שהרבה פעמים יותר נוח לזהות קומיט לפי המיקום היחסי שלו בעץ. לדוגמא המזהה **HEAD~1** מתיחס לקומיט הקודם ו **HEAD~2** לזה שלפניו וכך הלאה. גם המזהה **HEAD^** מתיחס לקומיט הקודם ו **HEAD^^** לזה שלפניו.

בשביל להציג את הקומיט האחרון נשתמש ב:

$ git cat-file commit HEAD

tree 1b90de704f4539da54df4c6f16d736b734026c36

parent cc62d6140975e1ab806c3597b47e410b2daed37e

author ynon <ynon@tocode.co.il> 1534422351 -0700

committer ynon <ynon@tocode.co.il> 1534422351 -0700

moved twice to main

ובשביל להציג את הקומיט שלפניו נוכל לכתוב:

$ git cat-file commit HEAD^

tree 1662d70fb44e6d513856b8fa6ffe876539ee06e1

parent 611ef8ae1927f51dfb03ef2a4fef0a135bf76023

author ynon <ynon@tocode.co.il> 1534415791 -0700

committer ynon <ynon@tocode.co.il> 1534415791 -0700

ADD utils.py and using it from main.py

בשביל להציג את העצים המתאימים נשתמש ב **git ls-tree** על שני מזהי העצים:

$ git ls-tree 1b90d

040000 tree bc6bc4db54d9111b6e3ebe8dd6e9fd77725dd184 .idea

100644 blob 2035504a30b30a0c85ca7a037f2187d84d8753b5 main.py

040000 tree 1df55c1617b4b51498e8a12ec07faaa646e7edda venv

$ git ls-tree 1662d

040000 tree bc6bc4db54d9111b6e3ebe8dd6e9fd77725dd184 .idea

100644 blob 3a9428314e5e87a695019063b90e3f15689c868d main.py

100644 blob 931e58188dc815ef7e3d4d0f267b60290dd395c1 utils.py

040000 tree 1df55c1617b4b51498e8a12ec07faaa646e7edda venv

קל לראות שהקובץ utils.py מופיע בקומיט הישן יותר אך לא בקומיט החדש יותר, ולכן בגירסא 1b90d הקובץ כבר לא קיים. זכרו שהקובץ עצמו עדיין נשמר בהיסטוריה של גיט ואפשר לראות אותו במגוון דרכים. דרך אחת קלה היא להשתמש במזהה הקובץ ישירות ולכתוב:

$ git cat-file blob 931e5

def twice(x):

return x \* 2

בשיעורים הבאים נלמד דרכים נוספות.

2. קומיט עם שינוי שם

הפקודה **git mv** משנה שם של קובץ בתיקיית העבודה וגם מכינה קומיט חדש ב Staging Area בו שם הקובץ מתעדכן לשם החדש. לצורך הדוגמא נשנה את השם של הקובץ main.py ל demo.py:

$ git mv main.py demo.py

$ git commit -m 'renamed file main'

[master be251ba] renamed file main

1 file changed, 0 insertions(+), 0 deletions(-)

rename main.py => demo.py (100%)

3. יצירת קומיט מחיקה בלי למחוק את הקבצים

התיקיה **.idea** שנמצאת אצלנו בתיקיית הפרויקט היא דוקא שימושית: הכלי PyCharm נעזר בה כדי לנהל את מבנה הפרויקט והוא צריך שהתיקיה תישאר שם כדי לשמור כל מיני מידע מנהלתי על הפרויקט. אבל התיקיה אינה חלק מהקוד ואם אנחנו עובדים בצוות עם אנשים שכותבים ב IDE אחר התיקיה סתם תתפוס להם מקום ותפריע בעין.

לא מאוחר מדי למחוק את התיקיה מהיסטוריית הפרויקט - אבל בניגוד ל **utils.py** את התיקיה **.idea** כן נרצה לשמור על המכונה שלנו. הפרמטר **--cached** אותו אפשר להעביר לפקודה **git rm** מאפשר למחוק תיקיה רק מהיסטורית הפרויקט בלי להשפיע על מבנה הקבצים בתיקיה הנוכחית אצלנו. זה עובד ככה:

$ git rm -r --cached .idea

rm '.idea/helloworld.iml'

rm '.idea/misc.xml'

rm '.idea/modules.xml'

rm '.idea/vcs.xml'

rm '.idea/workspace.xml'

המתג **-r** גורם למחיקה של עץ התיקיות יחד עם כל הקבצים והתיקיות שבתוכו. שימו לב שתיקיות אלה עדיין נמצאות אצלכם בתיקיית העבודה:

$ ls .idea

helloworld.iml misc.xml modules.xml vcs.xml workspace.xml

ניצור קומיט חדש בלי תיקיית **.idea** עם:

$ git commit -m 'removed .idea folder from the repository'

[master 1272f32] removed .idea folder from the repository

5 files changed, 189 deletions(-)

delete mode 100644 .idea/helloworld.iml

delete mode 100644 .idea/misc.xml

delete mode 100644 .idea/modules.xml

delete mode 100644 .idea/vcs.xml

delete mode 100644 .idea/workspace.xml

וכעת אפשר לוודא שהתיקיה עברה למצב Untracked כלומר שגיט לא מנהל גירסאות על הקבצים שבתוכה:

$ git status

On branch master

Untracked files:

(use "git add <file>..." to include in what will be committed)

.idea/

\_\_pycache\_\_/

nothing added to commit but untracked files present (use "git add" to track)

שמירת היסטוריה של כל צעד בפרויקט מאפשרת לנו להיזכר איך נראו הקבצים בעבר וגם לקחת קבצים מהעבר חזרה להווה. ראינו כבר מספר דרכים לעשות זאת ובשיעור זה נדבר על הפקודה checkout ותפקידה בחזרה בזמן.

מעבר לגירסא ישנה של המערכת

הדרך הכי קלה לנסוע בזמן היא להשתמש בפקודה **git checkout**. פקודה זו מקבלת מזהה של קומיט (או בראנצ' אבל עליהם נדבר בעוד מספר שיעורים) ופשוט מחזירה אתכם לקומיט זה. נישאר עם הפרויקט שלנו ובואו נחזור יחד לימים שהיה קובץ utils.py. תחילה נזהה באיזה קומיט קובץ זה היה קיים באמצעות הלוג.

אני מזכיר שצפיה בלוג הרגיל אפשרית עם

$ git log

אבל הפרויקט כבר מתחיל להסתבך והלוג הזה נהיה ארוך. אנחנו ממילא לא מחפשים את כל הקומיטים אלא רק את אלה שהשפיעו על הקובץ utils.py ולכן נוכל לכתוב:

$ git log --follow -- utils.py

commit 31f3859a60bf02359d623d8f284da9ee4c12488a

Author: shay <shay@tocode.co.il>

Date: Thu Aug 16 05:25:51 2018 -0700

moved twice to main

commit cc62d6140975e1ab806c3597b47e410b2daed37e

Author: shay <shay@tocode.co.il>

Date: Thu Aug 16 03:36:31 2018 -0700

ADD utils.py and using it from main.py

אם גם הפלט הזה היה ארוך מדי אפשר להגביל את מספר התוצאות שלוג מדפיס באמצעות הפרמטר **-n** ואז נכתוב:

$ git log --follow -n 1 -- utils.py

commit 31f3859a60bf02359d623d8f284da9ee4c12488a

Author: shay <shay@tocode.co.il>

Date: Thu Aug 16 05:25:51 2018 -0700

moved twice to main

אנחנו יודעים שהקובץ כבר לא בפרויקט ולכן הקומיט האחרון שהשפיע עליו היה כנראה זה שבגללו הוא נמחק. ננסה לחזור אחורה בזמן לקומיט אחד לפני קומיט זה:

$ git checkout 31f38^

error: The following untracked working tree files would be overwritten by checkout:

.idea/helloworld.iml

.idea/misc.xml

.idea/modules.xml

.idea/vcs.xml

.idea/workspace.xml

Please move or remove them before you switch branches.

Aborting

הסימן **^** מציין קומיט אחד קודם ואפשר להוסיף אותו אחרי מזהה קומיט בדיוק כמו שראינו עם המילה המיוחדת HEAD.

עכשיו להודעת השגיאה: גיט אומר לנו שכל תיקיית idea שנמצאת אצלנו עכשיו בתיקייה עשויה להידרס על ידי הקבצים שיחזרו מהעבר. זיכרו שבעבר תיקיה זו היתה שמורה במערכת ניהול הגירסאות ולכן חזרה אחורה בזמן תחזיר גם את תיקיית idea מהעבר. גיט לא מוכן לסכן את הקבצים שלנו שאינם שמורים במאגר ולכן מסרב לבצע את המסע בזמן.

בשביל שנמשיך בדוגמא אמחוק את תיקיית idea הנוכחית. במקרים אחרים שאתם נתקלים בהודעה כזו תצטרכו להחליט בכל מקרה מה ההתנהגות הנכונה ביותר עבורכם: לפעמים תרצו להחזיר רק חלק מהקבצים, לפעמים לשים את הקבצים שעשויים להידרס בצד ולפעמים כמו שאנחנו עושים כאן תמחקו אותם ותמשיכו עם המסע בזמן.

משורת הפקודה נכתוב:

$ rm -rf .idea

וכעת ננסה שוב את ה checkout:

$ git checkout 31f38^

Note: checking out '31f38^'.

You are in 'detached HEAD' state. You can look around, make experimental

changes and commit them, and you can discard any commits you make in this

state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may

do so (now or later) by using -b with the checkout command again. Example:

git checkout -b <new-branch-name>

HEAD is now at cc62d61 ADD utils.py and using it from main.py

מוזמנים להסתכל מסביב בתיקיה ובלוג כדי להיזכר איך נראו החיים כשעוד היה לנו את הקובץ utils.py. כשתסיימו נעבור לדבר על האזהרה של גיט.

2. מהו מצב Detached Head

בזמן שאתם במסע בזמן הדברים לא עובדים בדיוק כמו שהם עובדים בהווה. כמובן שאפשר להסתכל בקבצים, להעתיק אותם החוצה מתיקיית הפרויקט ואפילו לשנות אותם. אבל מה יקרה אם ננסה ליצור עכשיו קומיט חדש? איפה קומיט זה יישמר? ומה יקרה איתו כשנחזור להווה?

ננסה ונראה.

עדכנו את הקובץ main.py על ידי שינוי המספר בשורה האחרונה מ-5 ל-6. אחרי השינוי הקובץ נראה כך:

import utils

def hello\_text():

return "Hello World! -- v2"

print(hello\_text())

print(utils.twice(6))

עכשיו נוסיף את הקובץ ל Staging Area וננסה לבצע קומיט:

$ git commit -a -m 'committing in detached head state'

[detached HEAD f5a353e] committing in detached head state

1 file changed, 1 insertion(+), 1 deletion(-)

אגב המתג **-a** שהעברתי לקומיט מבצע add באופן אוטומטי לכל הקבצים ששונו מאז הקומיט האחרון ובכך חוסך לנו לכתוב את ה add.

אז גיט איפשר לנו לעדכן קובץ וגם לשמור אותו מתוך מצב החזרה בזמן. צפיה בלוג תראה לנו שהקומיט גם נשמר:

$ git log -n 2

commit f5a353eb2fdfba4b4f03cb242a2d21e99b425f6b (HEAD)

Author: shay <shay@tocode.co.il>

Date: Thu Aug 16 07:15:54 2018 -0700

committing in detached head state

commit cc62d6140975e1ab806c3597b47e410b2daed37e

Author: shay <shay@tocode.co.il>

Date: Thu Aug 16 03:36:31 2018 -0700

ADD utils.py and using it from main.py

3. חזרה לגירסא העדכנית של המערכת

אבל מה יקרה כשננסה לחזור לגירסא העדכנית ביותר של המערכת? הפקודה:

$ git checkout master

חוזרת לגירסא העדכנית ביותר של הבראנצ' master. אני יודע שעוד לא דיברנו על בראנצ'ים אבל בגדול נגיד שבראנצ' מייצג רצף של קומיטים והבראנצ' master הוא ענף ברירת המחדל לכל פרויקט חדש. החזרה ל master היא בעצם חזרה למצב העדכני ביותר של הפרויקט. כשנדבר על בראנצ'ים נלמד איך לפתוח מספר רצפים מקבילים של קומיטים - כלומר מספר בראנצ'ים - ואז הכל הולך להסתבך מחדש.

בינתיים בואו נראה מה קרה. הפקודה הקודמת גרמה להדפסת הפלט הבא:

$ git checkout master

Warning: you are leaving 1 commit behind, not connected to

any of your branches:

f5a353e committing in detached head state

If you want to keep it by creating a new branch, this may be a good time

to do so with:

git branch <new-branch-name> f5a353e

Switched to branch 'master'

גיט אומר לנו שהשארנו קומיט אחד מאחור. הקומיט f5a353e לא שייך לרצף הזמן (או הקומיטים) בו אני נמצא. אחרי החזרה להווה אם תכתבו **git log** לא תראו אותו ברשימה. בואו נראה מה הכוונה - נצפה בקומיט האחרון שלנו:

$ git cat-file commit HEAD

tree 4847242776fe4bd66d6a692d99ce83bf4dd1dc8e

parent be251ba38a07891a77c06ad771a5a9a5a47be1d7

author shay <shay@tocode.co.il> 1534423426 -0700

committer shay <shay@tocode.co.il> 1534423426 -0700

removed .idea folder from the repository

ואפשר לראות שמתחת ל tree יש שדה נוסף שנקרא parent. שדה זה מייצג את הקומיט הקודם. אפשר להמשיך בשרשרת ולראות את אותו קומיט קודם ונגלה שגם לו יש parent:

$ git cat-file commit be251b

tree d056f66ea147c4da0f11d56c5e8cd1649b191614

parent 31f3859a60bf02359d623d8f284da9ee4c12488a

author shay <shay@tocode.co.il> 1534423052 -0700

committer shay <shay@tocode.co.il> 1534423052 -0700

renamed file main

ואפשר להמשיך בשרשרת הזו עד לקומיט הישן ביותר במערכת. ואם נעשה את זה בשום שלב בדרך לא נגיע לקומיט f5a353e, הקומיט שבוצע כשהיינו בעבר. קומיט זה לא נמצא על המסלול בין תחילת העבודה לבין הקומיט העדכני ביותר. זה קומיט שנוצר מהסתעפות שנראית בערך כך:

A -> B -> C -> D

|

V

E

שימו לב למסלול מ A ל D: הקומיט E פשוט לא נמצא עליו ולכן לכאורה אבד.

כשנלמד לעבוד עם בראנצ'ים נגלה שאפשר לשמור את קומיט E בשם ייחודי שלו וגם לחזור אליו להמשך פיתוח. בינתיים נזכור שהוא קיים אבל לא הכי קל להגיע אליו דרך הלוג הרגיל. פקודת גיט נוספת תציג לנו את כל הקומיטים המוכרים במערכת ותכלול גם קומיט זה:

$ git reflog

1272f32 (HEAD -> master) HEAD@{0}: checkout: moving from f5a353eb2fdfba4b4f03cb242a2d21e99b425f6b to master

f5a353e HEAD@{1}: commit: committing in detached head state

cc62d61 HEAD@{2}: checkout: moving from master to 31f38^

1272f32 (HEAD -> master) HEAD@{3}: commit: removed .idea folder from the repository

be251ba HEAD@{4}: commit: renamed file main

31f3859 HEAD@{5}: commit: moved twice to main

cc62d61 HEAD@{6}: commit: ADD utils.py and using it from main.py

611ef8a HEAD@{7}: commit: Yay I saved the world

b656e9f HEAD@{8}: commit: ADD: move hello text to its own function

10966a6 HEAD@{9}: commit (initial): initial commit

4. משיכת קובץ בודד או מספר קבצים מגירסא אחרת

הרבה פעמים אנחנו לא רוצים ממש לחזור אחורה לגירסא מסוימת אלא מספיק לנו לקחת קובץ אחד או מספר קבצים מגירסא ישנה. פעולה כזו היתה חוסכת לנו את הצורך למחוק את תיקיית **.idea** ומאפשרת לקחת רק את קובץ **utils.py** ששמור בגירסא הקודמת.

