

הכנת החבילה לפני פרסום

החלק הראשון בתהליך הפרסום, ובין המסובכים בו, הוא לתת שם (טוב) לספרייה. כל הספריות ב-pypi צריכות להיות עם שמות ייחודיים. עם כמעט רבע מיליון חבילות באתר סיכוי גדול שהשם של הספרייה שאנחנו רוצים כבר תפוס. טיפ: בשביל למצוא אם השם תפוס נוכל להשתמש בשורת החיפוש באתר.

השם של החבילה לא חייב להיות זהה לשם שאתו נייבא את הספרייה - נוכל לקרוא לספרייה `my_package`, וניתן לה שם ייחודי באתר, למשל `my_very_first_package` ואז שיורידו את החבילה מ-pypi יורידו אותה כך:

```
pip install my_very_first_package
```

אבל כשנייבא את הספרייה לפרויקט נייבא אותה כ-`my_package`:

```
import my_package
```

כמובן שעדיף לקרוא לחבילה באותו השם שאיתו אנחנו מייבאים אותה, אחרת סתם נבלבל את המשתמשים האחרים.

קונפיגורציה לחבילה:

בכדי שהספרייה תוכל לעלות ל-pypi, צריך לתת קצת אינפורמציה בסיסית עליה. את האינפורמציה אנחנו מספקים מקובץ שנקרא `setup.py`. הקובץ אמור להיות בתיקייה הראשית של הפרויקט. הקובץ אמור לייבא שני מודולים, הראשון מומלץ אבל אפשר להסתדר בלעדיו, והשני קשה מאוד להסתדר בלעדיו:

- `pathlib` שנועד לייבוא של קבצים מהאזור של התיקייה כמו קובץ `readme` או `requirement` וכדו'.
- מהספרייה `setuptools` צריך לייבא את הפונקציה `setup()`.

הספרייה `setuptools` היא ספרייה שמקלה על המתכנתים לבנות ולהפיץ ספריות פייתון, במיוחד כאלו שמסתמכות על חבילות אחרות. שתי הספריות נמצאות בספרייה הסטנדרטית של פייתון כך שאין צורך להתקין שום דבר חיצוני. הפונקציה `setup()` מחייבת כמה פרמטרים:

- שם (`name`): כאן אנחנו בעצם מגדירים איך תיקרא החבילה ב-pypi (מחרוזת).
- גרסה (`version`): הגרסה הנוכחית של החבילה, מגיע כמחרוזת.
- חבילות (`packages`): התיקיות ותתי התיקיות של הספרייה, הארגומנט אמור להישלח כרשימה של מחרוזות. אם יש לנו הרבה תיקיות אפשר להשתמש בפונקציה `setuptools.find_packages()` כדי לקבל את כל התיקיות בפרויקט. אפשר גם להורות לפונקציה אילו תיקיות לא נרצה לייבא עם פרמטר `exclude` שמקבל tuple או רשימה של שמות של תיקיות שלא נרצה שיילקחו בחשבון בחבילה, למשל טסטים (`tests`) יכול להיות שלא נרצה וכו'.

```
packages=find_packages(exclude=("tests",)),
```

אומנם רק שם, גרסה וחבילות הם פרמטרים שחייבים להוסיף, אבל אם נכניס עוד פרמטרים יהיה הרבה יותר קל למצוא את החבילה שלנו ב-pypi, לאמת שאכן זאת החבילה שלנו ונקל על מתכנתים אחרים בהתקנת החבילה. למשל ממומלץ להגדיר את הקובץ `readme` בפונקציה כדי שנוכל לראות אותו באתר של pypi. או להגדיר את שם המחבר של הספרייה ודרכי התקשרות.

כמו כן לפעמים הספרייה מצריכה כמה חבילות חיצוניות כמו `numpy` למשל, שלא מגיעות עם הספרייה הסטנדרטית, במקרה כזה נצטרך פרמטר `install_requires` שמכיל רשימה של ספריות שצריך להתקין כדי להשתמש בחבילה שלנו (בהמשך נראה כיצד להשתמש בקובץ `requirement.txt` שדיברנו עליו קודם לכן). עוד משהו מגניב שאפשר להגדיר הוא `entry_points`, `entry_points` הם פונקציות שאנחנו מגדירים שיופעלו מתוך שורת



ד"ר סגל הלוי דוד אראל

הפקודה בשימוש בפקודה מוגדרת מראש. למשל יש לנו בקובץ `__main__.py` פונקציה `main()` שנרצה שתופעל בשורת הפקודה כאשר מקלידים `my_package_main` נוכל להגדיר את זה ע"י הוספה של הפרמטר `entry_points` שמקבל מילון שהמפתח הוא איפה אמורה להיות מוגדרת הפקודה, והערך הוא רשימה של מחרוזות שמכילות את שם הפקודה שירשור עם '=' שירשור ה-`path` של קובץ, נקודותיים ושם הפונקציה. להמחשה:

```
setup(
    ...,
    entry_points={
        "console_scripts": ["my_package_main =my_package.__main__:main"],
    }
)
```

ולאחר שהמשתמש יוריד את הספרייה הוא יוכל פשוט לכתוב `my_package_main` בטרמינל או ב-`cmd` זה יריץ אוטומטית את הפונקציה `main()` של הקובץ `__main__.py` ב-`my_package`.

