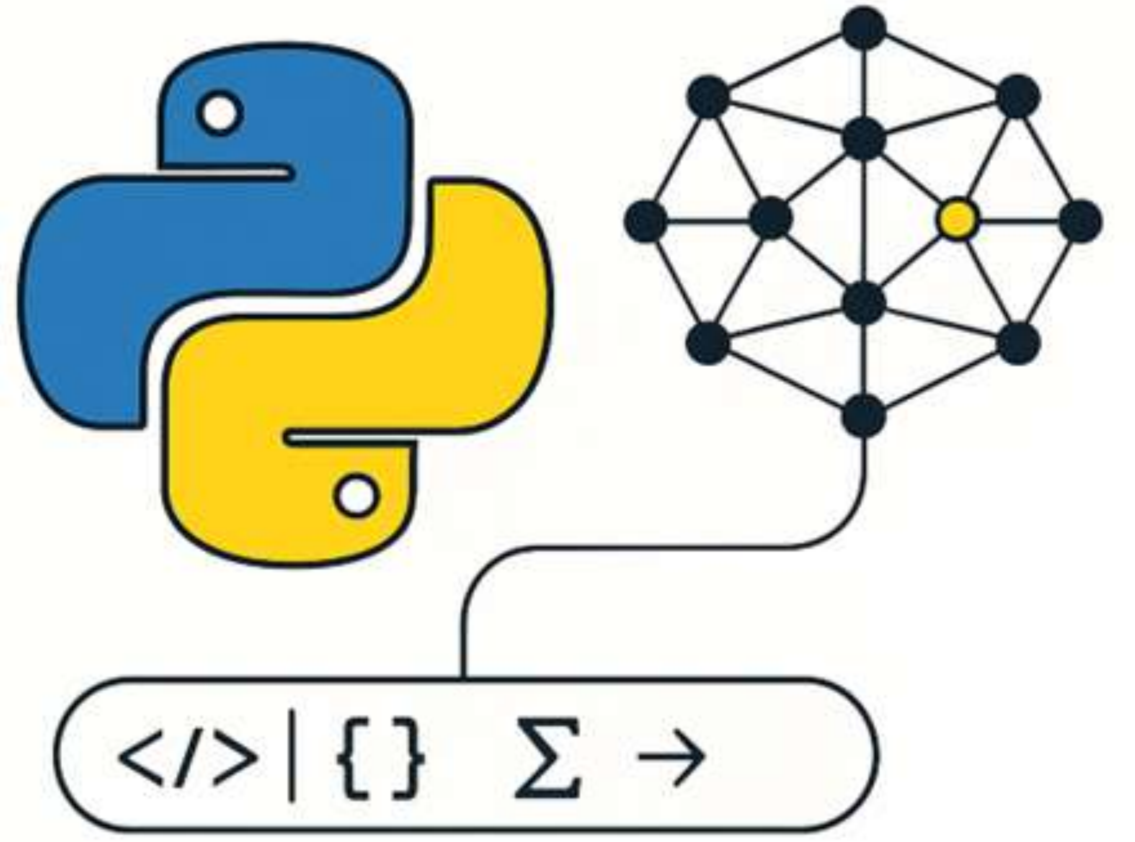
# פייתון פרקטי למתכנתים לעידן ה-AI




מאת: **תומר קדם**

**יוצר וארכיטקט התוכן של**
**AI Developer World Class Series**

גרסה 2025 | 4.6

# זכויות יוצרים

## ספרון 1: פייתון פרקטי למתכנתים לעידן ה-AI

**© 2025 תומר קדם** כל הזכויות שמורות.

ספרון זה מופץ ללא עלות במסגרת הסדרה
**AI Developer World-Class Series**
ומזמין אותך לבקר באתר הרשמי של הסדרה:

https://tomkedem.github.io/AI-Developer-World-Class-Series/

ההפצה מותרת אך ורק באמצעות שיתוף הקישור הרשמי לספרון או לקישור הראשי של הסדרה.
**אין לערוך, למכור או להשתמש בתוכן למטרות מסחריות** ללא אישור בכתב מן המחבר.

התוכן נוצר ונסקר בסיוע כלים מבוססי בינה מלאכותית,
בהנחיית המחבר **תומר קדם**.

שמות הספריות והטכנולוגיות המוזכרים בספרון
(כגון **NumPy**, **Python**, ו-**Pandas**)
הם סימנים מסחריים של בעליהם החוקיים,
והשימוש בהם נעשה לצורכי לימוד והסברה בלבד.

# תוכן העניינים

**מבוא ........................................................................ 1**

**פרק 1 - למה פייתון היא שפה חשובה בעידן ה-AI ........................ 3**

פייתון ככלי הנדסי (לא רק סקריפטים) ............................................ 4

איך פייתון מריצה AI מאחורי הקלעים .............................................. 6

כללי סגנון PEP 8 – וקריאות קוד ..................................................... 8

עבודה נקייה עם קוד: Separation of Concerns ............................ 10

דוגמה מרכזית: סקריפט שמקבל טקסט ומחזיר JSON ..................... 13

סיכום: איך פייתון משרתת מהנדסיAI ............................................ 16

**פרק 2 - יסודות פייתון למתכנתים מנוסים תחביר בסיסי ............... 17**

משתנים וטיפוסים מובנים (int, str, bool, None)............................ 17

תנאים (if / elif / else) והיגיון בוליאני............................................ 19

לולאות (for ו-while) ........................................................................ 23

חיתוכים ופעולות על מחרוזות(Slicing) ........................................... 27

f-strings לפורמט טקסט מודרני.......................................................... 31

List Comprehensions – כתיבה קומפקטית.................................... 35

דוגמה מרכזית: ניקוי רשימת משפטים עם תנאים ........................... 39

סיכום: איך לחשוב "פייתונית" ................................................... 42

**פרק 3 – מבני נתונים שימושיים ......................................... 44**

למה מבני נתונים קריטיים ב-AI........................................................ 44

list רשימות דינמיות ופעולות נפוצות........................................... 47

dict מילונים (key → value) ושימושים....................................... 50

set קבוצות של ערכים ייחודיים ................................................... 56

tuple רצף קבוע ואי-שינוי............................................................. 59

collections: defaultdict, Counter, deque ................................ 64

דוגמה מרכזית: סטטיסטיקות טקסט עם dict ו-Counter................... 68

סיכום: מתי להשתמש בכל מבנה נתונים.......................................... 71

**פרק 4 – פונקציות בפייתון.................................................... 74**

פונקציות כיחידת בניין קריטית ..................................................... 74