אני מזכיר שהפקודה הבאה מציגה את הקומיט האחרון שהשפיע על הקובץ utils.py:

$ git log --follow -n 1 -- utils.py

commit 31f3859a60bf02359d623d8f284da9ee4c12488a

Author: shay <shay@tocode.co.il>

Date: Thu Aug 16 05:25:51 2018 -0700

moved twice to main

והפקודה הבאה מושכת את הקובץ **utils.py** מגירסא אחת לפני 31f385 חזרה לתיקיה הנוכחית בלי לשנות שום דבר אחר בפרויקט ובלי לנסוע חזרה בזמן:

$ git checkout 31f385^ -- utils.py

עכשיו ב git status הקובץ מופיע כקובץ חדש ואפשר להוסיף אותו שוב למערכת, לשנות אותו או לקחת ממנו רק את מה שאתם צריכים ולמחוק חזרה:

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: utils.py

Untracked files:

(use "git add <file>..." to include in what will be committed)

\_\_pycache\_\_/

אנחנו יודעים איך נשמרת היסטוריית הפרויקט בגיט וגם איך לחזור אחורה ולהסתכל על נקודה מסוימת בהיסטוריה. בשיעור זה נלמד מספר שיטות לחפש דברים ספציפיים בהיסטוריית הפרויקט.

לאן נעלם הקובץ שלי

הפקודה **git log** מסוגלת לזהות שינויים שהשפיעו על קבצים מסוימים ולכן נוכל באמצעותה לזהות קבצים שנעלמו ולהבין מה קרה להם.

נראה דוגמא על הפרויקט שלנו. אני מזכיר שבפרויקט יש לנו בינתיים קובץ פייתון בודד בשם demo.py והתוכן שלו:

def twice(x):

return x \* 2

def hello\_text():

return "Hello World! -- v2"

print(hello\_text())

print(twice(5))

בנוסף יש לנו תיקיה בשם venv שם פייתון שומר את סביבת העבודה שלנו והמודולים בהם השתמשנו. תיקיה זו מאוד מגדילה לנו את הריפוזיטורי ולכן בפייתון לא נהוג לשמור אותה במלואה יחד עם היסטוריית הפרויקט. במקום זה מתכנתי פייתון שומרים קובץ יחיד שנקרא **requirements.txt** בו הם כותבים את שמות כל המודולים שמשתמשים בהם בפרויקט ואז כשמתקינים את המערכת על מחשב חדש מפעילים את הפקודה **pip install -r requirements.txt** כדי להתקין את כל המודולים האלה מהרשת. מנגנון זה אגב לא ייחודי לפייתון.

בואו נוציא את התיקיה **venv** ממאגר ההיסטוריה שלנו ובמקביל נוסיף קובץ readme.txt שמסביר שבשביל להתחיל לעבוד על מכונה חדשה צריך תחילה להתקין את המודולים.

צרו קובץ חדש בשם readme.txt עם התוכן:

Hello World

-----------

To start working on the project you'll first need to install the mdoules it uses by runnign:

pip install -r requirements.txt

באותה הזדמנות כדאי ליצור גם את הקובץ requirements.txt כדי שאף אחד לא ייבהל. מאחר ואנחנו עדיין לא משתמשים במודולים חיצוניים בפרויקט נשאיר אותו ריק - לכן צרו קובץ ריק בשם requirements.txt בתיקיית הפרויקט.

בחזרה לשורת הפקודה ונבצע את השינוי:

$ git rm -r venv

$ git add requirements.txt readme.txt

$ git commit -m 'replaced virtual env with requirements.txt file in source control'

אם הכל הלך חלק תקבלו את הסטטוס הבא:

$ git status

On branch master

Untracked files:

(use "git add <file>..." to include in what will be committed)

.idea/

\_\_pycache\_\_/

השבועות עוברים ושכחתם מכל הסיפור הזה של venv ו requirements.txt. כנראה שהמשכתם לעבוד והיו עוד הרבה קומיטים מאז אותה מחיקה. פתאום מופיעה איזה הודעת שגיאה מוזרה או שהתקנתם את הפרויקט במקום אחר ואתם רוצים לדעת לאן נעלמה ספריית venv.

באמצעות git log נוכל לזהות את כל הקומיטים הקשורים לקובץ או תיקיה מסוימים:

$ git log -- venv

commit 932937d16179e8a96ff54d9e59a4ebdb108d5aa6 (HEAD -> master)

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 00:12:35 2022 -0700

replaced virtual env with requirements.txt file in source control

commit 10966a66b9a80f9fc3b55fca4bcfe172b40bd7bd

Author: shay <shay@tocode.co.il>

Date: Thu Aug 16 02:43:28 2022 -0700

initial commit

שימו לב לשני המינוסים בין הפקודה לקבצים בהם אנחנו מתעניינים. הם הכרחיים ונמצאים בעוד הרבה פקודות גיט באותה משמעות.

קל לראות שהתיקיה venv הושפעה בסך הכל משני קומיטים: הקומיט הראשון שכנראה הוסיף אותה לפרויקט והקומיט האחרון בו היא נמחקה. אנחנו גם יכולים לראות את ההסבר בתיאור הקומיט וכך להיזכר למה הספריה יצאה ומה עושים במקומה.

שתי יכולות נוספות שכדאי להכיר של log הן הצגת תקציר השינויים באמצעות:

$ git log --shortstat -- venv

commit 932937d16179e8a96ff54d9e59a4ebdb108d5aa6 (HEAD -> master)

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 00:12:35 2022 -0700

replaced virtual env with requirements.txt file in source control

356 files changed, 102130 deletions(-)

commit 10966a66b9a80f9fc3b55fca4bcfe172b40bd7bd

Author: shay <shay@tocode.co.il>

Date: Thu Aug 16 02:43:28 2022 -0700

initial commit

356 files changed, 102130 insertions(+)

כך אפשר לראות כמה קבצים השתנו, נמחקו או נוספו בכל קומיט.

ואפשרות אחרונה היא הצגת השינויים עצמם. לא אראה את זה על venv כי התוצאה ארוכה מדי אז בואו נלך שינוי אחד אחורה. הפקודה:

$ git log -n 1

מציגה את השינוי האחרון שבוצע. תוסיפו מזהה קומיט ותוכלו ללכת אחורה למשל הפקודה:

$ git log -n 1 HEAD~1

מציגה את הקומיט לפני האחרון, והפקודה:

$ git log -n 1 HEAD~2

את זה שלפניו. אני אלך עוד אחד אחורה לקומיט:

$ git log -n 1 HEAD~3

commit 31f3859a60bf02359d623d8f284da9ee4c12488a

Author: shay <shay@tocode.co.il>

Date: Thu Aug 16 05:25:51 2022 -0700

moved twice to main

ואפעיל את הפקודה:

$ git log -n 1 -p HEAD~3

$ git log -p -n 1 HEAD~3

commit 31f3859a60bf02359d623d8f284da9ee4c12488a

Author: shay <shay@tocode.co.il>

Date: Thu Aug 16 05:25:51 2022 -0700

moved twice to main

diff --git a/main.py b/main.py

index 3a94283..2035504 100644

--- a/main.py

+++ b/main.py

@@ -1,7 +1,8 @@

-import utils

+def twice(x):

+ return x \* 2

def hello\_text():

return "Hello World! -- v2"

print(hello\_text())

-print(utils.twice(5))

\ No newline at end of file

+print(twice(5))

\ No newline at end of file

diff --git a/utils.py b/utils.py

התוצאה הקצת קשה לקריאה היא הצגת השינויים שקומיט זה הכניס למערכת. אפשר להתאמץ ולראות בקובץ main.py את המינוס מימין לשורה **import utils** שמסמן ששורה זו נמחקה ומתחתיה את הפלוס ליד שתי השורות הבאות לסמן שהפונקציה **twice** נוספה לקובץ. בשיעורים הבאים נדבר יותר לעומק על איך הבדלים אלה מוצגים וגם נלמד להשתמש בכלי גרפי להציג אותם.

2. מאיפה הגיעה השורה הזו?

אחד הכלים הכי חשובים בהסתכלות על קוד הוא לזהות את ההקשר (context) בו נכתבה שורה מסוימת או קטע קוד מסוים. הפקודה עם השם המוזר git blame נועדה בדיוק לזה. פקודה זו תציג לנו מי הכניס למערכת כל שורת קוד ובאיזה קומיט הם עשו את זה. אחרי זה נוכל להסתכל בפרטי הקומיט כדי להבין מה בדיוק קרה שם ואיזה עוד קבצים ושורות קשורים לשינוי זה.

בואו נישאר עם הקובץ main.py והשינוי בו העברנו את הפונקציה twice מהקובץ utils.py אליו. הפקודה הבאה מציגה את תוכן main.py המלא נכון לאותה נקודה ועבור כל שורה מראה לנו מי הכניס את השורה ובאיזה קומיט (בהרצה אצלכם תצטרכו לשנות את המזהה 31f385 למזהה הקומיט המתאים שראיתם בלוג):

$ git blame 31f385 main.py

31f3859a (shay 2022-08-16 05:25:51 -0700 1) def twice(x):

31f3859a (shay 2022-08-16 05:25:51 -0700 2) return x \* 2

cc62d614 (shay 2022-08-16 03:36:31 -0700 3)

b656e9ff (shay 2022-08-16 02:48:06 -0700 4) def hello\_text():

cc62d614 (shay 2022-08-16 03:36:31 -0700 5) return "Hello World! -- v2"

b656e9ff (shay 2022-08-16 02:48:06 -0700 6)

cc62d614 (shay 2022-08-16 03:36:31 -0700 7) print(hello\_text())

31f3859a (shay 2022-08-16 05:25:51 -0700 8) print(twice(5))

אם לא תציינו מזהה קומיט תקבלו תיעוד מלא של השורות בקובץ מהגירסא הנוכחית למשל:

$ git blame demo.py

31f3859a main.py (shay 2022-08-16 05:25:51 -0700 1) def twice(x):

31f3859a main.py (shay 2022-08-16 05:25:51 -0700 2) return x \* 2

cc62d614 main.py (shay 2022-08-16 03:36:31 -0700 3)

b656e9ff main.py (shay 2022-08-16 02:48:06 -0700 4) def hello\_text():

cc62d614 main.py (shay 2022-08-16 03:36:31 -0700 5) return "Hello World! -- v2"

b656e9ff main.py (shay 2022-08-16 02:48:06 -0700 6)

cc62d614 main.py (shay 2022-08-16 03:36:31 -0700 7) print(hello\_text())

31f3859a main.py (shay 2022-08-16 05:25:51 -0700 8) print(twice(5))

באמצעות לוג כפי שכבר ראינו אפשר לקחת כל קומיט כזה ולראות איזה עוד קבצים הוא שינה ומה בדיוק השינויים שהוכנסו בו וגם על מה חשבתם כשיצרתם את הקומיט, כלומר את הודעת התיעוד שכתבתם.

3. נדמה לי שהיתה שם שורת קוד

ויש גם מצבים הפוכים: אתה זוכר שהיתה איזה שורת קוד חשובה בקובץ אבל כבר לא מצליח למצוא אותה. בואו ניצור אחד כזה. כנסו לקובץ demo.py והחליפו את התוכן שלו בטקסט הבא:

def twice(x):

return x \* 2

print(hello\_text())

print(twice(5))

מחקנו את הגדרת הפוקנציה **hello\_text** אבל שכחנו למחוק את הקריאה לפונקציה זו כך שעכשיו הקוד לא יעבוד. לא נורא זה בכוונה. נכניס את השינוי לריפוזיטורי:

$ git add demo.py

$ git commit -m 'ADD demo.py better hello text'

שבועות עוברים ופתאום מישהו מנסה להריץ את הסקריפט ורואה שנכשל. מבט על הקוד מראה שאין הגדרה לפונקציה **hello\_text**. לאן היא נעלמה אתם שואלים? גיט יודע לענות. נשתמש באותו git blame שראינו אבל הפעם נרצה לדעת מי מחק שורה מסוימת. בשביל זה צריך לזהות קודם כל איזשהו קומיט בו השורה הזו היתה בקוד. אני זוכר שלפני שבועיים היא היתה ולכן אוכל לכתוב:

$ git log -n 1 --before 14.days.ago

כדי לקבל קומיט מלפני שבועיים או יותר. בואו נניח שמספר הקומיט שמצאתם הוא 31f385. עכשיו נכתוב:

$ git blame --reverse 31f385..HEAD -- main.py

4ff8942f demo.py (shay 2022-08-18 00:43:36 -0700 1) def twice(x):

4ff8942f demo.py (shay 2022-08-18 00:43:36 -0700 2) return x \* 2

4ff8942f demo.py (shay 2022-08-18 00:43:36 -0700 3)

932937d1 demo.py (shay 2022-08-18 00:12:35 -0700 4) def hello\_text():

932937d1 demo.py (shay 2022-08-18 00:12:35 -0700 5) return "Hello World! -- v2"

932937d1 demo.py (shay 2022-08-18 00:12:35 -0700 6)

4ff8942f demo.py (shay 2022-08-18 00:43:36 -0700 7) print(hello\_text())

4ff8942f demo.py (shay 2022-08-18 00:43:36 -0700 8) print(twice(5))

אנחנו רואים את הקובץ כפי שהופיע בגירסא 31f385 כאשר המספר הראשון בכל שורה הוא מזהה הקומיט האחרון בו שורה זו הופיעה. כך אפשר לראות שהגדרת הפונקציה **hello\_text** נראתה לאחרונה בגירסא 932937d1. רוצים לדעת מה קרה אחרי? נשתמש בגיט לוג:

$ git log 932937d1..HEAD

commit 4ff8942fea91b7bcfa2ad885dbb4fa9d8255ee55 (HEAD -> master)

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 00:43:36 2022 -0700

ADD demo.py better hello text

התקנת כלי גרפי להצגת השינויים

שורת הפקודה היא לא המקום הכי מוצלח להציג שינויים בין מספר רב של קבצים והרבה פעמים לקרוא פלט של diff בגיט עשוי לקחת לכם יותר זמן ממה שהתכוונתם לבזבז על החיפוש. החדשות הטובות הן שאתם ממש לא חייבים לאמץ את העיניים ויכולים לעבוד עם כל כלי להצגת שינויים שתרצו.

אם אין לכם פייבוריט אני ממליץ על diffmerge שהוא חינמי ונותן תצוגה די נוחה. אם אתם כבר מכירים ומעדיפים כלי אחר (למשל Araxis או Beyond Compare) אין בעיה להגדיר אותם באותו אופן שאני אציג כאן עבור diffmerge.

תחילה נוריד את הכלי בחינם מהאתר שלהם:

<https://sourcegear.com/diffmerge/downloads.php>

לחיצה כפולה ועוד מספר לחיצות על Next ואתם מסודרים. הפעילו מתפריט "התחל" כדי לראות שהכל הותקן כמו שצריך ובדקו באיזה תיקיה הוא הותקן (בדרך כלל זה Program Files). לאחר מכן נמשיך בהגדרות.

