## שלום קארגו!

קארגו (Cargo) הוא מערכת הבניה ומנהל החבילות של ראסט. מכיוון שקארגו מטפל בהרבה מטלות עבורכם, רוב הראסטיונרים (Rustaceans) משתמשים בכלי זה על מנת לנהל את פרוייקטי הראסט שלהם. פעולות ניהול אלו כוללות למשל את בניית הקוד, הורדת הספריות עליהן נשען הקוד שכתבתם, ובנייתן ושילובן בתוך התכנית שלכם (ספריות חיצוניות מעין אלה, שהקוד שלכם צריך כחלק מפעולת התוכנית, נקראות \*תלותות\*.)

לתוכנות ראסט פשוטות, כמו אלו שכתבנו עד עכשיו, אין תלותות. לו היינו בונים את פרוייקט ה-"!Hello, world" עם קארגו, אז היה נעשה שימוש רק באותו חלק של קארגו המטפל בבניית הקוד. ככל שתכתבו תוכנות ראסט מורכבות יותר, סביר שתוסיפו תלותות רבות יותר; ואם מראש תתחילו פרוייקט דרך קארגו, הוספת תלותות תהפוך קלה ופשוטה לביצוע.

כיוון שהרוב העצום של פרוייקטי ראסט משתשמשים בקארגו, שאר הספר מניח שגם אתם משתמשים בקארגו. קארגו מותקנת כחלק מראסט, במידה והשתמשתם בהתקנה הרשמית כפי שהסברנו בסעיף ["התקנה"][installation]<!-- ignore -->. אם התקנתם את ראסט בדרך אחרת, בדקו אם קארגו מותקן ע"י הקלדת הפקודה הבאה בטרמינל:

```console

$ cargo --version

```

אם מספר גרסה כלשהו מופיע על המסך, אז קארגו מותקן! אם אתם מקבלים שגיאה, כמו `command

not found`, קראו שוב את התיעוד עבור אופן ההתקנה שבחרתם בכדי להבין איך להתקין את קארגו בנפרד.

### יצירת פרוייקט ראסט חדש עם קארגו

הבה ניצור פרוייקט חדש באמצעות קארגו ונראה כיצד זה שונה מפרוייקט ה-"!Hello, world" המקורי שלנו. נווטו חזרה אל תיקיית \*הפרוייקטים\* שלכם (קרי, לתיקייה בה נמצא הקוד שלכם). עכשיו, לא משנה באיזו מערכת הפעלה אתם משתמשים, הריצו את הפקודות:

```console

$ cargo new hello\_cargo

$ cd hello\_cargo

```

הפקודה הראשונה יוצרת תיקייה חדשה ופרוייקט בשם \*hello\_cargo\*. לפרוייקט שלנו קראנו \*hello\_cargo\*, וקארגו מייצר את הקבצים שלו בתיקייה בעלת אותו שם.

היכנסו לתיקייה \*hello\_cargo\* והתבוננו את הקבצים ששם. תראו שקארגו יצר שני קבצים ותיקייה נוספת עבורנו: הקובץ \*Cargo.toml\* והתיקייה \*src\*, ובתוכה הקובץ \*main.rs\*.

בנוסף, בוצע אתחול של מאגר Git ביחד עם קובץ \*gitignore.\* מתאים. קבצי Git לא ייווצרו במידה ותריצו `cargo new` בתוך מאגר Git קיים; ניתן לעקוף ברירת מחדל זו ע"י הרצת `cargo new --vcs=git`.

> הערה: Git הינו מערכת בקרת גרסאות נפוצה. ניתן לשנות את פקודת ה-`cargo new` בכדי להשתמש במערכת בקרת גרסאות אחרת, או לוותר על מערכת כזו, ע"י שימוש בדגל `--vcs`. הריצו `cargo new --help` כדי לראות את כל האפשרויות.

פתחו את \*Cargo.toml\* בעורך הטקסט החביב עליכם. תראו משהו דומה לכתוב ברשימה 1-2.

<span class="filename">Filename: Cargo.toml</span>

```toml

[package]

name = "hello\_cargo"

version = "0.1.0"

edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]

```

<span class="caption">רשימה 1-2: תוכן הקובץ \*Cargo.toml\* שנוצר בעקבות הרצת `cargo new`</span>

הפורמט של קובץ זה הוא [\*TOML\*][toml]<!-- ignore --> (\*Tom's Obvious Minimal Language\*), שהוא פורמט קביעת התצורה של קארגו.

השורה הראשונה, `[package]`, היא כותרת סעיף המציינת שההוראות הבאות קובעות את תצורתה של חבילה מסוימת. סעיפים נוספים יתווספו ככל שנוסיף מידע לקובץ.

שלושת השורות הבאות מספקות לקארגו מידע הנחוץ לו בכדי לקמפל את התוכנית: השם, הגרסה, והמהדורה הרצויה של ראסט. נדבר על מילת המפתח `edition` [בנספח ה'][appendix-e]<!-- ignore -->.