פרמטרים והחזרת ערכים (כולל tuple unpacking) ........................ 76

ערכי ברירת מחדל ו-pitfalls עם mutable defaults....................... 79

args* ו -kwargs**  מתי ואיך ....................................................... 82

lambda פונקציות אנונימיות ושימוש עםmap/filter ........................ 91
Scope: LEGB (Local, Enclosing, Global, Built-in) ..................... 94
Docstrings תיעוד קוד מקצועי ........................................................ 99
Best Practices שמות, פיצול, אחריות יחידה ................................ 103
דוגמה מרכזית: utility functions לעיבוד טקסט............................ 106
סיכום: מה מתכנת מנוסה צריך לזכור ........................................... 110

**פרק 5 – מודולים, חבילות וארגון פרויקט ............................ 113**

למה לא קובץ אחד גדול ................................................................ 113
Imports בסיסיים(import, from, alias) ...................................... 114
מבנה תיקיות מומלץ לפרויקטProduction .................................. 116
Imports יחסיים מול מוחלטים........................................................ 119
Docstrings למודולים: תיעוד ברמת הקובץ .................................. 120
Best Practices: שמות, אחריות וסדר imports ............................ 123
"__name__ == "__main__: הפרדה בין מודול להרצה ............... 125
Utility modules: איחוד פונקציות עזר.......................................... 127
דוגמה מרכזית: פרויקט mini_text_analyzer מחולק למודולים...... 129

סיכום: איך ארגון נכון מקל על הרחבה ........................................ 132

**פרק 6 – סביבות עבודה ותלויות ................................ 134**

למה סביבה וירטואלית חיונית ................................................ 134

venv: יצירה והפעלה (Windows / Linux / Mac) .......................... 135

pip: התקנת חבילות ו-requirements.txt .................................. 136

Poetry: הכלי המודרני לניהול תלויות ו-סביבות ............................ 138

pyproject.toml מול requirements.txt: מתי לבחור במה ........... 142

דוגמה מרכזית: סביבת פיתוח ל-mini_text_analyzer .................. 148

Best Practices: סביבה לכל פרויקט ו-gitignore ........................... 150

סיכום: איזה כלי מתאים לפרויקטיAI ........................................ 155

**פרק 7 – קבצים, נתיבים וקונפיגורציה ............................ 157**

למה עבודה עם קבצים קריטית ב-AI .......................................... 157

pathlib: הדרך המודרנית לעבוד עם נתיבים ................................. 158

קריאה וכתיבה של טקסט (UTF-8 תמיד) ...................................... 160

עבודה עם JSON .................................................................. 161

עבודה עם CSV .................................................................... 162

קונפיגורציה חיצונית (JSON/YAML) ........................................ 163

os.environ – משתני סביבה ו-dotenv........................................ 165

דוגמה מרכזית: קריאת Dataset, ניקוי ושמירה ........................... 166

Best Practices ........................................................................ 168

סיכום – למה קונפיגורציה נכונה חוסכת כאב ראש.......................... 169

**פרק 8 – חריגות, לוגים ואבחון ........................................ 170**

למה טיפול בשגיאות הוא קריטי ב-AI ........................................ 170

try / except / else / finally – המבנה הבסיסי ............................ 170

יצירת חריגות מותאמות(Custom Exceptions) ............................ 171

logging בסיסי – רמות INFO/WARNING/ERROR ........................ 172

extra dict – Structured Logging ו-correlation ID.................... 174

דוגמה מרכזית: עטיפת pipeline עם לוגים וחריגות......................... 175

סיכום – לוגים טובים הם העיניים של המערכת ............................ 178

**פרק 9 – תכנות מונחה עצמים בסיסי ................................ 179**

למה OOP חשוב גם כשיש functions ........................................ 179

הגדרת מחלקה בסיסית (init, self)........................................ 180

מתודות מיוחדות(str, repr, len) ........................................ 181

מתודות של מופע – גישה למשתני מחלקה ........................... 182

@classmethod ו-@staticmethod ........................................ 183

@dataclass – פחות boilerplate, יותר קריאות .......................... 189

קומפוזיציה מול ירושה – מתי מה.................................................. 195

דוגמה מרכזית: מחלקת TextCleaner עם API נקי ........................ 199

סיכום – OOP בזהירות, לא הכל צריך להיות מחלקה ...................... 199

**פרק 10 – טיפוסיות סטטית עםtyping ................................ 200**

למה type hints בשפה דינמית ................................................... 200

type hints בסיסיים (int, str, list, None)...................................... 201

טיפוסים מורכביםUnion, Optional, Literal ................................ 202

עבודה עם מבני נתונים (List[str], Dict[str, int])........................... 204

TypedDict – מבנה דמוי אובייקט מוגדר ......................................... 205

Protocols ו-Duck Typing – ממשקים ללא ירושה........................ 206

דוגמה מרכזית:.................................................................... 210

type hints מלאים ל-mini_text_analyzer ................................ 210

Best Practices ........................................................ 211

סיכום – איך type hints מגדילים אמינות .................................... 212

**פרק 11 – דקורטורים, Context Managers ו-dataclass .......... 213**

כלים שמקצרים ומפשטים קוד ............................................... 213

דקורטורים – פונקציה עוטפת פונקציה (@decorator) ................... 214

דקורטורים עם פרמטרים ((@repeat(n=3) .................................. 217

שימושים נפוצים(@lru_cache, @measure_time) .................... 218

Context Managers – with ו-enter/exit ................................... 219

contextlib – הפשטה עם @contextmanager............................ 220

@dataclass – קיצור למחלקות נתונים......................................... 220

דוגמה מרכזית: measure_time + @dataclass Result................ 221

Best Practices ........................................................................ 223