חיזרו לשורת הפקודה וכתבו שם:

$ git config --global --edit

הפקודה תפתח לכם בעורך טקסט את קובץ ההגדרות הגלובלי של גיט. עוד נדבר עליו יותר לעומק באחד השיעורים הבאים. בינתיים אל תשנו את הבלוק user שכבר נמצא שם עם הגדרות המשתמש שלכם והוסיפו אחריו את התוכן הבא:

[merge]

tool = diffmerge

[diff]

tool = diffmerge

[mergetool]

prompt = false

[difftool]

prompt = false

[difftool "diffmerge"]

cmd = C:/Program\\ Files/SourceGear/Common/DiffMerge/sgdm.exe "$LOCAL" "$REMOTE"

[mergetool "diffmerge"]

cmd = C:/Program\\ Files/SourceGear/Common/DiffMerge/sgdm.exe -merge -result="$MERGED" "$LOCAL" "$BASE" "$REMOTE"

אחרי השינוי הפקודה git diff תציג לכם את השינויים בשורת הפקודה ואילו הפקודה git difftool תציג את השינויים בכלי הגרפי difftool.

נראה מספר דוגמאות על הפרויקט שלנו.

2. הצגת שינויים בקובץ בין שתי גירסאות

הפקודה **git diff** מקבלת מספר גירסא וקובץ ומציגה את השינויים בקובץ בין הגירסא שאצלכם לגירסא שביקשתם. שני הקומיטים האחרונים שלי בפרויקט היו:

$ git log -n 2

commit 4ff8942fea91b7bcfa2ad885dbb4fa9d8255ee55 (HEAD -> master)

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 00:43:36 2022 -0700

ADD demo.py better hello text

commit 932937d16179e8a96ff54d9e59a4ebdb108d5aa6

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 00:12:35 2022 -0700

replaced virtual env with requirements.txt file in source control

ולכן אפשר לבדוק מה השתנה בקובץ demo.py בצורה הבאה:

$ git diff 93293 -- demo.py

diff --git a/demo.py b/demo.py

index 2035504..443818e 100644

--- a/demo.py

+++ b/demo.py

@@ -1,8 +1,5 @@

def twice(x):

return x \* 2

-def hello\_text():

- return "Hello World! -- v2"

-

print(hello\_text())

print(twice(5))

\ No newline at end of file

או בצורה גרפית עם:

$ git difftool 93293 -- demo.py

הרבה פעמים לפני שנתחיל לגשת ולהוסיף קבצים לקומיט הבא נרצה להריץ את הפקודה:

$ git diff

שתדפיס לנו איזה שורות בדיוק השתנו בעץ הפרויקט שלנו שעדיין לא יצרנו מהם קומיט.

נחזור לשנות את demo.py ונכתוב בו את התוכן הבא:

def twice(x):

return x \* 2

print(twice(5))

עכשיו נריץ ונקבל:

$ git diff

diff --git a/demo.py b/demo.py

index 443818e..4077b79 100644

--- a/demo.py

+++ b/demo.py

@@ -1,5 +1,4 @@

def twice(x):

return x \* 2

-print(hello\_text())

print(twice(5))

\ No newline at end of file

כמובן שאפשר להחליף את diff ב difftool ואז נראה את השינויים בצורה גרפית. נשנה גם את הקובץ readme.txt ונוסיף לו שורה כך שתוכן הקובץ הוא:

Hello World

-----------

Author: shay dgani

To start working on the project you'll first need to install the mdoules it uses by runnign:

pip install -r requirements.txt

עכשיו תוכלו להפעיל:

$ git difftool

כדי לראות את השינויים בשני הקבצים. נשים לב שגיט פותח קודם את השינויים בקובץ הראשון ורק אחר כך את השינויים בקובץ השני. במצב שיש לנו שינויים במספר קבצים נוח להוסיף את הפרמטר **--dir-diff** וכך לקבל חלון אחד שמראה את כל הקבצים שהשתנו ובו אפשר לבחור לאיזה קובץ להיכנס:

$ git difftool --dir-diff

אחרי שהשתכנענו איזה קבצים נרצה להוסיף לקומיט הבא נוכל להוסיף אותם עם:

$ git add readme.txt demo.py

עכשיו הרצה של diff תציג פלט ריק מאחר וכל השינויים מתוכננים להיכנס בקומיט הבא. הפקודה הבאה מראה לנו איזה שינויים אלה כלומר את ההבדל בין אזור ה Staging לקומיט האחרון:

$ git diff --cached

diff --git a/demo.py b/demo.py

index 443818e..4077b79 100644

--- a/demo.py

+++ b/demo.py

@@ -1,5 +1,4 @@

def twice(x):

return x \* 2

-print(hello\_text())

print(twice(5))

\ No newline at end of file

diff --git a/readme.txt b/readme.txt

index 8f2b8ca..4217f68 100644

--- a/readme.txt

+++ b/readme.txt

@@ -1,6 +1,8 @@

Hello World

-----------

+Author: shay dgani

To start working on the project you'll first need to install the mdoules it uses by runnign:

3. הצגת שינויים במספר קבצים בין שתי גירסאות

באמצעות git diff קל לראות שינויים בכל הפרויקט או בתת תיקיה מסוימת שלו בין שתי גירסאות. הפקודה:

$ git difftool --dir-diff <commit**1**> <commit**2**>

מציגה את כל השינויים בין שתי תיקיות הפרויקט בשתי הגירסאות.

אחת הפקודות החשובות בעבודה עם גיט היא git reset. בדומה ל checkout גם reset מאפשרת לטייל בהיסטוריה אבל reset מסוגלת גם לשנות את ההיסטוריה. בשיעור זה נלמד איך.

על HEAD ועץ ההיסטוריה

באופן רגיל כל פעם שאנחנו יוצרים קומיט אנחנו מוסיפים עוד גירסא של הפרויקט למאגר ההיסטוריה. כל גירסא כזו יודעת מי הגירסא שהגיעה לפניה ורצף הגירסאות בפרויקט נקרא Branch. המילה המיוחדת HEAD מתיחסת לקומיט האחרון ב Branch הנוכחי כלומר לרצף הגירסאות שאנחנו עכשיו עובדים עליו.

בכל פעם שאנחנו מפעילים git commit המצביע HEAD עובר להצביע על הקומיט החדש שהוספנו. מאחר וכל קומיט "יודע" מי הקומיט שבא לפניו אפשר דרך HEAD לבנות את כל רצף הגירסאות מהמקום הנוכחי אחורה עד תחילת הפרויקט.

יש מספר דרכים להזיז את HEAD: בפרק הבא נלמד על בניית Branch-ים חדשים ועל הזזת HEAD לבראנצ'ים אלה. כרגע מה שמעניין אותנו הוא היכולת להזיז את HEAD בתוך ה Branch הנוכחי ובמקביל להשפיע על הקבצים בתיקיית העבודה או ב Staging Area. נראה את זה רגע עם git log:

$ git log -n 3

commit 4ff8942fea91b7bcfa2ad885dbb4fa9d8255ee55 (HEAD -> master)

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 00:43:36 2022 -0700

ADD demo.py better hello text

commit 932937d16179e8a96ff54d9e59a4ebdb108d5aa6

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 00:12:35 2022 -0700

replaced virtual env with requirements.txt file in source control

commit 1272f3215b482d5df5005a39c37f9646b1a48d8a

Author: shay <shay@tocode.co.il>

Date: Thu Aug 16 05:43:46 2022 -0700

removed .idea folder from the repository

שימו לב לקומיט הראשון: בסוגריים אחרי מזהה הקומיט אנחנו רואים את המילה HEAD שמצביעה למילה master. זה אומר שהרצף הנוכחי שלנו נקרא master והקומיט האחרון ברצף (העדכני ביותר) הוא קומיט מספר 4ff89.

צפיה בסטטוס מלמדת אותנו:

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: demo.py

modified: readme.txt

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: readme.txt

Untracked files:

(use "git add <file>..." to include in what will be committed)

.idea/

\_\_pycache\_\_/

שיש לנו שני קבצים שעודכנו ונשמרו ב Staging Area ומחכים לקומיט - אלה הקבצים demo.py ו readme.txt.

2. מחיקת קבצים מ Staging

ההתנהגות הראשונה של reset היא להחזיר את ה Staging Area או חלק מהקבצים בו למצב שהם היו בקומיט מסוים. בדרך כלל נוותר על מזהה הקומיט ונשתמש ב HEAD כדי להוציא קובץ מסוים מה Staging Area.

במקרה שלנו נכתוב:

$ git reset readme.txt

Unstaged changes after reset:

M readme.txt

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: demo.py

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: readme.txt

Untracked files:

(use "git add <file>..." to include in what will be committed)

.idea/

\_\_pycache\_\_/

ותראו מה קרה: הקובץ readme.txt שהיה ב Staging Area חזר לגירסא שהיתה לו ב HEAD. מאחר והקובץ בתיקיית העבודה שלנו שונה מזה שהיה ב HEAD אנחנו רואים את הקובץ מחוץ ל Staging Area.

דרך אחרת לראות את ההתנהגות היא עם **ls-files**. נוסיף שוב את readme.txt ונריץ **ls-files**:

$ git ls-files --stage

100644 4077b792ab3a1b3e3b1f6eb3b8e66126858cc22f 0 demo.py

100644 8f2b8caed52e95a115118b3a111a6255b43e1e6e 0 readme.txt

100644 e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 0 requirements.txt

$ git add readme.txt

$ git ls-files --stage

100644 4077b792ab3a1b3e3b1f6eb3b8e66126858cc22f 0 demo.py

100644 2bf144b495cca7bcb7c33ecd4bf018532e1fcce4 0 readme.txt

100644 e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 0 requirements.txt

הגירסא של readme.txt ב Staging Area לפני ההוספה היתה 8f2b8, ואילו אחרי ההוספה הגירסא הפכה ל 2bf144 שהיא גירסת הקובץ בתיקיית העבודה שלנו.

נפעיל reset ונראה שהקובץ ב Staging Area חוזר לגירסא שהיתה לו ב HEAD:

$ git reset readme.txt

$ git ls-files --stage

100644 4077b792ab3a1b3e3b1f6eb3b8e66126858cc22f 0 demo.py

100644 8f2b8caed52e95a115118b3a111a6255b43e1e6e 0 readme.txt

100644 e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 0 requirements.txt

כלומר הפעלת git reset עם שם קובץ מאפסת את גירסת הקובץ ב Staging Area כך שתתאים למה שנמצא ב HEAD (שזה בעצם ההפך של git add). בלי להעביר קובץ הפקודה תאפס את ה Staging Area ותוציא משם את כל הקבצים.

3. ביטול שינויים בקבצים

מטרה אחרת של reset היא איפוס תיקיית העבודה שלכם. זה מה שמראה git status בפרויקט שלי כרגע:

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: demo.py

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: readme.txt

Untracked files:

(use "git add <file>..." to include in what will be committed)

.idea/

\_\_pycache\_\_/

הקובץ demo.py עודכן ומצפה להיכנס לקומיט הבא והקובץ readme.txt עודכן אבל לא מתוכנן להיכנס לקומיט הבא. עכשיו נניח שהתחרטנו ואנחנו רוצים לאפס את כל השינויים באחד או יותר מהקבצים. הפקודה reset יכולה לעזור לנו בזה. נכתוב:

$ git reset --hard

HEAD is now at 4ff8942 ADD demo.py better hello text

IEUser@MSEDGEWIN10 MINGW64 ~/Documents/helloworld (master)

$ git status

On branch master

Untracked files:

(use "git add <file>..." to include in what will be committed)

.idea/

\_\_pycache\_\_/

וכל השינויים המקומיים נמחקו והפרויקט חזר למצב בו היה ב HEAD.

שווה לציין ש **git reset --hard** לא ימחק קבצים שנמצאים בתיקיית העבודה אבל לא ב HEAD ובגלל זה התיקיות **.idea** ו **\_\_pycache\_\_** עדיין נשארו.

כדי למחוק מעץ הפרויקט קבצים ותיקיות ש git לא מכיר תוכלו להשתמש בפקודה clean. תחילה הפקודה:

$ git clean -d -n

Would remove .idea/

Would remove \_\_pycache\_\_/

תספר לכם איזה תיקיות וקבצים היא מתכננת למחוק אם תופעל. במידה ששוכנעתם שבאמת אתם לא צריכים את התיקיות והקבצים האלה תוכלו להשתמש בפקודה:

$ git clean -d -f

כדי באמת למחוק אותם.

4. שינוי עץ ההיסטוריה

התנהגות שלישית של reset שכדאי להכיר היא היכולת להעביר את HEAD לקומיט מסוים בלי לשנות את הקבצים בתיקיית העבודה. ראינו משהו דומה עם checkout: כשחזרנו עם checkout לגירסא ישנה בזמן הפקודה לקחה את כל הקבצים מהמאגר והעתיקה אותם לתיקית העבודה שלנו וגם שינתה את HEAD כך שיצביע על אותה גירסא ישנה. אבל אני מזכיר ששם התזוזה היתה זמנית. בסיום העבודה HEAD חזר לחזית הפרויקט וכל השינויים שעשינו כשהיינו באותו מצב "מנותק" נמחקו.

בעזרת reset נוכל לחזור אחורה בזמן, לשנות משהו בהיסטוריה ולהמשיך משם. ברוב המקרים נרצה לחזור רק קומיט אחד אחורה בשביל לתקן הודעה או להוסיף קבצים לקומיט האחרון. בואו נראה את זה על הפרויקט שלנו.

תחילה ניכנס לקובץ demo.py ונשנה את תוכנו לקוד הבא:

def fib(n):

x, y = 1, 1

for i in range(n):

x, y = y, x + y

return x

print(fib(5))

ניצור קומיט עם השינוי ונכניס אותו למאגר:

$ git add demo.py

$ git commit -m 'CHANGED demo.py to print fibonacci number'

[master ef29b25] CHANGED demo.py to print fibonacci number

1 file changed, 6 insertions(+), 4 deletions(-)

לא עוברות עשר דקות לפני שאתם מתחילים לקבל רגליים קרות: ככה לקלקל את demo.py? ומה אם מישהו היה צריך אותו? לא עדיף לשמור את התוכן הישן ולשים את הקוד החדש שלנו בקובץ אחר? ואולי ההודעה לא היתה טובה?

מזל שיש לנו את git reset.

הפעילו:

$ git reset --soft HEAD^

הפקודה לא שינתה את הקבצים בתיקיית העבודה, אבל הזיזה את HEAD קומיט אחד אחורה. נפעיל:

$ git log -n 3

commit 4ff8942fea91b7bcfa2ad885dbb4fa9d8255ee55 (HEAD -> master)

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 00:43:36 2022 -0700

ADD demo.py better hello text

commit 932937d16179e8a96ff54d9e59a4ebdb108d5aa6

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 00:12:35 2022 -0700

replaced virtual env with requirements.txt file in source control

commit 1272f3215b482d5df5005a39c37f9646b1a48d8a

Author: shay <shay@tocode.co.il>

Date: Thu Aug 16 05:43:46 2022 -0700

removed .idea folder from the repository

והקומיט האחרון שלנו פשוט לא מופיע כאן!

(אגב אל דאגה אתם תמיד יכולים לראות אותו עם **git reflog**).

מבחינת הסטטוס הקובץ demo.py החדש שלנו שונה מזה ששמור במאגר ולכן נקבל:

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: demo.py

Untracked files:

(use "git add <file>..." to include in what will be committed)

.idea/

\_\_pycache\_\_/

אפשר לראות diff בין demo.py הנוכחי לקובץ הקודם עם:

$ git diff HEAD demo.py

diff --git a/demo.py b/demo.py

index 443818e..9c0a66c 100644

--- a/demo.py

+++ b/demo.py

@@ -1,5 +1,7 @@

-def twice(x):

- return x \* 2

+def fib(n):

+ x, y = 1, 1

+ for i in range(n):

+ x, y = y, x + y

+ return x

-print(hello\_text())

-print(twice(5))

\ No newline at end of file

+print(fib(5))

וקצת יותר מסובך אבל עדיין אפשרי לראות את הקובץ הקודם במלואו. הפקודה הבאה מדפיסה את מזהי כל הקבצים ששמורים במאגר בגירסא האחרונה:

$ git ls-tree HEAD^{tree}

100644 blob 443818e46de4dbbe5bd559274876a65528fbba7d demo.py

100644 blob 8f2b8caed52e95a115118b3a111a6255b43e1e6e readme.txt

100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391 requirements.txt

המזהה אצלי של demo.py הוא 443818e ולכן אפשר להפעיל את הפקודה הבאה כדי לצפות בו:

$ git cat-file blob 443818

def twice(x):

return x \* 2

print(hello\_text())

print(twice(5))

אני רוצה לקחת את התוכן הנוכחי של הקובץ demo.py שהוא בעצם התוכן החדש שכתבתי ועשיתי לו כבר קומיט ולהעביר אותו לקובץ אחר. צרו קובץ בשם fib.py והעתיקו את כל demo.py לשם.