דוקומנטציה:

לפני שמפרסמים את החבילה יש צורך בהסבר על החבילה. בהתאם לגודל החבילה הדוקומנטציה יכולה להיות קטנה כקובץ `readme` פשוט, או גדולה כמו אתר אינטרנט, לרוב חבילות ה-`numpy stack` למשל יש אתר אינטרנט. לכל הפחות יש צורך בקובץ `readme` עם הפרויקט. קובץ `readme` טוב אמור לתאר את הפרויקט ולהסביר למשתמש איך להתקין אותו ולהשתמש בו. בדרך כלל נרצה להוסיף את קובץ ה-`readme` עם הפרמטר `long_description` של הפונקציה `setup()`. זה יציג את הקובץ באתר של `pypi`. קבצי `readme` אמורים להגיע בפורמט `markdown` (קבצים עם הסיומת '`.md`'), כמו קבצי ה-`readme` של גיטהאב, או בפורמט `reStructuredText` (סיומת '`.rst`'). אם צירפתם את קובץ ה-`readme` לפונקציה צריך להוסיף עוד פרמטר שנקרא `long_description_content_type` שמתאר את סוג הקובץ שצורף למשל עבור קובץ `markdown` צריך לפרמטר את הערך '`text/markdown`', ועבור קובץ `reStructuredText` צריך לתת לו את הערך '`text/x-rst`'.

כדי להוסיף קובץ מתוך הספרייה לפונקציה נוכל להשתמש במודול `pathlib` או בקריאת קובץ פשוטה:

```
with open("README.md", "r") as fh:
    README = fh.read()
```

או עם הספרייה `pathlib` כדי לא להשתמש ב-`context manager`:

```
# The directory containing this file
HERE = pathlib.Path(__file__).parent
# The text of the README file
README = (HERE / "README.md").read_text()
```

אם לסכם את מה שראינו קובץ `setup` פשוט אמור להיראות בערך כך:

```
import pathlib
```



ד"ר סגל הלוי דוד אראל

```

from setuptools import setup, find_packages

# The directory containing this file
HERE = pathlib.Path(__file__).parent

# The text of the README file
README = (HERE / "README.md").read_text()

# This call to setup() does all the work
setup(
    name="my_package",
    version="1.0.0",
    description="An example of a pypi package",
    long_description=README,
    long_description_content_type="text/markdown",
    url= '...', # link to your github
    author="...",
    author_email="...",
    packages=find_packages(exclude=("tests",)),
    include_package_data=True,
    entry_points={
        "console_scripts": ["my_package_main=my_package.__main__:main"],
    },
)

```

בנוגע ל-requirement : אם יש צורך להוסיף requirement לפונקציה setup() צריך להעביר אותו בצורה של רשימה של מחרוזת, וכל תא ברשימה יהיה שם של ספרייה עם התיאור שלה כפי שמופיע בקובץ requirement.txt (או כל קובץ שבו כתבתם את רשימת החבילות הנלוות לפרויקט). בשביל זה נצטרך לקרוא שורה אחר שורה מתוך הקובץ, ולהכניס אותה בנפרד כתא חדש לרשימה:

```

def parse_requirements_file(filename):
    with open(filename) as fid:
        requires = [lin.strip() for lin in fid.readlines()]
    return requires

REQUIRES = parse_requirements_file("requirement.txt")

```

אחר כך בפונקציה setup() נוסיף את המשתנה לפרמטר install_requires

```

setup(

```



ד"ר סגל הלוי דוד אראל

```
....
install_requires = REQUIRES
...
```

לתת גירסה לחבילה:

כפי שראינו כל חבילה חייבת להגיע עם גירסה, וניתן לעדכן חבילות רק פעם אחת עם אותה גירסה שלהן, כלומר כל פעם שנעדכן את החבילה נהיה חייבים גם לעדכן את גרסת החבילה. זה דווקא דבר טוב, זה מבטיח שיחזור של המערכת: שתי מערכות עם אותה הגירסה אמורות להתנהג אותו דבר. יש הרבה מאוד סכמות שיכולות לשמש כמספר הגירסה של החבילה. עבור פרויקטים בפייתון יש המלצה של הדקומנטציה של פייתון ([PEP 440](#)), אבל משום שהדוקומנטציה מאוד מסובכת נשאר עם סכמה פשוטה לניתנת גרסאות - [השיטה הסמנטית](#). הסכמה או השיטה הסמנטית היא ברירת מחדל טובה לשימוש. מספר הגירסה ניתן כשלושה רכיבים מספריים, למשל 0.1.2 הרכיבים נקראים עיקרי (MAJOR), משני (MINOR) ותיקונים (PATCH), ויש חוק פשוט מתי לעלות את ערכו של כל רכיב:

- את העיקרי נעלה כשאנחנו יוצרים שינוי מהותי של הספרייה, כלומר שהגירסה החדשה והישנה שונות לגמרי בדרך פעולתן, למשל שימוש בספריות שונות כדי לבצע את אותו תהליך - החלטנו שחשוב דרך פייתון הוא איטי מיד ועברנו לחישוב דרך cython או c++ וכו'.
- נעלה את המשני כאשר הוספנו פונקציונאליות חדשה שלא שינתה מהותית את הגירסה הישנה, למשל הוספנו עוד ספריות חדשות.
- את התיקונים נעלה כשתיקנו באגים שהתגלו בגירסה האחרונה (תקלות ברמת הרכיב המשני, תקלות שעלולות לגרום לשינוי כל הגירסה יחשבו ברכיב העיקרי).

יכול להיות שנצטרך לעדכן את הגירסה בכמה מקומות, למשל אם הגדרנו את גרסת הפרויקט גם בפונקציית `setup()` וגם בקובץ `__init__.py` (נהוג לציין את שם את הגירסה אם יש תיקייה ראשית לפרויקט). הערה: יש מוסכמה לכתוב את המשתנה שמציין את הגירסה של הפרויקט כ- `__version__`, והדבר נחוץ בעיקר עבור המודול הבא שנראה:

כדי לוודא שהגרסה תשאר עקבית נוכל להשתמש במודול [bumpversion](#).

בשביל להשתמש בו נצטרך להתקין אותו:

```
pip install bumpversion
```

ואז אם נרצה לעלות אחד הרכיבים בעדכון הגירסה עם `bumpversion` נצטרך לציין בטרמינל או ב-cmd מה הגירסה הנוכחית של הפרויקט, איזה רכיב נרצה לשנות והיכן נמצא המשתנה שמורה על הגירסה:

```
$ bumpversion --current-version 1.0.0 minor setup.py my_package/__init__.py
```

זה יעלה את גרסת החבילה מ-1.0.0 ל-1.1.0.

הוספת קבצים לחבילה-

לפעמים יש חבילות שמכילות קבצים שאינם קבצי קוד מקור. לדוגמה קבצי דאטה, קבצים בינאריים, דוקומנטציות, קבצי קונפיגורציה וכו'.

כדי להגדיר לפונקציה `setup()` להכליל את הקבצים האלו, נצטרך להשתמש בקובץ שנקרא קובץ מניפסט.

עבור רוב הפרויקטים אין צורך לדאוג מקובץ הזה, היות ו-`setup()` יוצר אחד שמכיל בתוכו את כל קבצי הקוד וקבצי ה-`README`.

אבל אם נרצה לשנות את קובץ המניפסט, נצטרך ליצור תבנית של קובץ מניפסט (`manifest template`) שחייב להיקרא `'MANIFEST.in'`. הקובץ אמור להגדיר חוקים מה יכלול בחבילה, ומה לא:



ד"ר סגל הלוי דוד אראל

```
include my_package/*.txt
exclude tests/test.py
```

בדוגמא לעיל הורנו לקבל את כל קבצי הטקסט בתיקייה my_package, ולדחות את הקובץ test.py מהתיקייה test. אין גם מה לדאוג מכתיבת הקובץ היות ואין יותר מידי פקודות שאפשר לעשות:

מה היא עושה	דוגמא	הפקודה
מכלילה את כל הקבצים שבאים אחרי הפקודה. (ספציפי יותר)	include *.rst README.md	Include pat1 pat2 ...
מתעלם מהקבצים שבאים לאחר הפקודה (ספציפי יותר)	exclude *.cnf no_to_use.md	exclude pat1 pat2 ...
מכליל את כל הקבצים המוגדרים שנמצאים התיקייה ספציפית	recursive-include my_package *.txt	recursive-include dir pat1 ...
מתעלם מכל הקבצים המוגדרים שנמצאים בתיקייה ספציפית	recursive-exclude my_package *.md	recursive-exclude dir pat1 ...
מכליל את כל הקבצים שנמצאים בעץ מהסוג שמוגדר לאחר הפקודה	global-include *.txt	global-include pat1 pat2 ...
מתעלם מכל הקבצים שנמצאים בעץ מהסוג שמוגדר לאחר הפקודה	global-exclude *.cnf *.log	global-exclude pat1 pat2 ...
מתעלם מכל הקבצים שבתקייה מסוימת לא משנה מה הפורמט שלהם	prune tests	prune dir
מכליל את כל הקבצים בתיקייה מסוימת לא משתנה מה הפורמט שלהם	graft my_package	graft dit

בנוסף ליצירת המניפסט צריך להגדיר לפונקציה setup() להעתיק את הקבצים שאינם קוד מקור (non-code file) את זה נעשה ע"י הוספת הפרמטר include_package_data וניתן לו את הערך הבוליאני True:

```
setup(
    ...,
    include_package_data=True
    ...
```

הפרמטר מגדיר אם קבצים שאינם קוד יועתקו כאשר מתקינים את החבילה.