השורה האחרונה, `[dependencies]`, היא תחילת הסעיף המיועד להזנת תלותות הפרוייקט שלכם. בראסט, חבילות קוד נקראות \*מכולות\* (crates). לא נזדקק למכולות נוספות בפרוייקט זה, אבל בפרק 2 תהיה לנו הזדמנות לעשות זאת, ואז נחזור לסעיף התלותות.

כעת, פתחו את הקובץ \*src/main.rs\*:

<span class="filename">Filename: src/main.rs</span>

```rust

fn main() {

println!("Hello, world!");

}

```

קארגו יצר עבורכם תוכנית "!Hello, world", בדיוק כמו שכתבתם ברשימה 1-1! עד כאן, ההבדלים בין הפרוייקט שלנו והפרוייקט שקארגו יצר הם שקארגו מיקם את הקוד בתיקייה \*src\* ולנו יש את קובץ התצורה \*Cargo.toml\* בתיקייה הראשית.

קארגו מצפה שקבצי הקוד שלכם יימצאו בתיקייה \*src\*. התיקייה המרכזית של הפרוייקט היא רק עבור קבצי README, רשיונות, קבצי תצורה, וכל דבר שאינו חלק מהקוד עצמו. שימוש בקארגו מסייע בארגון פרוייקטים - יש מקום לכל דבר, וכל דבר נמצא במקומו.

אם התחלתם פרוייקט שלא באמצעות קארגו, כפי שעשינו בפרוייקט "!Hello, world", תמיד תוכלו להמיר אותו לפרוייקט המנוהל ע"י קארגו. פשוט העבירו את קוד הפרוייקט לתוך התיקייה \*src\* וצרו קובץ \*Cargo.toml\* מתאים.

### בניה והרצה של פרוייקט קארגו

הבה נראה מה משתנה כאשר בונים ומריצים את תכנית ה-"!Hello, world" דרך קארגו! מתוך התיקייה \*hello\_cargo\*, בנו את הפרוייקט ע"י הקלדת הפקודה:

```console

$ cargo build

Compiling hello\_cargo v0.1.0 (file:///projects/hello\_cargo)

Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs

```

פקודה זו יוצרת קובץ הרצה בתיקייה \*target/debug/hello\_cargo\* (או ב-\*target\debug\hello\_cargo.exe\* על Windows) במקום בתיקיה הנוכחית. בגלל שברירת המחדל היא בניה במצב debug, קארגו ממקם את הקבצים הבינארים בתיקייה בשם \*debug\*. את קובץ ההרצה תוכלו להריץ עם הפקודה:

```console

$ ./target/debug/hello\_cargo # or .\target\debug\hello\_cargo.exe on Windows

Hello, world!

```

אם הכל כשורה, תודפס על המסך השורה `Hello, world!`. הרצת `cargo build` בפעם הראשונה גם גורמת לקארגו ליצור קובץ חדש בתיקייה המרכזית: \*Cargo.lock\*. קובץ זה עוקב אחר הגרסאות המדוייקות של התלותות בפרוייקט. בפרוייקט זה אין תלותות, ולכן קובץ זה דל למדי. אין צורך לשנות קובץ זה ידנית; קארגו מנהל את תוכנו עבורכם.

אם כן, בנינו פרוייקט תוך שימוש ב-`cargo build` והרצנו אותו עם `./target/debug/hello\_cargo`, אבל ניתן גם להשתמש ב-`cargo run` כדי לקמפל את הקוד ולהריץ את קובץ ההרצה בפקודה אחת:

```console

$ cargo run

Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs

Running `target/debug/hello\_cargo`

Hello, world!

```

נוח להשתמש ב-`cargo run` כי אז אין צורך לזכור להריץ `cargo build` ואח"כ להשתמש במסלול המלא אל הקובץ הבינארי, ולכן רוב המפתחים משתמשים ב-`cargo run`.

שימו לב שהפעם לא התקבלו הודעות בעת הקימפול. קארגו הבין שהקבצים לא עברו שינוי, ולכן הוא אינו נדרש לבנות שוב את הקובץ, אלא רק להריץ את הקובץ הקיים. לעומת זאת, אם הייתם משנים את קוד המקור, קארגו היה בונה מחדש את הפרוייקט לפני ההרצה, והייתם רואים את הפלט:

```console

$ cargo run

Compiling hello\_cargo v0.1.0 (file:///projects/hello\_cargo)

Finished dev [unoptimized + debuginfo] target(s) in 0.33 secs

Running `target/debug/hello\_cargo`

Hello, world!