לאחר מכן אני רוצה לקחת את demo.py האחרון מהמאגר חזרה לתיקיה ובשביל זה אפעיל:

$ git reset demo.py

Unstaged changes after reset:

M demo.py

$ git checkout demo.py

המצב במאגר נראה כך:

$ git status

On branch master

Untracked files:

(use "git add <file>..." to include in what will be committed)

.idea/

\_\_pycache\_\_/

fib.py

נוסיף את fib.py החדש למאגר ונבצע קומיט עם הערה חדשה:

$ git add fib.py

$ git commit -m 'ADD new file that calculates fibonacci numbers'

[master a6fabeb] ADD new file that calculates fibonacci numbers

1 file changed, 7 insertions(+)

create mode 100644 fib.py

ונסתכל בלוג:

$ git log -n 3

commit a6fabeb7560cc07fc5d8dcba9531e5ed51f08ba2 (HEAD -> master)

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 12:06:41 2022 -0700

ADD new file that calculates fibonacci numbers

commit 4ff8942fea91b7bcfa2ad885dbb4fa9d8255ee55

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 00:43:36 2022 -0700

ADD demo.py better hello text

commit 932937d16179e8a96ff54d9e59a4ebdb108d5aa6

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 00:12:35 2022 -0700

replaced virtual env with requirements.txt file in source control

אין זכר להודעת הקומיט הקודמת או לתוכן הקומיט. הצלחנו לשנות את ההיסטוריה! אגב למקרה שדאגתם תמיד אפשר להשתמש ב reflog כדי לראות את ההתפצלות והתיקון:

$ git reflog

a6fabeb (HEAD -> master) HEAD@{0}: commit: ADD new file that calculates fibonacci numbers

4ff8942 HEAD@{1}: reset: moving to HEAD^

ef29b25 HEAD@{2}: commit: CHANGED demo.py to print fibonacci number

4ff8942 HEAD@{3}: reset: moving to HEAD

4ff8942 HEAD@{4}: reset: moving to HEAD

4ff8942 HEAD@{5}: commit: ADD demo.py better hello text

932937d HEAD@{6}: commit: replaced virtual env with requirements.txt file in source control

1272f32 HEAD@{7}: reset: moving to HEAD

הקומיט הנעלם הוא מספר ef29b25 ונכון שהוא לא מופיע בלוג שלנו אבל הוא עדיין שמור במאגר ואתם יכולים לצפות בו ובקבצים שלו כדי להיזכר באיך שהעולם יכל היה להיות באמצעות:

$ git show ef29b25

commit ef29b258e2929eadb2b62f38e244e4af16998a75

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 11:49:12 2022 -0700

CHANGED demo.py to print fibonacci number

diff --git a/demo.py b/demo.py

index 443818e..9c0a66c 100644

--- a/demo.py

+++ b/demo.py

@@ -1,5 +1,7 @@

-def twice(x):

- return x \* 2

+def fib(n):

+ x, y = 1, 1

+ for i in range(n):

+ x, y = y, x + y

+ return x

-print(hello\_text())

-print(twice(5))

\ No newline at end of file

+print(fib(5))

איתור מידע במאגר גיט

בקובץ התרגול המצורף תמצאו 3 מאגרי גיט בשמות: ex1, ex2 ו ex3. את הקובץ עם המאגרים ניתן להוריד מכאן:

<https://github.com/tocodeil/tocode-git/raw/master/lab-find-data-in-repos.zip>

בכל אחד מהמאגרים מוחבאת סיסמא סודית שהמתכנת ניסה להסתיר - המשימה שלכם היא למצוא אותה.

2. הודעות קומיט טובות יותר ופחות

באתר github.com תמצאו המון פרויקטים ציבוריים שמשתמשים ב git בתור כלי ניהול הגירסאות שלהם. אפשר להתחיל את הסיור מהקישור הזה:

<https://github.com/explore>

בחרו מספר פרויקטים וחפשו את לוג הקומיטים של כל פרויקט. מצאו 4-5 הודעות קומיט שאהבתם במיוחד ושסיפרו לכם משהו מעניין על הקוד. שימו לב גם להודעות קומיט שלא חידשו דבר או שרק בילבלו אתכם.

אנחנו יודעים שגיט שומר גירסאות של הפרויקט שלנו במאגר ההיסטוריה וגם יודעים די טוב לטייל בין הגירסאות. בשיעור זה נתחיל לדבר על הסתעפויות והיסטוריה מקבילית.

שני סיפורי אהבה

הצורך לעבוד על מספר גירסאות במקביל מגיע לרוב מהעולם האמיתי: יש לנו גירסא אחת של הפרויקט שרצה על שרת הייצור ובאותו הזמן אנחנו עובדים על פיצ'רים חדשים במחשב הפיתוח. או שיש לנו מספר גירסאות של הפרויקט למספר לקוחות שונים ולכולם יש איזשהו שלד משותף אבל גם דברים ייחודיים שלהם.

אני רוצה להשאיר בינתיים בצד את השאלה למה פרויקט צריך הסתעפויות כי נדבר עליה יותר לעומק בשיעור הבא שם גם נדבר על שיטת עבודה נכונה עם הסתעפויות. בינתיים בואו נראה פרקטית איך זה עובד.

אנחנו יודעים ש HEAD הוא מצביע מיוחד על הקומיט האחרון, ואנחנו יודעים ש **git log** מציג את כל ההיסטוריה החל מ HEAD ואחורה עד תחילת הפרויקט. ייתכן ואתם גם זוכרים שראינו את השם master מדי פעם בעץ הפרויקט.

השם master, בדיוק כמו HEAD, מודבק לקומיט העדכני ביותר בפרויקט. הוא מייצג סיפור מסוים או רצף קומיטים מסוים. עד עכשיו היה לנו רק רצף קומיטים יחיד ולכן אמרנו שבפרויקט יש ענף יחיד. הפקודה הראשונה שנראה היא **git branch** שיוצרת ענף חדש. כנסו לתיקיית הפרויקט ורשמו:

$ git branch hello-branches

$ git checkout hello-branches

או אם אתם חובבי קיצורים יכולים במקום זה לכתוב את הקיצור:

$ git checkout -b hello-branches

שתי הגירסאות של הפקודה יוצרות ענף חדש. ענף הוא בסך הכל מצביע על קומיט כלומר קובץ טקסט שכתוב בו מזהה של הקומיט העדכני ביותר בענף. אנחנו יכולים לראות את כל הענפים בפרויקט עם הפקודה:

$ git branch

\* hello-branches

master

או פשוט להיכנס לתוך תיקיית **.git** ולהסתכל על התיקיה בה הענפים השונים שמורים:

$ ls .git/refs/heads/

hello-branches master

ומאחר וענפים הם בסך הכל קבצי טקסט שמתיחסים לקומיט נוכל להציג אותם עם:

$ cat .git/refs/heads/\*

a6fabeb7560cc07fc5d8dcba9531e5ed51f08ba2

a6fabeb7560cc07fc5d8dcba9531e5ed51f08ba2

כרגע שני הענפים מצביעים בדיוק על אותו קומיט. נסתכל בלוג לראות איזה קומיט זה ונקבל:

$ git log -n 2 a6fabe

commit a6fabeb7560cc07fc5d8dcba9531e5ed51f08ba2 (HEAD -> hello-branches, master)

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 12:06:41 2022 -0700

ADD new file that calculates fibonacci numbers

commit 4ff8942fea91b7bcfa2ad885dbb4fa9d8255ee55

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 00:43:36 2022 -0700

ADD demo.py better hello text

העניין בענפים זה שעכשיו כשנוסיף קומיט חדש רק אחד הענפים יתקדם אליו.

2. ביצוע קומיט בתוך ענף

אנחנו נמצאים בתוך הענף **hello-branches** (כפי שראינו דרך הרצת **git branch**). נוסיף קובץ בענף זה. צרו קובץ חדש בשם trees.txt וכיתבו בו מספר שמות של עצים. לאחר מכן נחזור לשורת הפקודה ונכתוב:

$ git add trees.txt

$ git commit -m 'adding a list of trees'

[hello-branches e0646f1] adding a list of trees

1 file changed, 3 insertions(+)

create mode 100644 trees.txt

יצרנו קומיט חדש שמספרו e0646f1 בענף hell-branches. נתבונן ברשומות האחרונות בלוג:

$ git log -n 3

commit e0646f17e1f0fe2746564ef83fdf0d2226991f3b (HEAD -> hello-branches)

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 23:22:58 2022 -0700

adding a list of trees

commit a6fabeb7560cc07fc5d8dcba9531e5ed51f08ba2 (master)

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 12:06:41 2022 -0700

ADD new file that calculates fibonacci numbers

commit 4ff8942fea91b7bcfa2ad885dbb4fa9d8255ee55

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 00:43:36 2022 -0700

ADD demo.py better hello text

ואנחנו רואים שכל אחד מהענפים master ו hello-branches מתיחס כעת לקומיט אחר: הענף master מצביע על הקומיט a6fabe והענף hello-branches על הקומיט e0646f1.

במילים אחרות כל ענף הוא מצביע לקומיט. מקומיט אפשר לבנות את כל רצף ההיסטוריה אחורה עד לתחילת הפרויקט ולכן כל ענף מציין נקודה בזמן ממנה אפשר להמשיך ולבצע קומיטים נוספים.

3. מעבר בין ענפים

נחזור לענף הראשון master ונתקדם במקביל גם בו. הפקודה checkout עוברת בין ענפים וכבר ראינו שאפשר להוסיף לה **-b** כדי ליצור ענף חדש. תורידו את ה **-b** ותוכלו לעבור בין ענפים קיימים:

$ git checkout master

Switched to branch 'master'

בואו נוסיף קומיט חדש לענף master. צרו קובץ בשם flowers.txt וכיתבו בו שמות של מספר פרחים שאתם מכירים. לאחר מכן הפעילו:

$ git branch

hello-branches

\* master

כדי לוודא שאתם על master (רואים לפי הכוכבית) ואז:

$ git add flowers.txt

$ git commit -m 'adding flowers'

[master 01d5778] adding flowers

1 file changed, 3 insertions(+)

create mode 100644 flowers.txt

נתבונן בלוג:

$ git log -n 3

commit 01d577819808d2a5b019cd4a648625114e50a7aa (HEAD -> master)

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 23:33:49 2022 -0700

adding flowers

commit a6fabeb7560cc07fc5d8dcba9531e5ed51f08ba2

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 12:06:41 2022 -0700

ADD new file that calculates fibonacci numbers

commit 4ff8942fea91b7bcfa2ad885dbb4fa9d8255ee55

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 00:43:36 2022 -0700

ADD demo.py better hello text

נשים לב שאנחנו לא רואים את hello-branches ברשימה וגם לא את הקומיט החדש שהוספנו לענף ההוא. הסיבה היא פשוט האופן בו git log מסתכל על ההיסטוריה: מתחילים מהקומיט העדכני ביותר ב HEAD ומדפיסים אותו, לאחר מכן מסתכלים מי ה parent שלו ואז מדפיסים אותו וכך הלאה. לכל קומיט יש את הקומיט הקודם לו. אני מזכיר שראינו את הפרטים של קומיט מסוים עם git cat-file למשל:

$ git cat-file commit 01d577

tree faf16da0f96b03f72e02fefe225cf44bdac6b9f1

parent a6fabeb7560cc07fc5d8dcba9531e5ed51f08ba2

author shay <shay@tocode.co.il> 1534660429 -0700

committer shay <shay@tocode.co.il> 1534660429 -0700

adding flowers

וכאן קל לראות שהקומיט הקודם היה a6fabe ואתם יכולים להמשיך את התרגיל הזה עד תחילת הפרויקט. הענף השני hello-branches פשוט התקדם לכיוון אחר ולכן הליכה אחורה מ master לעולם לא תגיע אליו.

בשביל לראות את כל הקומיטים מכל הענפים נוכל עדיין להשתמש ב reflog:

$ git reflog

01d5778 (HEAD -> master) HEAD@{0}: commit: adding flowers

a6fabeb HEAD@{1}: checkout: moving from hello-branches to master

e0646f1 (hello-branches) HEAD@{2}: commit: adding a list of trees

a6fabeb HEAD@{3}: checkout: moving from master to hello-branches

a6fabeb HEAD@{4}: commit: ADD new file that calculates fibonacci numbers

4ff8942 HEAD@{5}: reset: moving to HEAD^

ef29b25 HEAD@{6}: commit: CHANGED demo.py to print fibonacci number

4ff8942 HEAD@{7}: reset: moving to HEAD

4ff8942 HEAD@{8}: reset: moving to HEAD

...

לסיכום ענפים מאפשרים לנו לעבוד במקביל על מספר גירסאות ולהתקדם עם כל אחת לכיוון שלה. בשיעור הבא נראה מה קורה כששני ענפים נפגשים שוב ואחר כך נלמד איך משתמשים במנגנון הזה לשיפור תהליכי העבודה בגיט.

מיזוג מהיר

המיזוג הכי קל בין ענפים קורה כששניהם נמצאים על אותו רצף. נראה דוגמא על הפרויקט שלנו. בהנחה שאתם עכשיו על ענף master צרו ענף חדש בשם ff ועברו אליו:

$ git checkout -b ff

Switched to a new branch 'ff'

בענף החדש כנסו לקובץ demo.py ועדכנו את תוכנו לקוד הבא:

def twice(x):

return x \* 2

print(twice(10))

הכניסו את השינוי למאגר:

$ git commit -a -m 'FIX demo.py to print 20'

[ff 1c5aafa] FIX demo.py to print 20

1 file changed, 1 insertion(+), 2 deletions(-)

(\*) אגב משמעות המתג **-a** שהעברתי לקומיט היא קיצור דרך ל add: מתג זה גורם לקומיט להוסיף אוטומטית את כל הקבצים בתיקיה שיש בהם שינויים לקומיט הבא. המתג לא יוסיף קבצים חדשים לפרויקט רק כאלה שגיט כבר מכיר.

ונפתח את הלוג:

$ git log -n 3

commit 1c5aafaee90c16f75816118f9ba5d14433256fa0 (HEAD -> ff)

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 23:52:53 2022 -0700

FIX demo.py to print 20

commit 01d577819808d2a5b019cd4a648625114e50a7aa (master)

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 23:33:49 2022 -0700

adding flowers

commit a6fabeb7560cc07fc5d8dcba9531e5ed51f08ba2

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 12:06:41 2022 -0700

ADD new file that calculates fibonacci numbers

הענף ff מתיחס לקומיט 1c5aaf. ההורה של קומיט זה הוא 01d577 שעליו בדיוק מצביע הענף master. במקרה כזה הפיצול כאן הוא למראית עין בלבד: אם master יתקדם קומיט אחד קדימה שני הענפים יצביעו על אותו קומיט.