סיכום – איך הכלים האלה הופכים קוד לפייתוני .............................. 223

pytest: התקנה, מבנה קבצים, assert בסיסי ................................ 224

**פרק 12 – בדיקות אוטומטיות וארגונומיה למפתח ..................... 226**

למה בדיקות קריטיות בפרויקטיAI ............................................ 226

Fixtures: הכנה משותפת לבדיקות....................................227

בדיקות חריגות(pytest.raises) ................................................227

Mocking – סימולציה של API חיצוני...........................................228

כלים משלימים(black, ruff, pre-commit) ..................................229

דוגמה מרכזית: בדיקות ל-tokenize ו-clean..................................230

שילוב ב-CI/CD (GitHub Actions)..............................................231

Best Practices ..................................................................232

סיכום – בדיקות הן חלק מהפיתוח, לא "עוד משימה"........................232

**פרק 13 – ביצועים, זיכרון וקצתNumPy ................................233**

למה ביצועים חשובים גם בפייתון.................................................233

מדידת זמן ריצה – כי ניחושים לא משפרים ביצועים..........................234

עבודה חכמה עם זיכרון – לא כל דבר צריך רשימה ..........................236

NumPy בפועל – מערכים ופעולות וקטוריות חכמות ........................238

דוגמה מרכזית: תדירויות מילים (נאיבי מול NumPy) .........................241

Best Practices ..................................................................242

סיכום – איך להגיע לביצועים גבוהים בפייתון...................................242

**פרק 14 – Pandas למהנדסי AI ........................................ 243**

למה Pandas (ולא "רק" NumPy או dict) ........................................ 243

Series ו-DataFrame: הבסיס ........................................ 247

קריאה וכתיבה: CSV, JSON, Excel, Parquet ........................................ 249

בחירה וסינון: loc, iloc, Boolean Indexing ........................................ 254

טרנספורמציות: apply, map, groupby ........................................ 256

טיפול בנתונים חסרים: NaN, fillna, dropna ........................................ 259

מיזוג datasets: merge, concat, join ........................................ 262

דוגמה מרכזית: עיבוד dataset של טקסטים בעברית ........................................ 265

ביצועים וסקיילינג: מתי Pandas לא מספיקה ........................................ 268

סיכום: Pandas כשכבת הנתונים במערכת AI ........................................ 271

**פרק 15 – אסינכרוניות בסיסית וממשקי רשת ........................................ 274**

למה async חשוב בפרויקטי AI ........................................ 274

async/await: הבסיס ........................................ 275

לולאת האירועים: asyncio.run ו-gather ........................................ 276

aiohttp: קריאות רשת אסינכרוניות ........................................ 278

Timeouts ו-Retry: הגנה על קריאות API ........................... 279

Cancellation: ביטול משימות אסינכרוניות ................................ 281

דוגמה מרכזית: שליחת בקשות רבות ל API במקביל ....................... 284

Best Practices: אסינכרוניות ל-I/O בלבד וטיפול חכם בשגיאות ..... 289

סיכום: למה async ו-aiohttp הם חובה בפרויקטי AI ....................... 291

**פרק 16 – ממשק שורת פקודה(CLI) ....................................... 293**

למה CLI חשוב בפרויקטי AI.................................................... 293

argparse – הכלי המובנה....................................................... 295

Typer – הכלי המודרני........................................................... 298

Subcommands – פקודות משנה................................................ 303

קודי יציאה (Exit Codes) – 0 מול שאר......................................... 304

תיעוד אוטומטי (--help) ......................................................... 306

דוגמה מרכזית – CLI מלא למיני-טקסט (mintx)............................... 307

Best Practices – שמות ברורים ו-Defaults הגיוניים ....................... 310

סיכום – איך להפוך קוד לכלי שימושי ........................................... 312

**פרק 17 – תרגול מאוחד: בניית mini_text_analyzer............... 314**

תרגיל 1 – מבנה פרויקט ראשוני ........ 314
תרגיל 2 – טוקניזציה וניקוי בסיסי (פרקים 2–4) ........ 315
תרגיל 3 – מודולים וארגון (פרק 5) ........ 315
תרגיל 4 – קונפיגורציה חיצונית (פרק 7) ........ 316
תרגיל 5 – חריגות ולוגים מובנים (פרק 8) ........ 316
תרגיל 6 – OOP: מחלקת TextPipeline (פרק 9) ........ 317
תרגיל 7 Type hints – מלאים (פרק 10) ........ 317
תרגיל 8 – דקורטורים למדידה (פרק 11) ........ 317
תרגיל 9 – בדיקות אוטומטיות (פרק 12) ........ 318
תרגיל 10 – אופטימיזציה עם Pandas (פרקים 13–14) ........ 318
תרגיל 11 – CLI עם Typer (פרקים 15–16) ........ 318
תרגיל 12 – הרכבה סופית + CI/CD ........ 319
פתרון מלא ........ 319

**פרק 18 – תבנית פרויקט Production ל-AI Engineer ........ 320**

למה להתחיל עם שלד מסודר ........ 320
מבנה תיקיות מומלץ ........ 321

קונפיגורציה: JSON / YAML + משתני סביבה ................................ 322

scripts: אוטומציה של הרצות......................................................... 322

בדיקות: תיקיית tests + pytest.................................................... 323

Git ו-gitignore.: מה לא לשמור ..................................................... 324

Dockerfile מינימלי........................................................................ 324

CI/CD בסיסי .................................................................................. 325

דוגמה מרכזית: mini_text_analyzer כתבנית לפרויקט חדש......... 325

Best Practices ............................................................................... 326

סיכום הספרון – מה למדנו ואיך להמשיך ...................................... 326

# מבוא

## תוכנית מקצועית למתכנתים בעידן ה-AI

הבינה המלאכותית משנה את עולם הפיתוח.
מנועי חיפוש, אפליקציות ו-APIs חכמים

כולם נשענים על מתכנתים שמבינים לא רק קוד, אלא גם מודלים.

הסדרה **AI Developer World-Class Series**

נועדה להפוך מתכנתים למובילי פיתוח בעידן הזה.

אנשים שיודעים לתכנן, לכתוב ולבנות כמו אנשי AI מודרניים.

זו סדרת לימוד מעשית שבנויה כסולם מקצועי:
מהבסיס התכנותי ועד להובלת פרויקטי AI בארגון.
כל ספרון הוא שלב עצמאי במסע שמלמד לא רק **איך לכתוב קוד, אלא איך לחשוב כמו מפתח AI**.

הסדרה כוללת נושאים מגוונים:
**Python**, Data Engineering, Machine Learning, LLMs, RAG Systems, AI Agents, MLOps ו-AI Strategy.