```

קארגו כולל בנוסף את הפקודה `cargo check`. פקודה זו מבצעת בדיקה מהירה של הקוד ומוודאת שהוא אכן עובר קומפילציה, אך אינה בונה קובץ הרצה:

```console

$ cargo check

Checking hello\_cargo v0.1.0 (file:///projects/hello\_cargo)

Finished dev [unoptimized + debuginfo] target(s) in 0.32 secs

```

מדוע שלא תרצו לייצר קובץ הרצה? הפקודה `cargo check` לרוב מהירה הרבה יותר ביחס ל-,`cargo build` בדיוק משום שהיא אינה כוללת את שלב ייצור קובץ ההרצה. אם אתם בודקים את עבודתכם באופן תדיר בזמן כתיבת הקוד, שימוש ב-`cargo check` יאפשר לכם לוודא במהירות שהקוד עובר קומפילציה! לכן, ראסטיונרים רבים מריצים `cargo check` לעיתים קרובות, תוך כדי כתיבת התוכנית. את הפקודה `cargo build` הם מריצים רק כאשר הם מוכנים להשתמש בקובץ ההרצה.

הבה נסכם את שלמדנו עד כה על קארגו:

\* ניתן ליצור פרוייקט באמצעות `cargo new`.

\* ניתן לבנות פרוייקט באמצעות `cargo build`.

\* ניתן לבנות ולהריץ פרוייקט בצעד אחד באמצעות `cargo run`.

\* ניתן לבנות פרוייקט ללא בניית קובץ בינארי על מנת לבדוק שגיאות באמצעות `cargo check`.

\* במקום לשמור את תוצאת הבניה באותה תיקייה בה מאוחסן הקוד, קארגו שומר אותה בתיקייה \*target/debug\*.

יתרון נוסף לשימוש בקארגו הוא ששמות הפקודות אינן תלויות במערכת ההפעלה שלכם. לכן, מכאן ואילך, נפסיק לספק הוראות ספציפיות ל-Linux ו-macOS לעומת Windows.

### בניה להפצה

בשעה טובה, כאשר הפרוייקט שלכם מוכן להפצה, ניתן להשתמש ב-`cargo build --release` כדי לבצע קומפילציה עם אופטימיזציות. פקודה זו תיצור קובץ הרצה בתיקייה \*target/release\* במקום ב-\*target/debug\*. הקוד המופק תוך שימוש באופטימיזציות רץ מהר יותר, אם כי תהליך הקומפילציה דורש זמן רב יותר. משום כך קיימות שתי דרכי פעולה שונות: הראשונה לפיתוח, כאשר מעוניינים לבנות מהר ולעיתים תכופות; והשניה, כאשר מעוניינים לבנות תכנית סופית המיועדת עבור המשתמש, שלא תיבנה שוב ושוב, ושתרוץ הכי מהר שאפשר. אם אתם מודדים את מהירות ריצת הקוד שלכם וודאו להשתמש ב-`cargo build --release` ולמדוד את זמן הריצה של קובץ הריצה הנמצא ב-\*target/release\*.

### קארגו כמוסכמה

קארגו אינו מספק ערך מוסף רב מעבר לשימוש ב-`rustc`, עבור פרוייקטים פשוטים. אבל ככל שהתכניות שלכם יהפכו מורכבות יותר, כך יגדל ערכו של קארגו. ברגע שתכנות מערבת מספר קבצים או תלותות, הרבה יותר נוח לתת לקארגו לנהל את תהליך הבניה עבורכם.

למרות הפשטות של הפרוייקט `hello\_cargo`, כבר עושה שימוש ברוב הכלים האמיתיים בהם תיעזרו לאורך קריירת הראסט שלכם. למעשה, בכדי לעבוד על כל פרוייקט קיים שהוא, ניתן תמיד לעשות כך בכדי להשיג את הקוד באמצעות Git, להיכנס לתיקיית הפרוייקט, ולבנות:

```console

$ git clone example.org/someproject

$ cd someproject

$ cargo build

```

למידע נוסף אודות קארגו, פנו [לתיעוד][cargo] שלו.

## סיכום

אתם כבר בדרך הנכונה במסע לעבר שליטה בשפת ראסט! בפרק זה למדתם כיצד:

\* להתקין את הגרסה היציבה החדשה ביותר של ראסט באמצעות `rustup`

\* לעדכן לגרסה חדשה של ראסט

\* לפתוח תיעוד המותקן מקומית

\* לכתוב ולהריץ תוכנית "!Hello, world" באמצעות `rustc` ישירות

\* ליצור ולהריץ פרוייקט חדש באמצעות המוסכמות של קארגו

על מנת להתרגל לקרוא ולכתוב קוד ראסט, נתחיל כעת לבנות תכנית מעט יותר משמעותית. על כן, בפרק 2, נבנה תכנית המיישמת משחק ניחוש מספר. אם אתם מעדיפים לפני כן להבין כיצד עקרונות תכנות נפוצים מיושמים בראסט, קפצו לפרק 3 וחזרו אח"כ לפרק 2. הבחירה בידכם.

[installation]: ch01-01-installation.html#installation

[toml]: https://toml.io

[appendix-e]: appendix-05-editions.html

[cargo]: https://doc.rust-lang.org/cargo/