במצב כזה אפשר למזג את שני הענפים והאפקט של המיזוג יהיה פשוט להזיז את master לקומיט העדכני ביותר. נפעיל את הפקודות:

$ git checkout master

Switched to branch 'master'

$ git merge ff

Updating 01d5778..1c5aafa

Fast-forward

demo.py | 3 +--

1 file changed, 1 insertion(+), 2 deletions(-)

הדבר הראשון שעשינו היה לחזור לענף master. כשאנחנו בתוך ענף זה נרצה למזג את הקומיטים החדשים מהענף ff. פקודת merge של גיט מקבלת ענף וממזגת את הקומיטים החדשים בו אל תוך הענף הנוכחי.

הפלט Fast-Forward אומר לנו שבוצע מיזוג מסוג Fast Forward. כל מה שהיה צריך לעשות הוא לקדם את הענף הנוכחי לקומיט החדש יותר ועכשיו שני הענפים מצביעים על אותו קומיט וכך נפגשו שוב.

אגב אחרי שסיימנו למזג ענף והוא לא מוסיף מידע חדש למערכת אפשר למחוק אותו עם:

$ git branch -d ff

Deleted branch ff (was 1c5aafa).

2. מיזוג הדיף המשולש

לא תמיד החיים כל כך פשוטים. זוכרים את הענף hello-branches שיצרנו בשיעור הקודם? אני מזכיר:

$ git branch

hello-branches

\* master

הענף hello-branches התקדם בקומיט אחד ואז חזרנו ל master וקידמנו גם אותו בקומיט אחר. זה אומר שאי אפשר לקדם אף אחד מהם כדי להגיע לשני: הם פשוט לא על אותו רצף.

במצב כזה גיט יצטרך לייצר קומיט מיזוג. זהו קומיט חדש שמכיל את כל השינויים משני הענפים באמצעות מנגנון שנקרא Three Way Diff (או הדיף המשולש).

שיטת הדיף המשולש אומרת שגיט מסתכל על הקומיט האחרון בכל ענף ועל הקומיט האחרון המשותף לשניהם ומשווה את שלושת הגירסאות. הקומיט שיווצר יכיל את כל השינויים שהכניס כל אחד משני הענפים לפרויקט ביחס לקומיט האחרון המשותף לשניהם.

נראה את זה על הפרויקט שלנו. כתבו:

$ git merge -m 'merging back hello-branches' hello-branches

Merge made by the 'recursive' strategy.

trees.txt | 3 +++

1 file changed, 3 insertions(+)

create mode 100644 trees.txt

נוצר קומיט חדש ובו הקובץ trees.txt. אפשר לראות את זה גם בלוג:

$ git log -n 3

commit 1e62f343d07aa79c8ad71aafc5c75e5b7e272ce9 (HEAD -> master)

Merge: 1c5aafa e0646f1

Author: shay <shay@tocode.co.il>

Date: Sun Aug 19 00:10:15 2022 -0700

merging back hello-branches

commit 1c5aafaee90c16f75816118f9ba5d14433256fa0

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 23:52:53 2022 -0700

FIX demo.py to print 20

commit 01d577819808d2a5b019cd4a648625114e50a7aa

Author: shay <shay@tocode.co.il>

Date: Sat Aug 18 23:33:49 2022 -0700

adding flowers

אז אומנם master ו hello-branches עדיין לא על אותו רצף אבל זה לא נורא. כל השינויים ש hello-branches הכניס נמצאים גם ב master, ולכן זה זמן טוב למחוק את hello-branches:

$ git branch -d hello-branches

Deleted branch hello-branches (was e0646f1).

3. מיזוג שינויים באותו קובץ

מצב פחות נעים של מיזוג קורה כששני ענפים התקדמו לכיוונים שונים אבל עדכנו את אותו מידע. בואו נראה את זה על הפרויקט שלנו. תחילה ניצור ענף חדש בשם add-comments:

$ git checkout -b add-comments

Switched to a new branch 'add-comments'

כשאתם בענף החדש כנסו לקובץ demo.py וכתבו בו את התוכן הבא:

def twice(x):

"""

twice is a function that takes a number and returns it multiplied by 2

:param x: number

:return: x \* 2

"""

return x \* 2

print(twice(10))

בצעו קומיט לשינוי שלכם:

$ git commit -a -m 'add comments to demo.py'

[add-comments db89a46] add comments to demo.py

1 file changed, 5 insertions(+)

חיזרו ל master:

$ git checkout master

Switched to branch 'master'

כנסו לקובץ demo.py וכתבו בו את התוכן הבא:

def twice(x):

return x \* 2

def thrice(x):

return x \* 3

print(twice(10))

וכמובן בצעו קומיט לשינוי:

$ git commit -a -m 'ADD a new function thrice to demo.py'

[master e3a76e8] ADD a new function thrice to demo.py

1 file changed, 3 insertions(+)

ונעבור למיזוג- הפעם שני הענפים כוללים שינויים באותו קובץ אבל במקומות שונים בקובץ. גיט מספיק חכם כדי להבין את זה ושימו לב מה קורה:

$ git merge add-comments

Auto-merging demo.py

Merge made by the 'recursive' strategy.

demo.py | 5 +++++

1 file changed, 5 insertions(+)

גיט זיהה ששני הקבצים שינו את demo.py אבל במקומות שונים. אחד הוסיף פונקציה והשני הוסיף הערות לפונקציה קיימת. זיהוי זה עבד בגלל שגיט לא השווה את שני השינויים אחד לשני אלא כל אחד מהשינויים הושווה בנפרד לקומיט האחרון המשותף בין הענפים.

הקובץ demo.py אחרי המיזוג נראה כך וכולל את השינויים משני הפיתוחים:

def twice(x):

"""

twice is a function that takes a number and returns it multiplied by 2

:param x: number

:return: x \* 2

"""

return x \* 2

def thrice(x):

return x \* 3

print(twice(10))

בשביל הסדר הטוב נמחק את הענף add-comments ונמשיך לדוגמא האחרונה בשיעור:

$ git branch -d add-comments

4. התמודדות עם קונפליקטים

המצב הכי פחות נעים זה כששני אנשים משנים ממש את אותן שורות בשני ענפים שונים. נראה איך זה קורה ואפילו נעשה את זה במספר קבצים.

צרו ענף חדש בשם evil-branch ועברו אליו:

$ git checkout -b evil-branch

Switched to a new branch 'evil-branch'

בתוך הענף החדש עדכנו את הקובץ demo.py תוך שינוי שורת ההערות הקיימת וכיתבו כתוכן הקובץ את הטקסט:

def twice(x):

"""

twice multiplies a number by 2

:param x: number

:return: x \* 2

"""

return x \* 2

def thrice(x):

return x \* 3

print(twice(10))

בשביל שיהיה מעניין ניכנס לקובץ trees.txt ונשנה שם את סדר העצים:

willow

brass

oak

קומיט וחזרה ל master:

$ git commit -a -m 'done evil stuff'

[evil-branch 9bc87d1] done evil stuff

2 files changed, 3 insertions(+), 3 deletions(-)

$ git checkout master

ב master נשנה שוב את שני הקבצים אבל הפעם בצורה שונה. בקובץ demo.py כתבו את התוכן הבא:

def twice(x):

"""

:param x: number

:return: x \* 2

"""

return x \* 2

def thrice(x):

return x \* 3

print(twice(10))

ובקובץ העצים הוסיפו עוד שני עצים בתחילת הקובץ:

cherry

apple

oak

willow

brass

קומיט ואנחנו מוכנים להמשיך:

$ git commit -a -m 'ADDing trees and fixed comments in demo'

[master d64f427] ADDing trees and fixed comments in demo

2 files changed, 2 insertions(+), 1 deletion(-)

עכשיו נפעיל את פקודת המיזוג ונקבל:

$ git merge evil-branch

Auto-merging trees.txt

CONFLICT (content): Merge conflict in trees.txt

Auto-merging demo.py

CONFLICT (content): Merge conflict in demo.py

Automatic merge failed; fix conflicts and then commit the result.

גיט מזהה בעיות במיזוג בשני הקבצים: trees.txt ו demo.py. זה ברור, הרי שני הענפים עדכנו בדיוק את אותן השורות בשני הקבצים ולכן אין דרך לדעת איזה גירסא היא הנכונה. במצב כזה גיט צריך שאתם תתערבו ותגידו לו במה לבחור.

אחרי ביצוע מיזוג כזה שנכשל הקבצים לא במצב טוב. הם כוללים את התוכן משתי הגירסאות בתחביר שמוכר טוב לאנשי יוניקס אבל לא להרבה אחרים, למשל הקובץ demo.py נראה כך:

def twice(x):

"""

<<<<<<< HEAD

=======

twice multiplies a number by 2

>>>>>>> evil-branch

:param x: number

:return: x \* 2

"""

return x \* 2

def thrice(x):

return x \* 3

print(twice(10))

בעברית זה אומר שבגירסת HEAD, כלומר הגירסא שממנה התחלנו את המיזוג, שורה מסוימת לא מופיעה (כי החצים וסימן השווה צמודים), אבל אותה השורה בגירסת evil כוללת את הטקסט שמופיע בין סימני השווה לחצים של evil-branch.

אם אתם מרגישים מספיק נוח להיכנס ולערוך את הקובץ למצב הרצוי יכולים לעשות זאת כעת בתוך ה IDE. אם לא נוכל להשתמש בכלי ה merge שהתקנו בתחילת הקורס. כתבו:

$ git mergetool

וגיט יפתח לכם את הכלי הגרפי שהגדרתם לצורך ביצוע מיזוג. בכלי זה תוכלו לזהות שורות שהשתנו או נמחקו ולאחד את הקבצים. המיזוג הוא בין שלוש גירסאות: בצד שמאל הגירסא שלכם, בצד ימין הגירסא שאיתה אתם רוצים למזג ובאמצע הגירסא המשותפת האחרונה של שני הענפים.

בחרו את הטקסט מאיזה גירסא שאתם רוצים ושימרו. הגירסא הסופית של המיזוג היא מה שמופיע באמצע, או במילים אחרות אתם בוחרים איך לקדם את הגירסא המשותפת האחרונה כך שתתאים לאחת הגירסאות מאחד הענפים.

בסיום השמירה תוכלו לראות שהקבצים במצב הנכון ולוודא בסטטוס שיש שינויים בשני הקבצים שמיזגתם:

$ git status

On branch master

All conflicts fixed but you are still merging.

(use "git commit" to conclude merge)

Changes to be committed:

modified: demo.py

modified: trees.txt

Untracked files:

(use "git add <file>..." to include in what will be committed)

.idea/

\_\_pycache\_\_/

demo.py.orig

trees.txt.orig

נכניס את השינויים לקומיט חדש ולאחר מכן נמחק את קבצי ה orig שכלי המיזוג השאיר בתיקיה:

$ git commit -m 'merged evil branch'

[master e14ebb5] merged evil branch

$ rm \*.orig

מעניין לראות איך המיזוג השפיע על הלוג. פתחו את הלוג ותוכלו לראות דבר מפתיע:

$ git log -n 3

commit e14ebb54063415fb86046826b922b074bdec56dd (HEAD -> master)

Merge: d64f427 9bc87d1

Author: shay <shay@tocode.co.il>

Date: Sun Aug 19 00:53:51 2022 -0700

merged evil branch

commit d64f427991fafcff61a06e3c2bb779e7e96dca06

Author: shay <shay@tocode.co.il>

Date: Sun Aug 19 00:37:20 2022 -0700

ADDing trees and fixed comments in demo

commit 9bc87d1475402df86a665d546714c1e9cbb7d076 (evil-branch)

Author: shay <shay@tocode.co.il>

Date: Sun Aug 19 00:35:23 2022 -0700

done evil stuff

אנחנו יודעים ששני הענפים התפצלו ולכן evil-branch לא אמור להיות באותו רצף כמו master. ובכל זאת אנחנו רואים אותו בלוג.

ההסבר טמון בקומיט המיזוג שיצרנו. נתבונן בקומיט זה עם git cat-file:

$ git cat-file commit e14ebb

tree 2c0659b65b85495bec0e6b6b99acd438f55be568

parent d64f427991fafcff61a06e3c2bb779e7e96dca06

parent 9bc87d1475402df86a665d546714c1e9cbb7d076

author shay <shay@tocode.co.il> 1534665231 -0700

committer shay <shay@tocode.co.il> 1534665231 -0700

merged evil branch

לקומיט מיזוג יש שני הורים: כל אחד מהענפים אותם מיזגנו. בלוג מוצגים שני הרצפים במקביל.

צפיה רגילה בלוג במצב כזה יכולה להיות מבלבלת ולכן לגיט לוג יש עוד שני מתגים שעוזרים להבנה:

$ git log -n 4 --oneline --graph

\* e14ebb5 (HEAD -> master) merged evil branch

|\

| \* 9bc87d1 (evil-branch) done evil stuff

\* | d64f427 ADDing trees and fixed comments in demo

|/

\* 439d354 Merge branch 'add-comments'

בקומיט 439d354 שני הענפים master ו evil-branch התפצלו ולאחר מכן קומיט e14ebb5 איחד ביניהם שוב.

ניצול טוב של מנגנון יצירת ומיזוג הענפים של גיט יעזור לנו לפתח בצורה יעילה יותר כולל פיתוח במקביל של מספר פיצ'רים. בשיעור זה נראה שיטה יחסית פשוטה לפיתוח כזה וגם נדבר איך להרחיב אותה למקרים יותר מורכבים.

רעיון 1: כדאי לפתח על Branch נפרד

מאחר וקל מאוד לייצר ענפים חדשים ולמזג אותם חזרה המסקנה הראשונה היא שכדאי לנו לפתח כל פיצ'ר על ענף משלו: כשמתחילים לעבוד על פיצ'ר יוצרים ענף חדש עבור פיצ'ר זה, בסיום העבודה ממזגים את הפיצ'ר חזרה ל master.

בצורה כזו קל לנו לעבוד על מספר פיצ'רים במקביל: כל פעם שאתם רוצים להתקדם בפיצ'ר מסוים פשוט עוברים לענף שלו ומתקדמים בפיתוח. מפתחים אחרים בצוות או אתם עצמכם יכולים לעבוד על פיצ'רים שונים במקביל ואף אחד לא משנה דברים שאחר מתבסס עליהם.

כל עוד הפיצ'רים משנים חלקים אחרים בפרויקט לא תהיה לנו בעיה למזג אותם ל master בסיום העבודה ואז למחוק אותם.

כדי להתחיל לעבוד על פיצ'ר נכתוב:

$ git checkout -b my-feature

בסיום העבודה נמזג את הפיצ'ר חזרה ל master:

$ git checkout master

$ git merge my-feature

ואחרי שהמיזוג הצליח נמחק את הענף כי אין בו יותר צורך. ממילא כל התיעוד נמצא בגרף ההיסטוריה הראשי:

$ git branch -d my-feature

2. רעיון 2: כדאי לתקן באגים קריטיים בבראנצ'ים נפרדים

מה קורה כשאנחנו באמצע פיתוח ולקוח מצלצל עם תקלה קריטית? במקום לתקן ישירות על master ניצור גם לזה ענף נפרד ונמזג אותו ל master:

$ git checkout -b critical-bugfix

$ ... fix bug

$ git checkout master

$ git merge critical-bugfix

מתכנתים שעובדים על פיצ'רים אחרים יכולים לבחור למזג את ה critical-bugfix לפיצ'רים שלהם (או שלא, אם זה לא רלוונטי עבורם). בסיום כל המיזוגים נמחק את הענף:

$ git branch -d critical-bugfix

3. הרחבה: כדאי לתייג גירסאות מרכזיות כשאתם מוכנים לעלות לייצור

שני הרעיונות הראשונים לבד כבר יעזרו לכם לפתח בצורה יעילה יותר את המערכת. בשביל הסדר הטוב מתכנתים רבים אוהבים לדעת על איזה גירסא הם מדברים ולכן אוהבים לתת שם שיישמר לקומיט מסוים. בגיט ניתן לעשות זאת עם הפקודה git tag. כשענף master מוכן לעלות לייצור הפעילו:

$ git tag 1.0.0

המספר 1.0.0 הוא מספר גירסא ואתם יכולים לבחור שם כל שם או מספר שתרצו. נחזור ללוג ונקבל:

$ git log -n 2

commit e14ebb54063415fb86046826b922b074bdec56dd (HEAD -> master, tag: 1.0.0)

Merge: d64f427 9bc87d1

Author: shay <shay@tocode.co.il>

Date: Sun Aug 19 00:53:51 2022 -0700

merged evil branch

commit d64f427991fafcff61a06e3c2bb779e7e96dca06

Author: shay <shay@tocode.co.il>

Date: Sun Aug 19 00:37:20 2022 -0700

ADDing trees and fixed comments in demo

מכאן נוכל תמיד לחזור לגירסא 1.0.0 כדי לבדוק דברים או לתקן באגים שהופיעו בגירסא זו.