## הספרון הראשון – פייתון לעידן ה-AI

פייתון היא שפת הדבק של עולם הבינה המלאכותית.
היא לא הכי מהירה, אבל היא גמישה, אינטואיטיבית ובעלת אקוסיסטם עצום שמאפשר לך לבנות כמעט הכול.

ספריות כמו **NumPy** ו-**Pandas** משמשות לעיבוד נתונים,
ובהמשך הדרך נכיר גם את: PyTorch, TensorFlow
ו-Hugging Face

הכלים שמביאים את פייתון לעולם המודלים וה-LLMs.

פייתון לא נועדה לביצועים גולמיים - אלא לארכיטקטורה נקייה וברורה.
בספרון זה נלמד לכתוב קוד קריא, מודולרי וניתן לתחזוקה בסיס הנדסי אמיתי לעולם ה-AI.

### הצעד הראשון שלך במסע

במהלך הלמידה נבנה יחד את **mini_text_analyzer**

כלי שורת פקודה (CLI) לעיבוד טקסטים, שיבצע ניקוי, **Tokenization**, חישוב סטטיסטיקות והשוואת דמיון סמנטי (**Cosine Similarity**).

זהו לא רק תרגיל - זה הבסיס לכל פרויקט AI שתפתח בעתיד.
מכאן תצא לדרך שבה כל שורה של קוד מקרבת אותך להבנת מודלים חכמים.

## פרק 1 - למה פייתון היא שפה חשובה בעידן ה-AI

אם אתה קורא את השורות האלה, כנראה שאתה כבר לא צריך שמישהו יסביר לך מה זה משתנה, לולאה או API.

אתה מתכנת מנוסה.

אולי ב-#C, ב-Java או ב-JavaScript

שכבר בנה מערכות אמיתיות, אבל רוצה להבין איך פייתון הפכה לשפה שמנהלת היום כמעט כל מערכת בינה מלאכותית רצינית.

המטרה של הספרון הזה היא **ללמד אותך לחשוב כמו מהנדס AI**, לא כמו "מתכנת מתחיל בפייתון".

לא נעסוק בתחביר הבסיסי (אם כי נרענן אותו בקצרה), אלא בדרך שבה מתכנת מנוסה משתמש בפייתון ככלי הנדסי, כלי שבאמצעותו בונים מודולים, מנהלים זרימות נתונים, מתעדים, בודקים ומריצים מערכות שצריכות לעבוד בלי הפסקה.

הספרון הזה לא נכתב למי שרוצה "להתנסות קצת ב-AI", אלא למי שרוצה **להכניס את ה-AI לקוד האמיתי שלו**

לעולמות של פרודקשן, ביצועים, תחזוקה והרחבה.

במילים אחרות, הוא מיועד למהנדסים ומתכנתים שרוצים להפוך את פייתון ממגרש משחקים לשפת תשתית מקצועית.

המטרה השנייה היא לגרום לך **לאהוב את פייתון מהצד ההנדסי שלה**.
לא רק בגלל שהיא "קלה לכתיבה", אלא כי היא מאפשרת לחשוב במונחים של מבנה, אחריות, מודולריות וניקיון.

כשניגשים אליה נכון, היא לא רק שפה,

היא **כלי תכנון** שמחבר בין הרעיון למימוש.

בסוף הספרון הזה תדע לא רק לכתוב קוד שעובד, אלא לכתוב קוד **שאפשר לבנות עליו מערכת שלמה**:
קריא, בדיק, נוח להרחבה, ומוכן לעבוד מול מודלים של AI כבר מהיום הראשון.

## פייתון ככלי הנדסי (לא רק סקריפטים)

כשמתכנתים מנוסים נתקלים לראשונה בפייתון, קל לטעות ולחשוב שהיא "שפת סקריפטים".
מספר שורות קוד, וכבר יש פלט.
אין טיפוסים מחייבים, אין הגדרות ארוכות, והכול רץ מיד.

אבל מאחורי הפשטות הזו מסתתרת **שפה הנדסית לכל דבר**, רק עם פילוסופיה אחרת.

פייתון לא נועדה להחליף את ++C או Rust בעומס חישובי. היא נועדה **לקשור ביניהן**.

היא יודעת לדבר עם קוד שנכתב בשפות אחרות, לנהל תהליכי pipeline שלמים, לטעון מודלים, להריץ אותם, לאסוף נתונים ולתעד תוצאות.

הכול מבלי שתצטרך להמציא תשתית מאפס.

במובן הזה, פייתון היא כמו "מערכת העצבים" של עולם ה-AI:
היא לא מבצעת את כל העבודה בעצמה, אבל היא זו שמחברת בין כל האיברים, האלגוריתמים, הנתונים, הספריות, הממשקים, וה-APIs.

שפה הנדסית אמיתית לא נמדדת רק במהירות ריצה, אלא גם ביכולת **לייצר מערכת שעובדת לאורך זמן**.

פייתון מאפשרת לך לעשות את זה בקלות יחסית:
להפריד אחריויות לקבצים ולמודולים, לעבוד עם מבני נתונים חזקים, להשתמש ב-type hints כדי לשמור על אמינות, ולשלב תיעוד ולוגים ברמה תעשייתית.

כמעט כל רכיב בארכיטקטורת AI, מ-Data Ingestion ועד Serving יכול להיכתב בפייתון.

ולכן כשאנחנו מדברים על "פייתון למהנדסי AI", אנחנו מתכוונים לשימוש בה לא בתור כלי להרצת דוגמה, אלא בתור **בסיס ארכיטקטוני**:
שפה שבאמצעותה מתכננים את כל ה-flow.

החל מטעינת הנתונים ועד הפקת התובנות.

מי שרואה בפייתון "שפת סקריפטים", מפספס את הסיפור האמיתי.

מי שלומד להתייחס אליה כשפת הנדסה, מגלה שהיא אחד הכלים העוצמתיים ביותר לבניית מערכות חכמות מודרניות.

## איך פייתון מריצה AI מאחורי הקלעים

כשאנחנו אומרים שפייתון היא "שפת הדבק" של עולם ה-AI, זו לא סיסמה, זו אמת טכנית.
פייתון כמעט אף פעם לא מבצעת את החישובים הכבדים בעצמה; היא מפעילה מנועים שנכתבו בשפות אחרות.