4. הרחבה: כדאי לפתח על ענף פיתוח מרכזי

ככל שהפרויקט גדל זה הופך להיות קשה לזכור איזה תיקוני באגים קריטיים אתם מכניסים לכל Feature Branch. כשיש לכם הרבה מתכנתים שעובדים על הרבה פיצ'רים משתלם לייצר ענף נוסף שנקרא ענף הפיתוח (יכולים לקרוא לו dev או development) ואז כל פעם שיש באג קריטי שתיקנתם בענף הראשי ושעדיין רלוונטי גם לגירסת הפיתוח הבאה לשלב אותו בענף הפיתוח.

בצורה כזו כל מי שעובד על Feature Branch מתחיל יום עבודה במיזוג כל התיקונים מענף הפיתוח ואז ממשיך ורואה שכלום לא נשבר. ה Feature Branches ימוזגו חזרה לענף הפיתוח כשהם מוכנים.

בשלב מסוים אחרי שיש לכם את כל הפיצ'רים עובדים בענף הפיתוח אתם יוצרים ממנו ענף חדש שנקרא rc או testing לצורך בדיקות. לענף זה ייכנסו רק תיקוני באגים שרלוונטים לפיצ'רים החדשים שפיתחתם. תיקוני באגים קריטיים שנכנסים ל master בזמן העבודה ישולבו גם הם לכאן.

כשענף הבדיקות מתייצב מתייגים את הגירסא וממזגים אותו ל master לצורך שיחרור הגירסא הבאה.

5. קריאת המשך: git flow

המנגנון שתואר כאן הוא גירסא קצת יותר פשוטה של רצף עבודה שנקרא git flow. אם אתם מחפשים לגלות שיטות עבודה נוספות או הרחבות נוספות למנגנון ממליץ לקרוא גם את התיאור המלא של git flow בקישור:

<https://nvie.com/posts/a-successful-git-branching-model/>

מתג נחמד של diff מאפשר לראות שינויים בין ענפים בצורה גרפית ובכל הקבצים כפי שנראה בשיעור זה.

הצגת הבדלים בצורה גרפית

נחזור לפרויקט שלנו ואני מזכיר שלא מזמן מיזגנו את evil-branch חזרה לענף הראשי כך שהרשומות האחרונות בלוג נראות כך:

$ git log --graph --oneline -n 3

\* e14ebb5 (HEAD -> master, tag: 1.0.0) merged evil branch

|\

| \* 9bc87d1 (evil-branch) done evil stuff

\* | d64f427 ADDing trees and fixed comments in demo

|/

נחזור לנקודה שלפני המיזוג באמצעות:

$ git checkout d64f427

Note: checking out 'd64f427'.

You are in 'detached HEAD' state. You can look around, make experimental

changes and commit them, and you can discard any commits you make in this

state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may

do so (now or later) by using -b with the checkout command again. Example:

git checkout -b <new-branch-name>

HEAD is now at d64f427 ADDing trees and fixed comments in demo

גיט מזכיר לנו שחזרנו לעבר וקומיטים שנעשה עכשיו יאבדו מאחר והם מחוץ לרצף ההיסטוריה. לא נורא ממילא באנו רק בשביל להסתכל.

הפעילו את הפקודה:

$ git difftool -d evil-branch

וגיט יפתח באופן אוטומטי את הכלי הגרפי להצגת שינויים ויראה לכם בחירה בין שני הקבצים שהשתנו. לחיצה על כל אחד מהם תציג את כל השינויים בין master לבין evil-branch בקובץ זה.

2. הצגת הבדלים בשורת הפקודה

אם לא הגדרתם כלי גרפי להצגת הבדלים או שסתם לא מתחשק לכם לחכות שיפתח אפשר גם להציג הבדלים אלה משורת הפקודה. הקלידו:

$ git diff evil-branch

diff --git a/demo.py b/demo.py

index fa7d5f0..f86eb20 100644

--- a/demo.py

+++ b/demo.py

@@ -1,6 +1,5 @@

def twice(x):

"""

- twice multiplies a number by 2

:param x: number

:return: x \* 2

"""

diff --git a/trees.txt b/trees.txt

index 146d443..4394318 100644

--- a/trees.txt

+++ b/trees.txt

@@ -1,3 +1,5 @@

-willow

-brass

+cherry

+apple

oak

+willow

+brass

\ No newline at end of file

וקיבלתם את כל ההבדלים בין שני הענפים.

תזכורת: מה קורה לקומיט בלי ראש

את השיעור הקודם סיימנו עם HEAD שמצביע על קומיט d64f42. אפשר לוודא את זה בקלות עם הפקודה:

$ cat .git/HEAD

d64f427991fafcff61a06e3c2bb779e7e96dca06

נניח ששכחנו את זה ואנחנו חושבים שאנחנו בענף הראשי. נכנס לקובץ readme.txt ונכתוב בו את התוכן הבא:

Hello World

-----------

To start working on the project you'll first need to install the mdoules it uses by runnign:

pip install -r requirements.txt

2. Contribute

Want to help? Cool drop me a line to my email and if you know git I'll let you write some code ```

חזרה לשורת הפקודה ונקליד:

$ git commit -a -m 'ADD exlpain how to contribute'

[detached HEAD 7d1a937] ADD exlpain how to contribute

1 file changed, 3 insertions(+)

וגיט מזכיר לי שהקומיט בוצע במצב Detached Head. חזרה ל master כדי לתקן:

$ git checkout master

Warning: you are leaving 1 commit behind, not connected to

any of your branches:

7d1a937 ADD exlpain how to contribute

If you want to keep it by creating a new branch, this may be a good time

to do so with:

git branch <new-branch-name> 7d1a937

Switched to branch 'master'

ותראו מה קרה כאן: גיט מזכיר לי שאני חוזר מהעבר להווה אבל שבדרך יצרתי קומיט שמספרו 7d1a937. קומיט זה לא שייך לאף ענף ולכן יאבד לנצח. בשביל להציל אותו נרצה ליצור ענף חדש שמצביע עליו וגיט גם אומר לי איזה פקודה להריץ. בואו נריץ אותה:

$ git branch contrib 7d1a937

הפעלת git branch כדי לראות שהענף נוצר:

$ git branch

add-comments

contrib

evil-branch

\* master

והנה 3 הרשומות האחרונות בענף contrib:

$ git log -n 3 contrib

commit 7d1a9373c55c2f3077860ce451e4ae1a4e7939e5 (contrib)

Author: shay <shay@tocode.co.il>

Date: Sun Aug 19 02:06:50 2022 -0700

ADD exlpain how to contribute

commit d64f427991fafcff61a06e3c2bb779e7e96dca06

Author: shay <shay@tocode.co.il>

Date: Sun Aug 19 00:37:20 2022 -0700

ADDing trees and fixed comments in demo

commit 439d354438c390a479c5205e41404cff6e24d076

Merge: e3a76e8 db89a46

Author: shay <shay@tocode.co.il>

Date: Sun Aug 19 00:27:31 2022 -0700

Merge branch 'add-comments'

עכשיו אתם יכולים לחזור ל contrib ולהמשיך לעבוד על הפיצ'ר החדש שלכם, וכשזה יהיה גמור למזג אותו חזרה ל master:

$ git checkout master

Already on 'master'

$ git merge contrib

Merge made by the 'recursive' strategy.

readme.txt | 3 +++

1 file changed, 3 insertions(+)

לבסוף צפיה בגרף תראה לנו:

$ git log --graph --oneline -n 5

\* 1a36920 (HEAD -> master) Merge branch 'contrib'

|\

| \* 7d1a937 (contrib) ADD exlpain how to contribute

\* | e14ebb5 (tag: 1.0.0) merged evil branch

|\ \

| |/

|/|

| \* 9bc87d1 (evil-branch) done evil stuff

\* | d64f427 ADDing trees and fixed comments in demo

קצת מבולגן אבל בכל זאת כולל את כל השינויים.

לגיט יש שתי דרכים למיזוג ענפים שהתפצלו: על הראשונה, merge, כבר דיברנו בשיעורים קודמים ובשיעור זה נעבור לדבר על rebase שהיא הדרך השניה.

הכנת הסביבה

בשביל להבין את ריבייס נרצה תחילה לדבר על הרעיון של Patch (או בעברית טלאי). טלאי הוא רשימת השינויים הנדרשים כדי לעבור ממצב A למצב B של פרויקט. גיט יודע לייצר טלאי כדי לעבור בין כל קומיט לכל קומיט אחר.

ניצור פרויקט חדש בשביל הדוגמא בשיעור זה. כנסו לתיקיה חדשה לגמרי והתחילו בה פרויקט גיט חדש באמצעות:

$ git init

בפרויקט החדש ניצור קובץ אחד ונרשום בו שורה טקסט יחידה. אפשר לעשות את זה ישירות משורת הפקודה עם:

$ echo one > myfile.txt

נוסיף את הקובץ ונבצע קומיט:

$ git add myfile.txt

$ git commit -m 'initial commit'

נמשיך לעשות את זה עוד פעמיים עם עוד שני שינויים:

$ echo two > myfile.txt

$ git commit -a -m 'second commit'

$ echo three >> myfile.txt

$ git commit -a -m 'third commit'

שימו לב שבפעם השניה כתבתי פעמיים את סימן החץ. התוצאה היא קובץ שיש בו שתי שורות:

$ cat myfile.txt

two

three

ויומן פעולות שנראה כך:

$ git log

commit 4fc7ffb05361bea49dc043e9fbb95a05a85d9871 (HEAD -> master)

Author: shay <shayd@gmail.com>

Date: Sun Aug 19 14:57:40 2022 +0300

third commit

commit e4c12f55e70724f2f321b89fcdb08e66ecd37439

Author: shay <shayd@gmail.com>

Date: Sun Aug 19 14:57:31 2022 +0300

second commit

commit d3f942f777c003a6afbe6cb20498631d41b36b86

Author: ynonp <ynonperek@gmail.com>

Date: Sun Aug 19 14:56:17 2018 +0300

initial commit

2. יצירת patch בין קומיטים

הפעילו את הפקודה:

git format-patch d3f942..4fc7ff

אבל החליפו את מזהי הקומיטים לקומיט הראשון והאחרון שמופיעים אצלכם בלוג. התוצאה היא יצירת שני קבצים חדשים על הדיסק:

0001-second-commit.patch

0002-third-commit.patch

כל אחד מהם הוא קובץ טלאי, כלומר קובץ שמתאר מה צריך לעשות כדי להגיע מהקומיט שהיה לפניו לקומיט שלו. שני שלושה קומיטים ולכן שני קבצי טלאי.

נתבונן בקובץ הטלאי הראשון:

$ cat 0001-second-commit.patch

From e4c12f55e70724f2f321b89fcdb08e66ecd37439 Mon Sep 17 00:00:00 2001

From: shay <shayd@gmail.com>

Date: Sun, 19 Aug 2022 14:57:31 +0300

Subject: [PATCH 1/2] second commit

---

myfile.txt | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

diff --git a/myfile.txt b/myfile.txt

index 5626abf..f719efd 100644

--- a/myfile.txt

+++ b/myfile.txt

@@ -1 +1 @@

-one

+two

--

2.15.2 (Apple Git-101.1)

שימו לב שהקובץ מתאר בדיוק מה צריך לעשות כדי לעבור מהקומיט שהיה לפניו לזה שהיה אחריו: הוא אומר לנו שצריך למחוק את השורה הראשונה בקובץ שקודם היה כתוב בה one ולהחליף אותה בטקסט two.

הטלאי השני נראה כך:

$ cat 0002-third-commit.patch

From 4fc7ffb05361bea49dc043e9fbb95a05a85d9871 Mon Sep 17 00:00:00 2001

From: shay <shayd@gmail.com>

Date: Sun, 19 Aug 2022 14:57:40 +0300

Subject: [PATCH 2/2] third commit

---

myfile.txt | 1 +

1 file changed, 1 insertion(+)

diff --git a/myfile.txt b/myfile.txt

index f719efd..1946f04 100644

--- a/myfile.txt

+++ b/myfile.txt

@@ -1 +1,2 @@

two

+three

--

2.15.2 (Apple Git-101.1)

והוא אומר שצריך להוסיף שורה חדשה לקובץ עם הטקסט three. אני יודע שלא תמיד קל לקרוא את ההוראות האלה אבל גם לא ממש צריך. אתם לא קהל היעד של ההוראות אלא גיט עצמו.

3. הפעלת ה patch שיצרנו במקום אחר

כי אתם מבינים מרגע שיש לנו הוראות שמראות איך לעבור מקומיט אחד לאחר אפשר להשתמש בהן ולהפעיל אותן גם על חלקים אחרים בפרויקט.

נחזור לגירסא הראשונה של הפרויקט וניצור ענף חדש להמשך פיתוח שיוצא ממנה:

$ git branch dev d3f942f

$ git checkout dev

עכשיו נייצר קובץ חדש כאן ב dev בשם otherfile.txt ונכתוב בו את קצת טקסט:

$ echo some other text > otherfile.txt

$ git add otherfile.txt

$ git commit -m 'adding another file'

$ git log

commit 833bf8c9760c9bf004a6e03186483c227f631813 (HEAD -> dev)

Author: shay <shayd@gmail.com>

Date: Sun Aug 19 15:06:24 2022 +0300

adding another file

commit d3f942f777c003a6afbe6cb20498631d41b36b86

Author: shay <shayd@gmail.com>

Date: Sun Aug 19 14:56:17 2022 +0300

initial commit

אתם זוכרים שאפשר לאחד את שני הענפים עם merge וכך להעביר את otherfile.txt לענף הראשי. אנחנו ננסה כאן תרגיל אחר כדי להבין את ריבייס.

תחילה נמחק את הטלאים ששמורים אצלנו בתיקיה ונייצר טלאי חדש עבור השינוי:

$ rm \*.patch

$ git format-patch d3f942..833bf8

0001-adding-another-file.patch

נחזור ל master ושם נפעיל את ה patch שיצרנו:

$ git checkout master

$ cat \*.patch | git am

התוצאה: גיט לקח את הטלאי, יישם אותו על הפרויקט שלנו ויצר מזה קומיט חדש. את הענף הישן אפשר למחוק וכל השינויים נכנסו ל master.

חשוב לשים לב שהקומיט החדש שנוצר ב master, למרות שיש לו את אותו תוכן, אותה הודעה ואותם קבצים - הוא עדיין קומיט שונה מזה שהיה ב dev ואפשר לראות שהמזהים של שני הקומיטים הם שונים. גיט לא העביר קומיט מ dev ל master, אלא יצר קומיט חדש עם אותם השינויים ואותו שמר ב master.

4. ועכשיו באמצעות git rebase

הפקודה git rebase עושה את כל התהליך שראינו אבל בצורה אוטומטית. ניצור שוב ענף חדש הפעם בשם stable שיוצא שוב מהקומיט הראשון וניצור בו מספר שינויים:

$ git branch stable d3f942f

$ git checkout stable

$ echo 1 > one.txt

$ git add one.txt

$ git commit -m 'ADD file one.txt'

$ echo 2 > two.txt

$ git add two.txt

$ git commit -m 'ADD file two.txt'

$ echo 3 > three.txt

$ git add three.txt

$ git commit -m 'ADD file three.txt'

$ rm \*.patch

$ ls -1

myfile.txt

one.txt

three.txt

two.txt

$ git branch

dev

master

\* stable

פעולת ריבייס תאפשר לנו להזיז את הענף stable כך שבמקום להתחיל מקומיט d3f942f (הקומיט הישן ביותר בעץ) נוכל להתחיל את הענף מהקומיט האחרון של master. כדי להגיע לשם ריבייס תחשב את כל הטלאים בין d3f942f, תשמור אותם בצד ואז תפעיל אותם אחד אחרי השני על master. כל הפעלת טלאי תיצור קומיט חדש ובסיום נקבל את השינויים משני הענפים במקום אחד.

הפקודה הראשונה תראה לנו איזה טלאים גיט מתכנן להפעיל:

$ git rebase --interactive

אפשר לבחור להפעיל את כל הטלאים או רק חלק מהם. שימרו את הקובץ וכנסו ל reflog כדי לראות מה קרה שם:

$ git reflog

48a8fda (HEAD -> stable) HEAD@{0}: rebase -i (finish): returning to refs/heads/stable

48a8fda (HEAD -> stable) HEAD@{1}: rebase -i (pick): ADD file 3

2f0a636 HEAD@{2}: rebase -i (pick): ADD file 2

b4262a1 HEAD@{3}: rebase -i (pick): ADD file 1

accbed1 (master) HEAD@{4}: rebase -i (start): checkout master

bebcd66 HEAD@{5}: commit: ADD file 3

7d9f604 HEAD@{6}: commit: ADD file 2

7720876 HEAD@{7}: commit: ADD file 1

c9e84ff HEAD@{8}: checkout: moving from master to stable

accbed1 (master) HEAD@{9}: am: adding another file

9a83034 HEAD@{10}: checkout: moving from dev to master

7ca9a4e (dev) HEAD@{11}: commit: adding another file

c9e84ff HEAD@{12}: checkout: moving from master to dev

9a83034 HEAD@{13}: commit: third commit

e2dd32c HEAD@{14}: commit: second commit

c9e84ff HEAD@{15}: commit (initial): initial commit

שימו לב מה קרה כאן: הקומיט ADD file 1 שהיה בענף stable בוצע מחדש בשורה:

b4262a1 HEAD@{3}: rebase -i (pick): ADD file 1

הוא גם קיבל מזהה חדש. כשאנחנו אומרים בוצע מחדש הכוונה שהטלאי שיוצר אותו הופעל אוטומטית על הענף master וכך נוצר הקומיט b4262a1.

באותו האופן גם הטלאים שיצרו את קומיטים ADD file 2 ו ADD file 3 הופעלו על master ובסוף גיט באופן אוטומטי הזיז את stable כך שיצביע למקום העדכני ביותר שנבנה על master.

מבט על הגרף:

\* 48a8fda (HEAD -> stable) ADD file 3

\* 2f0a636 ADD file 2

\* b4262a1 ADD file 1

\* accbed1 (master) adding another file

\* 9a83034 third commit

\* e2dd32c second commit

\* c9e84ff initial commit

ואנחנו רואים שעברנו ממצב של התפצלות לגרף לינארי. אם נמזג כעת את הענפים גיט יוכל לבצע הרצה קדימה של master ונישאר עם גרף היסטוריה שכולל רצף אחד בלבד:

$ git checkout master

$ git merge stable

Updating accbed1..48a8fda

Fast-forward

one.txt | 1 +

three.txt | 1 +

two.txt | 1 +

3 files changed, 3 insertions(+)

create mode 100644 one.txt

create mode 100644 three.txt

create mode 100644 two.txt

$ git log --graph --oneline

\* 48a8fda (HEAD -> master, stable) ADD file 3

\* 2f0a636 ADD file 2

\* b4262a1 ADD file 1

\* accbed1 adding another file

\* 9a83034 third commit

\* e2dd32c second commit

\* c9e84ff initial commit

5. ההבדל בין rebase ל merge

פעולת rebase דומה לפעולת merge במטרות שלה: שתיהן לוקחות שני ענפים שהתפצלו ועוזרות לחבר אותם יחד. אך כאן בערך הדמיון נגמר.

בפעולת merge החיבור נעשה באמצעות יצירת קומיט חדש שמאחד את השינויים משני הענפים. לקומיט זה שני הורים ולכן רצף הקומיטים בגרף ההיסטוריה מראה את ההתפצלות והאיחוד.

בפעולת rebase החיבור נעשה באמצעות הפעלת טלאים ויצירת קומיטים חדשים על אחד הענפים ואז שכתוב הענף החדש כך שיראה כאילו הוא התחיל מהענף הקודם. התוצאה היא גרף היסטוריה לינארי בו אפשר למחוק כל זכר לענף הפיתוח ולהישאר רק עם הענף המרכזי.

לכן נשתמש בפעולת ריבייס במצבים בהם התחלנו Feature Branch מתוך Feature Branch ואנו צריכים ששני הענפים יראו כאילו נכתבו יחד כשנשלב אותם בענף המרכזי.

שינוי הודעות קומיט ישנות

פעולת ריבייס היא הדרך המתוחכמת ביותר שראינו עד כה לשנות את ההיסטוריה וזה לפעמים מדהים כמה שזה פשוט. בואו נניח למשל שכל מה שאנחנו רוצים הוא לשנות הודעות קומיט ישנות, כלומר להסתכל על יומן האירועים ולהיות מסוגלים לערוך את האירועים ששמורים שם.

שינוי הודעות לעולם לא יוביל לקונפליקטים או צרות כאלה כך שברוב המקרים הטריק הזה יעבוד לכם חלק. שימו לב רק שהטריק מחליף את כל הקומיטים בקומיטים חדשים ולכן כמו כל ריבייס כדאי להיות זהירים אם אתם עובדים בסינכרון עם שרת מרוחק (ונדבר על זה יותר לעומק כשנלמד על עבודה מול שרת מרוחק).

בינתיים בואו נחזור לפרויקט פייתון שלנו וניצור מספר קומיטים חדשים:

$ echo lily >> flowers.txt

$ git commit -a -m 'lily'

$ echo iris >> flowers.txt

$ git commit -a -m 'iris'

עכשיו נפעיל את הפקודה:

$ git rebase --interactive HEAD~2

בתגובה גיט יפתח את עורך הטקסט שמוגדר לכם במערכת עם טקסט שנראה בערך כך:

pick fd938e4 lily

pick 3f2b7cc iris

# Rebase 1a36920..3f2b7cc onto 1a36920 (2 commands)

#

# Commands:

# p, pick <commit> = use commit

# r, reword <commit> = use commit, but edit the commit message

# e, edit <commit> = use commit, but stop for amending

# s, squash <commit> = use commit, but meld into previous commit

# f, fixup <commit> = like "squash", but discard this commit's log message

# x, exec <command> = run command (the rest of the line) using shell

# d, drop <commit> = remove commit

# l, label <label> = label current HEAD with a name

# t, reset <label> = reset HEAD to a label

# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]

# . create a merge commit using the original merge commit's

# . message (or the oneline, if no original merge commit was

# . specified). Use -c <commit> to reword the commit message.

גיט מתחיל את הריבייס באיתור כל השינויים שנמצאים בענף הנוכחי אבל לא נמצאים בקומיט שהעברנו כפרמטר. הקומיט שהעברנו כפרמטר הוא **HEAD~2** כלומר 2 קומיטים אחורה ולכן הקומיטים ברשימה הם בדיוק שני הקומיטים האחרונים בענף. בזכות המתג **-i** גיט רשם לנו את הקומיטים האלה ומאפשר לנו לבחור מה יהיה סדר הפעלת הטלאים ואיך בדיוק להפעיל כל טלאי.

אנחנו נבחר בפעול reword שהיא הפעולה לשינוי הודעת הקומיט. החליפו את התוכן בטקסט הבא (אבל שימרו על המזהים שמופיעים אצלכם) - כלומר החליפו רק את המילה pick במילה reword בכל אחת מהשורות:

reword fd938e4 lily

reword 3f2b7cc iris

אחרי שמירה ויציאה העורך ייפתח שוב כדי שתבחרו תוכן חדש להודעת הקומיט הראשונה ואחרי זה שוב כדי לכתוב תוכן בהודעת הקומיט השניה. בחרו הודעות חדשות וכנסו ללוג כשהפעולה מסתיימת לראות שההודעות החדשות שלכם נשמרו:

$ git log -n 2

commit 699418cfed9f860442c9bcf5298ae1722cc4d788 (HEAD -> master)

Author: shay <shay@tocode.co.il>

Date: Sun Aug 19 12:07:00 2022 -0700

ADD flower: iris

commit b1b75864089da04eaa526fc7707c67129d40176b

Author: shay <shay@tocode.co.il>

Date: Sun Aug 19 12:06:54 2022 -0700

ADD flower: lily

2. איחוד קומיטים

אני נשאר עם הריבייס האינטרקטיבי כדי להראות דוגמא נוספת הפעם של איחוד קומיטים. יש היגיון באיחוד שתי פעולות הוספת הפרחים לפעולת קומיט אחת. במיוחד בפרויקט גדול שיש הרבה שינויים לא בטוח שצריך כל כך הרבה קומיטים.

נפעיל שוב את ריבייס על שני הקומיטים האחרונים:

$ git rebase -i HEAD~2

אבל הפעם השתמשו במילה squash על הקומיט התחתון ברשימה כלומר כתבו (כמובן תוך שמירה על המזהים שמופיעים אצלכם):

pick b1b7586 ADD flower: lily

squash 699418c ADD flower: iris

ואחרי שמירה ויציאה ייפתח שוב עורך טקסט הפעם עם תוכן שמתחיל בשתי ההודעות של שני הקומיטים. עדכנו את ההודעות למשהו שתרגישו איתו בנוח ושמרו שוב. ניכנס ללוג ונראה ששני הקומיטים שלנו נמחקו ובמקומם יש קומיט יחיד עם ההודעה החדשה והשינויים של שני הקומיטים:

$ git log -n 2

commit 08e9be0cd5d381640df516708eca387c0a355bc4 (HEAD -> master)

Author: shay <shay@tocode.co.il>

Date: Sun Aug 19 12:06:54 2022 -0700

ADD flowers: lily and iris

commit 1a369201050a4287982fdb5414ac1bfdec671e13

Merge: e14ebb5 7d1a937

Author: shay <shay@tocode.co.il>

Date: Sun Aug 19 02:11:08 2022 -0700

Merge branch 'contrib'

3. מחיקת קומיט יחד עם כל השינויים שלו מההיסטוריה

אפשרות נוספת של ריבייס היא מחיקת קומיט מסוים מההיסטוריה יחד עם כל השינויים שלו. זה למעשה מאוד פשוט: כל מה שצריך זה למחוק את הקומיט מרשימת הקומיטים לריבייס. בואו ננסה את זה.

תחילה ניצור קומיט חדש בשביל שיהיה מעניין. עדכנו את הטקסט בקובץ readme.txt ולאחר מכן הוסיפו אותו לפרויקט ובצעו קומיט:

$ git add readme.txt

$ git commit -m 'new documentation'

זה המצב אצלי אחרי השינוי:

$ git log -n 3

commit a6d08a4f6c40b2837176310dffec92448ac4174b (HEAD -> master)

Author: shay <shay@tocode.co.il>

Date: Sun Aug 19 12:19:59 2022 -0700

new documentation

commit 08e9be0cd5d381640df516708eca387c0a355bc4

Author: shay <shay@tocode.co.il>

Date: Sun Aug 19 12:06:54 2022 -0700

ADD flowers: lily and iris

commit 1a369201050a4287982fdb5414ac1bfdec671e13

Merge: e14ebb5 7d1a937

Author: shay <shayd@tocode.co.il>

Date: Sun Aug 19 02:11:08 2022 -0700

Merge branch 'contrib'

נפעיל שוב ריבייס אינטרקטיבי על שני הקומיטים האחרונים:

$ git rebase -i HEAD~2

ופשוט נמחק מהרשימה את הקומיט של הפרחים כך שנישאר רק עם שורת קומיט אחת. שמירה, יציאה ומבט נוסף בלוג ותראו מה קיבלנו:

$ git log -n 3

commit 9cfb74fc38a1aa0eabd5e36902d3c42540263718 (HEAD -> master)

Author: shay <shayd@tocode.co.il>

Date: Sun Aug 19 12:19:59 2022 -0700

new documentation

commit 1a369201050a4287982fdb5414ac1bfdec671e13

Merge: e14ebb5 7d1a937

Author: shay <shay@tocode.co.il>

Date: Sun Aug 19 02:11:08 2022 -0700

Merge branch 'contrib'

commit 7d1a9373c55c2f3077860ce451e4ae1a4e7939e5 (contrib)

Author: shay <shayd@tocode.co.il>

Date: Sun Aug 19 02:06:50 2022 -0700

ADD exlpain how to contribute

הקומיט של הפרחים החדשים לא מופיע כאן. בנוסף קובץ הפרחים לא כולל את lily ולא את iris - שני הפרחים החדשים שנוספו באותו קומיט:

$ cat flowers.txt

daisy

jasmine

magnolia

את הקומיטים הישנים עדיין תוכלו למצוא ב reflog אבל מבחינת היסטוריית הפרויקט והיומן הרגיל לא תראו אותם.

קונפליקט בין ענפים

ניצור פרויקט חדש עם שני ענפים מתפצלים כדי ליצור קונפליקט בריבייס. הקלידו את הפקודות הבאות בתיקיה חדשה ריקה:

$ git init

$ touch myfile.txt

$ echo line 1 > myfile.txt

$ echo line 2 >> myfile.txt

$ git add myfile.txt

$ git commit -m 'initial commit'

$ git tag A

$ echo take two > myfile.txt

$ git add myfile.txt

$ git commit -m 'a second try'

$ git branch dev A

$ git checkout dev

$ echo just kidding > myfile.txt

$ echo I can has cheezburger >> myfile.txt

$ git add myfile.txt

$ git commit -m 'a third try from dev branch'

עכשיו הפרויקט מוכן עם שני ענפים ובכל אחד מהם שינויים מתנגשים על אותו קובץ ממש. אנחנו יודעים שאם נעשה merge יווצר לנו Merge Commit בו נוכל לאחד את השינויים. הפעם ננסה להפעיל rebase במקום. הישארו בענף dev וכתבו:

$ git rebase master

First, rewinding head to replay your work on top of it...

Applying: a third try from dev branch

Using index info to reconstruct a base tree...

M myfile.txt

Falling back to patching base and 3-way merge...

Auto-merging myfile.txt

CONFLICT (content): Merge conflict in myfile.txt

error: Failed to merge in the changes.

Patch failed at 0001 a third try from dev branch

The copy of the patch that failed is found in: .git/rebase-apply/patch

Resolve all conflicts manually, mark them as resolved with

"git add/rm <conflicted\_files>", then run "git rebase --continue".

You can instead skip this commit: run "git rebase --skip".

To abort and get back to the state before "git rebase", run "git rebase --abort".

גיט לא התקמצן על הסברים. בקצרה הוא מספר שהוא ניסה להפעיל את הטלאי שיוצר את הקומיט:

Applying: a third try from dev branch

אבל נתקל בבעיה. זה צפוי בהתחשב בנסיבות. אפשר לראות את הקונפליקט בקובץ myfile.txt באמצעות:

$ cat myfile.txt

<<<<<<< HEAD

take two

=======

just kidding

I can has cheezburger

>>>>>>> a third try from dev branch

אם התחרטתם ורוצים להפסיק עכשיו את ה rebase אין שום בעיה. כתבו:

$ git rebase --abort

כדי לחזור לענף dev לפני שהפעלתם את ריבייס.