אלגוריתמים של למידה עמוקה מבוססים על חישובי מטריצות ענקיים ואלפי פעולות חישוב קטנות במקביל.
פה נכנס **ה-GPU (Graphics Processing Unit)** מעבד שמכיל אלפי ליבות קטנות, שמסוגלות לבצע פעולות רבות בו-זמנית.
בניגוד ל-CPU שעובד "עמוק" עם כמה ליבות חזקות, ה-GPU עובד

"רחב" מחשב הרבה דברים קטנים במקביל, וזה בדיוק מה שנדרש לאימון מודלים.

כדי להריץ קוד כזה נדרש **כרטיס מסך (GPU) תומך CUDA של NVIDIA** וגרסה מתאימה של PyTorch.
אם אין כזה, אפשר להריץ את אותו קוד גם על ה-CPU.
רק בלי להשתמש ב-cuda().

פייתון לא מבצעת את החישוב, אלא **מנהלת אותו** דרך ספריות חכמות כמו PyTorch, TensorFlow, או NumPy, שמאחורי הקלעים מריצות קוד ב-C++ ובו-CUDA.
לדוגמה:

```
import torch

x = torch.ones((1000, 1000)).cuda()
y = torch.ones((1000, 1000)).cuda()

z = x @ y  # כפל מטריצות – מתבצע בפועל על ה-GPU
print(z)
```

השורה z = x @ y נראית תמימה,
אבל מאחוריה מתבצעים **מיליוני חישובים מקבילים** על כרטיס גרפי
במהירות ש-Python לבדה לא הייתה מגיעה אליה לעולם.

זו אחת הסיבות שפייתון ניצחה בעולם ה-AI:
היא מאפשרת למתכנתים לכתוב קוד קריא ופשוט,
וליהנות מביצועים של שפות מערכת, בלי לגעת באף שורת CUDA אחת.
היא לא מתחרה ב ++C, היא **מנהלת אותה**.

## כללי סגנון PEP 8 – וקריאות קוד

בפייתון, **קריאות היא לא המלצה, היא עקרון יסוד**.
השפה הזו נבנתה מתוך מחשבה שקוד טוב הוא כזה שאפשר להבין אותו במבט ראשון, גם אם אתה לא זה שכתבת אותו.

בשפות אחרות מקובל לדבר על "Best Practices".

בפייתון יש מסמך אחד שמרכז את כולם

**PEP 8** (Python Enhancement Proposal 8)

שהוא התקן הלא-רשמי לסגנון קוד אחיד.

הוא לא נועד כדי להרשים בסטנדרטים, אלא כדי לגרום לקוד שלך **להיראות, להיקרא ולהתנהג** כמו קוד של קהילה מקצועית אחת גדולה.

ההיגיון פשוט: כשכולם כותבים באותו סגנון,
ה-diff ב-Git קטן יותר,
הסקירות מהירות יותר,
והמוח שלך לא מתאמץ להבין "איך הפעם החליטו לקרוא למשתנה הזה".

מספר עקרונות חשובים שכדאי להכיר כבר עכשיו:

- **שמות משמעותיים:** משתנים ופונקציות נכתבים ב-`snake_case`, מחלקות ב-`PascalCase`.
  אל תכתוב x כשאפשר `token_count`. הקוד לא אמור להיות חידת מילים.
- **רווחים הם קריאות:** סביב אופרטורים כמו `=`, `+`, או `==` השאר רווח אחד. זה אולי נראה שולי, אבל העין סורקת קוד טוב יותר ככה.
- **הזחה של ארבעה רווחים:** לא טאב, לא שניים. ארבעה. זה הסרגל הבלתי נראה שמחזיק את הקריאות של השפה.
- **שורה אחת למחשבה אחת:** כשפונקציה עושה יותר מדי, פרק אותה.
  פייתון בנויה על הרעיון של clarity over cleverness, עדיף פשוט וברור על פני "מתוחכם".

ומה שיפה זה שהקהילה עצמה דואגת לשמור על זה קל.
יש כלים אוטומטיים כמו **Ruff**, **Black** ו-**flake8** שיכולים לעצב

ולבדוק את הסגנון שלך אוטומטית.
כך תשמור על קוד נקי בלי להתווכח עם הצוות על מספר רווחים או מיקום סוגריים.

אבל מעבר לכללים, יש כאן פילוסופיה:
בפייתון, הקוד הוא קודם כול **אמצעי תקשורת** בין בני אדם.
המחשב יבצע כל מה שתכתוב לו, אבל המהנדס שיבוא אחריך צריך להבין למה כתבת את זה ככה.
זו הסיבה שכללי PEP 8 הם לא "עונש".

הם פשוט הדרך שבה קהילה שלמה שומרת על שפה אחת משותפת.

## עבודה נקייה עם קוד: Separation of Concerns

כשמערכת מתחילה לגדול, גם השורות הקטנות שאתה כותב היום הופכות במהירות לרשת של תלויות.
פונקציה אחת נוגעת בלוגיקה של אחרת, מודול קטן יודע יותר ממה שהוא צריך לדעת, והכול נהיה שביר.
זו בדיוק הנקודה שבה נכנס אחד העקרונות הכי חשובים בתכנות הנדסי **הפרדת אחריויות**, או בשם הקלאסי שלה: Separation of Concerns.

הרעיון פשוט, אבל משנה חיים:
כל רכיב במערכת צריך **לעשות דבר אחד, ולעשות אותו טוב**.
כשאתה מפריד בין האחריויות, אתה מונע מצב שבו שינוי קטן בקובץ אחד שובר חצי מערכת.

בפייתון, בגלל שכל כך קל לכתוב, קל גם ליפול למלכודת הזו:
"נוסיף כאן print, נוסיף שם פתיחת קובץ, נעדכן JSON תוך כדי עיבוד..." ופתאום יש לך בלגן שקשה לבדוק, קשה להרחיב, ובעיקר קשה להבין.

מערכת חכמה בנויה בשכבות.
לדוגמה:

- שכבת **קריאה** שאחראית על קלט (input) בלבד.
- שכבת **עיבוד** שמבצעת את הלוגיקה העסקית.
- שכבת **פלט** ששומרת תוצאות לקובץ, לבסיס נתונים או ל-API.