אם לא התחרטתם תוכלו לתקן את הקובץ ולבחור את הגירסא שמתאימה לכם. בסיום הוסיפו אותו ל Staging Area ולאחר מכן הריצו:

$ git rebase --continue

כדי להמשיך להריץ טלאים נוספים.

שווה לציין שקונפליקטים בריבייס הרבה פעמים ממשיכים לרדוף אתכם בהפעלת כל הטלאים בריבייס. זה קורה בגלל שאם יש קונפליקט בתחילת הענף כנראה שגם הקומיטים הבאים באותו ענף התבססו על הגירסא המתנגשת ולכן ההתנגשות תמשיך גם אליהם.

תקנו את הקובץ וסיימו את הריבייס. לאחר מכן השתמשו בפקודות הבאות כדי להחזיר את המצב לקדמותו. תחילה פתחו את ה reflog כדי למצוא את מזהה הקומיט העדכני ביותר ב dev לפני הריבייס:

$ git reflog

0ec3d91 (HEAD -> dev) HEAD@{0}: rebase finished: returning to refs/heads/dev

0ec3d91 (HEAD -> dev) HEAD@{1}: rebase: a third try from dev branch

257f504 (master) HEAD@{2}: rebase: checkout master

b4f0c86 HEAD@{3}: commit: a third try from dev branch

6934bf1 (tag: A) HEAD@{4}: checkout: moving from master to dev

257f504 (master) HEAD@{5}: commit: a second try

6934bf1 (tag: A) HEAD@{6}: commit (initial): initial commit

אצלי מזהה זה הוא b4f0c86.

עזבו את ענף dev ועברו ל master:

$ git checkout master

עכשיו השתמשו ב git branch כדי להחזיר את dev לנקודה בה היה לפני הריבייס. אצלי הפקודה היא:

$ git branch -f dev b4f0c86

התאימו את מזהה הקומיט למה שמצאתם דרך ה reflog.

נחזור לענף dev:

$ git checkout dev

לאחר מכן פתחו את הקובץ myfile.txt בעורך טקסט וכתבו בפנים את התוכן הבא:

ok ok last conflict

just kidding

I can has cheezburger

שמרו בקומיט חדש:

$ git commit -am 'final conflict'

שימו לב מה קורה כשאני מפעיל שוב rebase:

First, rewinding head to replay your work on top of it...

Applying: a third try from dev branch

Using index info to reconstruct a base tree...

M myfile.txt

Falling back to patching base and 3-way merge...

Auto-merging myfile.txt

CONFLICT (content): Merge conflict in myfile.txt

error: Failed to merge in the changes.

Patch failed at 0001 a third try from dev branch

The copy of the patch that failed is found in: .git/rebase-apply/patch

Resolve all conflicts manually, mark them as resolved with

"git add/rm <conflicted\_files>", then run "git rebase --continue".

You can instead skip this commit: run "git rebase --skip".

To abort and get back to the state before "git rebase", run "git rebase --abort".

זה הקונפליקט שכבר פגשנו קודם וציפינו לו. נתקן אותו בעזרת עורך טקסט או כלי merge ולאחר מכן נוסיף את הקובץ ונמשיך בריבייס:

$ git add myfile.txt

$ git rebase --continue

Applying: a third try from dev branch

Applying: final conflict

Using index info to reconstruct a base tree...

M myfile.txt

Falling back to patching base and 3-way merge...

Auto-merging myfile.txt

CONFLICT (content): Merge conflict in myfile.txt

error: Failed to merge in the changes.

Patch failed at 0002 final conflict

The copy of the patch that failed is found in: .git/rebase-apply/patch

Resolve all conflicts manually, mark them as resolved with

"git add/rm <conflicted\_files>", then run "git rebase --continue".

You can instead skip this commit: run "git rebase --skip".

To abort and get back to the state before "git rebase", run "git rebase --abort".

והנה הקונפליקט שכאילו פתרנו מופיע שוב בגלל הקומיט השני בענף. תוכן הקובץ myfile.txt הפעם הוא:

<<<<<<< HEAD

take two

=======

ok ok last conflict

>>>>>>> final conflict

just kidding

I can has cheezburger

בניגוד ל merge, פעולת rebase מפעילה את הטלאים אחד אחרי השני ולכן השורה **ok ok last conflict** כלל לא הופיעה בתיקון הטלאי הקודם.

דוגמא מהחיים

נבנה פרויקט קטן עם קובץ יחיד ובו באג. צרו פרויקט גיט חדש בתיקייה חדשה ובפרויקט כתבו קובץ יחיד בשם main.py עם התוכן הבא:

def fib(n):

x, y = 1, 1

for i in range(n):

x = y

y = x + y

return x

print(fib(8))

הרצה של התוכנית מדפיסה את הערך 128 שהוא כמובן לא המספר השמיני בסידרת פיבונאצ'י. נפתח ענף חדש כדי לחקור ולתקן את הבאג:

$ git checkout -b bugfix

לצורך התיקון נרצה להוסיף הודעות דיבג והדפסות. בואו נוסיף שתיים כאלה בשני קומיטים שונים. תחילה כתבו בתוכן הקובץ את הטקסט הבא:

def fib(n):

x, y = 1, 1

for i in range(n):

print(f"DEBUG x = {x}, y = {y}")

x = y

y = x + y

return x

print(fib(8))

ועכשיו קומיט כי עדיין לא מצאנו את הבעיה אבל צריך להמשיך לעבוד על משהו אחר:

$ git commit -am 'ADD debug messages'

ניסיון נוסף לחזור לקובץ והפעם נוריד קצת את המספר מ 8 ל 3. כתבו את התוכן הבא בקובץ:

def fib(n):

x, y = 1, 1

for i in range(n):

print(f"DEBUG x = {x}, y = {y}")

x = y

y = x + y

return x

print(fib(3))

ושוב קומיט כי צריך ללכת לעבוד על משהו אחר:

$ git commit -am 'simplified program so bug hunting is easier'

אחרי עוד שבועיים של מחשבה נופל האסימון ואתם מבינים שחישוב הערך הבא בסידרה היה לא נכון. התיקון פשוט ותוכן הקובץ נראה כעת כך:

def fib(n):

x, y = 1, 1

for i in range(n):

print(f"DEBUG x = {x}, y = {y}")

x, y = y, x + y

return x

print(fib(3))

וכמובן קומיט:

$ git commit -am 'found it'

עכשיו יש לנו לוג שנראה כך:

commit 087e9a624124d0878f1a23bf7be26c439e4e4b7a (HEAD -> bugfix)

Author: shay <shayd@gmail.com>

Date: Mon Aug 20 09:28:51 2022 +0300

found it

commit cab3fcbd2166a01f24f003b6f4e6d3c2eeb808e5

Author: shay <shayd@gmail.com>

Date: Mon Aug 20 09:28:34 2022 +0300

simplified the program so bug hunting is easier

commit affb0793f97d43c976eb41fc0e1c3798cbf22ed9

Author: shay <shayd@gmail.com>

Date: Mon Aug 20 09:28:13 2022 +0300

ADD debug messages

commit d1e1291dafa7573a30bd34a9141679964b9c6f00 (master)

Author: shay <shayd@gmail.com>

Date: Mon Aug 20 09:18:30 2022 +0300

initial commit

אם נחזור ל master לבצע merge נקבל גם את כל הקומיטים באמצע עם הודעות הדיבג. חוץ מזה שהקוד ב bugfix עדיין לא נקי ומלא בהודעות הדפסה ואולי דברים אחרים שקילקלתי במהלך העבודה.

2. פיתרון עם cherry pick

הפקודה git cherry-pick יודעת לקחת קומיט או רצף קומיטים ממקום אחר, לזהות איזה שינויים כל קומיט הכניס ולבצע שינויים אלה בענף הנוכחי. במובן זה היא מאוד דומה לריבייס רק קצת יותר מדויקת ומתחילה מהענף הראשי (אליו רוצים לכתוב את השינויים) במקום מהענף שבו ביצענו את השינויים.

נחזור ל master ונייצר קומיט חדש שיהיה מבוסס רק על השינוי שהכניס הקומיט האחרון מ bugfix:

$ git checkout master

$ git cherry-pick bugfix

error: could not apply 087e9a6... found it

hint: after resolving the conflicts, mark the corrected paths

hint: with 'git add <paths>' or 'git rm <paths>'

hint: and commit the result with 'git commit'

localhost:22-cherry-pick shay$ vi main.py

localhost:22-cherry-pick shay$ vi main.py

localhost:22-cherry-pick shay$

localhost:22-cherry-pick shay$ git status

On branch master

You are currently cherry-picking commit 087e9a6.

(fix conflicts and run "git cherry-pick --continue")

(use "git cherry-pick --abort" to cancel the cherry-pick operation)

Unmerged paths:

(use "git add <file>..." to mark resolution)

both modified: main.py

no changes added to commit (use "git add" and/or "git commit -a")

גיט ניסה להכניס את הקומיט הרלוונטי אבל נתקל בקונפליקט. כניסה לקובץ מראה לנו שהסיבה לקומיט היא שהשארנו חלקים בקובץ עם הודעות דיבג וזה התנגש עם ההתנהגות בענף הראשי:

def fib(n):

x, y = 1, 1

for i in range(n):

<<<<<<< HEAD

x = y

y = x + y

=======

print(f"DEBUG x = {x}, y = {y}")

x, y = y, x + y

>>>>>>> 087e9a6... found it

return x

print(fib(8))

השורות שבין המילה HEAD לשורת סימני השווה מראות את הגירסא ב master והשורות בין סימני השווה לחצים מראות את הגירסא ב bugfix. נאחד את השתיים כדי לקבל:

def fib(n):

x, y = 1, 1

for i in range(n):

x, y = y, x + y

return x

print(fib(8))

ועכשיו קומיט ולוג:

commit f3cd270abc95b285c63f8b18d8a810c7b8dcf6eb (HEAD -> master)

Author: shay <shayd@gmail.com>

Date: Mon Aug 20 09:28:51 2022 +0300

found and fixed fibonacci bug

commit d1e1291dafa7573a30bd34a9141679964b9c6f00

Author: shay <shayd@gmail.com>

Date: Mon Aug 20 09:18:30 2022 +0300

initial commit

הקומיטים האחרים בענף לא נכנסו ויש לנו קומיט אחד בלבד עם התיקון.

3. איך בכל זאת נראה merge במקרה כזה

ל merge יש שני מצבי פעולה שיכולים לעניין אותנו במקרה כזה וברוב המקרים יהיו טובים יותר מ cherry pick. נתחיל עם המיזוג הרגיל. באותו הפרויקט בואו נבטל את ה cherry pick שעשינו ונחזיר את master ו HEAD להתחלה עם:

$ git reset --hard d1e129

$ git log

commit d1e1291dafa7573a30bd34a9141679964b9c6f00 (HEAD -> master)

Author: shay <shayd@gmail.com>

Date: Mon Aug 20 09:18:30 2022 +0300

initial commit

הדרך הרגילה לשלב ענף תיקון לענף הראשי היא merge. נפעיל ונראה מה קורה:

$ git merge bugfix

$ git log

commit 087e9a624124d0878f1a23bf7be26c439e4e4b7a (HEAD -> master, bugfix)

Author: shay <shayd@gmail.com>

Date: Mon Aug 20 09:28:51 2022 +0300

found it

commit cab3fcbd2166a01f24f003b6f4e6d3c2eeb808e5

Author: shay <shayd@gmail.com>

Date: Mon Aug 20 09:28:34 2022 +0300

simplified the program so bug hunting is easier

commit affb0793f97d43c976eb41fc0e1c3798cbf22ed9

Author: shay <shayd@gmail.com>

Date: Mon Aug 20 09:28:13 2022 +0300

ADD debug messages

commit d1e1291dafa7573a30bd34a9141679964b9c6f00

Author: shay <shayd@gmail.com>

Date: Mon Aug 20 09:18:30 2022 +0300

initial commit

פעולת merge הכניסה את כל הקומיטים שבענף bugfix לענף המרכזי שלנו למרות שדי ברור שאנחנו לא צריכים אותם כאן. בנוסף נסתכל על הקובץ main.py:

def fib(n):

x, y = 1, 1

for i in range(n):

print(f"DEBUG x = {x}, y = {y}")

x, y = y, x + y

return x

print(fib(3))

כל הלכלוך שבקובץ שהוכנס משני הקומיטים באמצע נשאר. גיט לא יודע להבדיל בין השינוי המיותר שעשינו בשורה האחרונה לשינוי החשוב בשורת החישוב. אין כאן הודעה על קונפליקט כי מבחינת גיט השינוי בשורה האחרונה מ-8 ל-3 היה חלק מהתיקון, כלומר חלק מהענף שאנחנו עושים לו merge. ברוב המקרים זו ההתנהגות הרצויה, אבל בדוגמא כאן זו לא - בגלל שמי שיצר את הענף שכח לנקות את כל הדפסות הדיבג שהכניס.

במצבים שאנחנו רוצים כן לקבל merge של כל הענף אבל לא לראות את כל הקומיטים שלנו בעץ הראשי נוכל להיעזר במתג **--squash** של פעולת הקומיט. מתג זה מבצע merge רגיל אבל בלי לבצע קומיט בסופו. נחזור למצב לפני ה merge האחרון ונראה איך זה עובד:

$ git reset --hard d1e129

$ git merge --squash bugfix

Updating d1e1291..087e9a6

Fast-forward

Squash commit -- not updating HEAD

main.py | 6 +++---

1 file changed, 3 insertions(+), 3 deletions(-)

$ git log

commit d1e1291dafa7573a30bd34a9141679964b9c6f00 (HEAD -> master)

Author: shay <shayd@gmail.com>

Date: Mon Aug 20 09:18:30 2022 +0300

initial commit

$ git status

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

modified: main.py

הקובץ main.py עודכן בתיקיית העבודה. עדכון זה עדיין לא יצר קומיט ולכן הוא מופיע בסטטוס כ Changes to be committed. נראה מה תוכן הקובץ:

def fib(n):

x, y = 1, 1

for i in range(n):

print(f"DEBUG x = {x}, y = {y}")

x, y = y, x + y

return x

print(fib(3))

כל הודעות ההדפסה והשינויים של bugfix נכנסו לקובץ ללא קונפליקטים. גיט עדיין לא יודע שחלק מהשינויים חשובים וחלק לא ולכן מבחינתו כל שינוי שהוכנס בענף הוא חלק מהתיקון.

ביצוע merge עכשיו יכניס את התיקונים ויביא אותנו למצב של מיזוג הענף דרך קומיט יחיד:

$ git commit -am 'merged bugfix'

$ git log

commit cefb43ea21717108d3f31a13a21dcb1ef311c71d (HEAD -> master)

Author: shay <shayd@gmail.com>

Date: Mon Aug 20 09:52:41 2022 +0300

merged bugfix

commit d1e1291dafa7573a30bd34a9141679964b9c6f00

Author: shay <shayd@gmail.com>

Date: Mon Aug 20 09:18:30 2022 +0300

initial commit

זיכרו:

1. נשתמש ב cherry-pick כשיש רק קומיט אחד (או מספר קומיטים) חשובים בענף אחר ואותם אנחנו רוצים.
2. נשתמש ב merge כשאנחנו צריכים את כל הענף האחר כולל את המידע על כל הקומיטים שבוצעו בו.
3. נשתמש ב merge --squash כשאנחנו צריכים את כל השינויים מענף אחר אבל לא את פרטי הקומיטים שלהם.

בדוגמא שלנו אם המתכנת שבנה את ענף bugfix היה מנקה את הענף כשסיים ומחזיר את הקוד למצב שהיה בו לפני הוספת הודעות ההדפסה היה עדיף לבחור ב merge --squash.