כשכל שכבה יודעת רק את מה שהיא צריכה לדעת, הקוד שלך נהיה גמיש, קל לבדיקה, וקל לתחזוקה.
רוצה לשנות את דרך הקריאה מקובץ לממשק רשת? אין בעיה.

שכבת העיבוד בכלל לא צריכה לדעת איך הנתונים הגיעו אליה.

פייתון מקלה מאוד ליישם את זה בזכות מבנה מודולרי טבעי:
אתה פשוט יוצר קובץ חדש, מייבא פונקציות רלוונטיות, וזהו.

יש לך שכבה נפרדת.

אין צורך להקים "פרויקט ענק" בשביל סדר.
לפעמים מספיק להוציא שלוש פונקציות לקובץ חדש, והקוד שלך הופך מניסוי לספריה אמיתית.

כשתעבוד ככה, תרגיש משהו מוזר קורה, הקוד שלך **נרגע**.
הוא מפסיק להילחם בעצמו, והופך להרמוני.
כל חלק יודע את מקומו, וכל שינוי מוגבל להקשר הנכון.

הפרדת אחריויות היא אולי אחד העקרונות הכי ישנים בתכנות, אבל בעולם ה-AI, שבו קוד, נתונים ומודלים משתלבים יחד, היא הופכת להיות קריטית יותר מתמיד.

## דוגמה מרכזית: סקריפט שמקבל טקסט ומחזיר JSON

לפני שנצלול לעומק הפרקים הבאים, נבנה יחד דוגמה קצרה שממחישה איך פייתון מרגישה כשכותבים בה כמו מהנדסים. לא סקריפט חד-פעמי, אלא בסיס למערכת אמיתית.

המטרה: לכתוב סקריפט שמקבל טקסט, מנקה אותו משוליים ורווחים, סופר מילים ומחזיר תוצאה כ-JSON תקין.

הקוד:

```
#!/usr/bin/env python3
"""
text_to_json.py
סקריפט פשוט שמחשב סטטיסטיקות בסיסיות על טקסט ומחזיר JSON.
"""

import json
from typing import Dict

def clean_text(text: str) -> str:
    """מסיר רווחים כפולים ותווי שורה מיותרים."""
    return " ".join(text.strip().split())

def text_stats(text: str) -> Dict[str, int]:
    """מחזיר מילון עם מספר מילים ותווים."""
    cleaned = clean_text(text)
    return {
```

```
    "word_count": len(cleaned.split()),
    "char_count": len(cleaned)
  }


def to_json(data: Dict) -> str:
  """ממיר מילון ל-JSON עם תמיכה בעברית."""
  return json.dumps(data, ensure_ascii=False, indent=2)

if __name__ == "__main__":
  sample_text = "  זהו טקסט קצר   עם   רווחים מיותרים.  "
  stats = text_stats(sample_text)
  result = to_json(stats)
  print(result)
```

הפלט:

```
{
  "word_count": 6,
  "char_count": 31
}
```

## למה זו דוגמה "הנדסית"?

לכאורה זה סקריפט קצר. אבל מאחוריו מסתתרת תפיסה שלמה:

- **פונקציות קטנות ומבודדות:** כל אחת עושה דבר אחד בלבד.
- **type hints:** מוסיפים בהירות, מאפשרים בדיקות סטטיות.

- **Docstrings:** תיעוד מובנה, נגיש לכל מי שיקרא את הקוד אחריך.
- **main guard** (if __name__ == "__main__")**:** מאפשר להשתמש בקוד גם כסקריפט עצמאי וגם כמודול ייבוא.

במקום סקריפט שמדפיס תוצאה "משוערת", יש כאן **יחידה הנדסית קטנה**: נקייה, קלה לבדיקה, ניתנת להרחבה.
אם נרצה בהמשך לשמור את הפלט לקובץ, או לקרוא את הקלט משורת פקודה נוכל לעשות זאת בלי לגעת בלוגיקה המרכזית.

זו בדיוק החשיבה שתרצה לאמץ לאורך כל הספרון:
לכתוב קוד קצר, קריא, עם גבולות ברורים בין שלבים, ולהרגיש שהכול בנוי לשימוש חוזר.

## סיכום: איך פייתון משרתת מהנדסי AI

לאורך הפרק ראינו שפייתון אינה רק שפה נוחה, אלא כלי הנדסי שמאפשר למהנדסי AI לבנות מערכות אמינות וגמישות.
הערך האמיתי שלה הוא בשילוב שבין פשטות, קריאות, ועוצמה.

לא במהירות או בתחביר.

פייתון מאפשרת לעבוד באותה קלות עם עולמות שונים:

- **מודלים מתמטיים** שדורשים דיוק,
- **קוד תשתיתי** שדורש סדר,
- **צינורות נתונים (Data pipelines)** שדורשים גמישות.

היא מאפשרת למהנדס לחשוב בצורה מערכתית, לכתוב קוד קריא ובדוק, ולחבר בין רכיבי המערכת בלי לאבד שליטה.

מהנדס AI טוב הוא זה שכותב קוד שאפשר להבין, לבדוק, ולשנות בביטחון.

**פייתון**

כשהיא נכתבת נכון

## מאפשרת בדיוק את זה:

שפה שמכבדת את הזמן של המתכנת, ואת הצורך של הצוות להבין מה קורה גם בעוד חצי שנה.

# פרק 2 - יסודות פייתון למתכנתים מנוסים תחביר בסיסי

## משתנים וטיפוסים מובנים (int, str, bool, None)

בפייתון הכול מתחיל בפשטות. אבל מאחורי הפשטות הזו מסתתרת הרבה חכמה.

בניגוד לשפות שמכריחות אותך להצהיר מראש על טיפוס המשתנה, בפייתון **הטיפוס שייך לערך, לא למשתנה עצמו**.

וזו אחת התפיסות החשובות ביותר להבין לפני שמתחילים לעבוד איתה ברצינות.

```
name = "Tomer"
age = 32
active = True
score = None
```

בארבע שורות קצרות יצרנו ארבעה טיפוסים שונים לגמרי.

בלי שום הצהרה מוקדמת:

`name` הוא מחרוזת (`str`),
`age` הוא מספר שלם (`int`),
`active` הוא בוליאני (`bool`),
ו-`score` מוגדר כ-`None`, המקבילה הפייתונית ל-`null`.

## הטיפוסים הבסיסיים בפייתון הם פשוטים אך עוצמתיים:

| טיפוס | דוגמה | תיאור קצר |
|---|---|---|
| **int** | 42 | מספר שלם |
| **float** | 3.14 | מספר עשרוני |
| **str** | "hello" | מחרוזת טקסט |
| **bool** | True / False | ערך לוגי |
| **None** | None | היעדר ערך |

אבל מה שחשוב להבין הוא שפייתון לא שומרת טיפוס "על המשתנה", אלא על האובייקט עצמו.

המשתנה הוא רק **מצביע**, מעין תווית שמפנה לאובייקט בזיכרון.

המשמעות היא שאפשר לשנות את הערך ואת הטיפוס של אותו שם משתנה בכל רגע:

```
x = 10
x = "עשר"
```

הקוד הזה חוקי לגמרי, אבל הוא גם מתכון לבאגים חמקמקים.

כי אם במקום אחד נניח ש-x הוא מספר, ובמקום אחר ננסה לחבר אותו למחרוזת, נקבל שגיאה בזמן ריצה.

זו אחת הסיבות לכך שמתכנתים מנוסים עובדים עם **type hints**.

גם אם פייתון לא דורשת אותם, הם הפכו לסטנדרט הנדסי שמקל על תחזוקה ובדיקות סטטיות.

```python
def add(a: int, b: int) -> int:
  return a + b
```

פייתון עצמה לא תכפה את זה בזמן ריצה,
אבל כלים כמו mypy או IDE חכם (VS Code, PyCharm) יתריעו
ברגע שתעביר טיפוס לא צפוי.
וכך נשמרת גם הגמישות של השפה וגם האמינות של הקוד.

אפשר לומר שפייתון בוטחת במתכנת.
היא נותנת לך חופש מוחלט. אבל גם דורשת ממך אחריות.

מי שיודע להשתמש בגמישות שלה נכון, מגלה שהיא שפה שיכולה להיות גם **מהירה לפיתוח** וגם **חזקה בהנדסה**.

## תנאים (if / elif / else) והיגיון בוליאני

היכולת "לקבל החלטות" היא הבסיס לכל תוכנה.
בפייתון, כמו בשפות אחרות, אנחנו עושים זאת באמצעות משפטי תנאי.

אבל הדרך שבה היא עושה את זה הרבה יותר קריאה וטבעית.

תנאי בפייתון הוא בעצם בדיקה שמחזירה ערך בוליאני: True או False.

התחביר פשוט להפליא:

```
temperature = 31

if temperature > 30:
    print("חם מאוד היום!")
elif temperature > 20:
    print("מזג אוויר נעים")
else:
    print("קריר בחוץ")
```

אין סוגריים, אין נקודה-פסיק, ואין מילים מיותרות.
הקוד נקרא כמעט כמו משפט בשפה טבעית.

אבל מאחורי הפשטות הזו יש כמה עקרונות שכדאי לזכור:

## ערכים "אמתיים" ו"שקריים"

בפייתון כמעט כל אובייקט יכול להתנהג כ-True או False בתוך תנאי.
הכלל פשוט:

| ערכים שמחזירים False | ערכים שמחזירים True |
| --- | --- |
| 0, 0.0, "", [], {}, None, False | כל דבר אחר |

```
name = ""
if name:
    print("יש שם משתמש")
else:
    print("לא הוזן שם")
```

כאן name הוא מחרוזת ריקה ולכן מתפרש כ-False.
זה אחד המאפיינים שהופכים את הקוד הפייתוני לקצר וברור
במקום לבדוק if len(name) > 0:, פשוט כותבים if name:.

## אופרטורים בוליאניים

פייתון שומרת על לוגיקה ברורה מאוד:

```
is_hot = temperature > 30
is_humid = True

if is_hot and is_humid:
    print("הפעל מזגן")
elif is_hot or is_humid:
    print("פתח חלון")
else:
    print("נעים בדיוק")
```

and ו-or הם לא מילים שמורות מקריות — הן קריאות ממש כמו אנגלית.
בניגוד לשפות כמו C# או Java שבהן תכתוב && ו-||, כאן אתה פשוט כותב את המילים עצמן.

## ביטוי תנאי מקוצר (ternary)

כשיש תנאי קצר, אין צורך בשלוש שורות.
אפשר לכתוב אותו כך:

```
status = "חם" if temperature > 30 else "נעים"
```

זהו ביטוי מלא שמחזיר ערך, לא רק פקודה.
במקום קוד מסורבל, אפשר לכתוב שורה אחת קריאה וברורה.
זו גם אחת הסיבות שפייתון הפכה לשפה שמפתחים אוהבים לכתוב בה
היא מאלצת אותך לכתוב **נקי, פשוט וקריא**.

## טיפ הנדסי קטן

בתוך מערכות אמיתיות, תנאים רבים נוטים להתפזר בקוד וליצור "סבך של ifים".
ככל שאתה מתבגר בתכנות, אתה לומד **להעביר את ההחלטות למבנה נתונים או לפונקציות**,
במקום להעמיס לוגיקה בוליאנית ארוכה.

לדוגמה:

```
actions = {
  "hot": "הפעל מזגן",
  "cold": "הפעל חימום",
  "normal": "אין צורך בפעולה"
}
```

```
state = "hot"
print(actions[state])
```

אותו רעיון של תנאים, אבל בלי if אחד אפילו.
זוהי הדרך הפייתונית: להעדיף **מבנים גמישים** על פני הסתעפויות לוגיות אינסופיות.

## לולאות (for ו-while)

לולאות הן מנוע הריצה של כל תוכנית.
הן אלה שחוזרות על פעולה שוב ושוב. עד שמגיעים לתוצאה, או עד שנמאס למחשב.
אבל בפייתון, גם כאן יש הבדל חשוב לעומת שפות אחרות:
היא נבנתה כך שכתיבת לולאה תהיה **ברורה וקריאה, לא מכאנית**.

### לולאת for

בפייתון, for לא נועדה "לספור צעדים" כמו ב-#C או Java.
היא **עוברת על אובייקטים איטרביליים**, כלומר על כל מבנה שאפשר לעבור עליו ברצף: רשימות, מחרוזות, קבצים, ואפילו *גנרטורים* (generators) – מבנים שיוצרים ערכים "תוך כדי תנועה" במקום לשמור את כולם מראש בזיכרון.

```
names = ["תמר", "נועם", "תומר"]

for name in names:
    print(f" היי{name}!")
```

הקוד הזה כמעט מדבר עברית:

“עבור על כל name בתוך names, והדפס שלום”.

אין כאן אינדקסים, אין תנאי עצירה, ואין צורך לכתוב לולאה אינסופית כדי לבדוק מתי להפסיק.  פייתון דואגת לזה בעצמה.

אם בכל זאת צריך את האינדקס – פשוט משתמשים ב-enumerate:

```
for i, name in enumerate(names):
    print(i, name)
```

התוצאה:

```
תמר 0
נועם 1
תומר 2
```

כך מקבלים גם את המיקום וגם את הערך, בלי לספור ידנית.

## לולאת while

בלולאת while משמשים כשלא ידוע מראש כמה פעמים צריך לחזור על משהו.

היא ממשיכה לרוץ **כל עוד התנאי מתקיים**.

```
count = 0

while count < 3:
    print("חוזרים על זה שוב...")
    count += 1

print("סיימנו!")
```

גם כאן אין סוגריים או נקודה-פסיק.

הכול פשוט, אך עם שליטה מלאה.

## שימוש ב-break ו-continue

לפעמים צריך לעצור לולאה באמצע, או לדלג על סבב אחד.

בפייתון עושים זאת בעזרת שתי מילים שמדברות בעד עצמן:

```
for n in range(10):
    if n == 5:
        break  # עצור לגמרי
    if n % 2 == 0:
        continue  # דלג על מספרים זוגיים
    print(n)
```

הפלט:

```
1
3
```

הלולאה נעצרת כשהיא פוגשת את המספר 5,
ודילגה על כל המספרים הזוגיים בדרך לשם.

## שימוש מתקדם: for ... else

אחד המאפיינים הפחות מוכרים של פייתון
לולאה שיכולה להכיל גם בלוק else,
שמתבצע רק אם **הלולאה הסתיימה באופן טבעי**, בלי break.

```
for n in range(5):
  if n == 3:
    print("נמצא!")
    break
else:
  print("לא נמצא")
```

אם break לא התרחש – יופעל ה-else.
זה מאפשר לכתוב לוגיקה נקייה בלי דגלים מיותרים.

## טיפ הנדסי: אל תכתוב לולאה איפה שצריך ביטוי

בפייתון יש עיקרון אחד שמלווה אותך לאורך כל הדרך:
**אם אפשר לבטא משהו בביטוי, אל תכתוב לולאה שלמה בשבילו.**

למשל, כדי לסכום מספרים, אין צורך לעבור עליהם בלולאה:

```
numbers = [1, 2, 3, 4]
total = sum(numbers)
```

הקוד הקצר הזה גם קריא יותר וגם יעיל יותר.

זו אחת התכונות שהופכות את פייתון לשפה שמתאימה לעידן ה-AI:

היא **מעודדת לחשוב ברמת הכוונה, לא ברמת השלבים**.

## חיתוכים ופעולות על מחרוזות (Slicing)

מעט מאוד תכונות בפייתון נראות פשוטות כל כך, אבל חוסכות כל כך הרבה קוד כמו **slicing**.

זו הדרך שבה ניגשים לחלקים מתוך רצף: רשימה, מחרוזת, או אפילו מערך NumPy. היא נראית טריוויאלית, אבל היא אחת הסיבות שפייתון הפכה לכל כך יעילה בעיבוד נתונים.

### הבסיס: תחילת החיתוך, סופו והצעד

התחביר הכללי הוא:

```
sequence[start:end:step]
```

שלושת הפרמטרים האלה הם כולם **אופציונליים**.

דוגמה פשוטה על מחרוזת:

```
text = "Artificial Intelligence"
print(text[0:10])  # 'Artificial'
print(text[:10])   # אותו דבר - מתחילת המחרוזת
print(text[11:])   # 'Intelligence'
print(text[::-1])  # היפוך מלא
```

מספר דברים שכדאי לשים לב אליהם:

- הטווח **אינו כולל** את האינדקס האחרון (end), אלא עוצר לפניו.
- ניתן להשמיט את ההתחלה, את הסוף, או את שניהם.
- צעד (step) שלילי מאפשר "ללכת אחורה" ברצף.

זו לא רק תחבולה נוחה. זו גישה שמחייבת אותך **לחשוב ברצפים** במקום בלולאות אינדקסים.
וכשמתחילים לעבוד על טקסטים או נתוני זמן, זה חוסך עשרות שורות קוד.

### Slicing במחרוזות, זה לא רק קיצור דרך

מחרוזת (str) היא בעצם רצף תווים, ולכן כללי החיתוך חלים עליה באופן טבעי:

```
word = "Python"
print(word[1:4])   # 'yth'
print(word[-2:])   # 'on'
```

אפשר לחשוב על זה כמו על "מבט מדויק פנימה" למחרוזת.
והכי חשוב slicing בפייתון **אף פעם לא זורק חריגה** אם הגבולות חורגים.
במקום זאת, הוא פשוט מחזיר את מה שקיים:

```
print(word[2:99])  # 'thon' - לא שגיאה!
```

זו החלטת עיצוב שמגנה עליך מטעויות קטנות,
ומאפשרת להריץ מניפולציות על טקסטים בביטחון.

### פעולות שימושיות על Strings

מכיוון שמחרוזת היא immutable (לא ניתנת לשינוי במקום),
כל פעולה מחזירה **עותק חדש**.

דוגמאות חשובות:

```python
text = "  Python is Awesome!  "
print(text.strip())     # הסרת רווחים מסביב
print(text.lower())     # אותיות קטנות
print(text.upper())     # אותיות גדולות
print(text.replace("Awesome", "Powerful"))
print(text.split())     # ['Python', 'is', 'Awesome!']
```

כל אחת מהפקודות האלה מחזירה מחרוזת חדשה, המקורית
נשארת כשהייתה.
זו התנהגות חשובה במיוחד בעיבוד טקסטים במערכות AI.

כשנרצה לנקות, לנרמל, או להכין טקסטים לאימון,
נשתמש באותן פונקציות בדיוק. אבל נוודא שתמיד נשמור על
המקור בצד.

## חיבור בין מחרוזות

בפייתון חיבור מחרוזות מתבצע בעזרת הסימן + או באמצעות
"".join().

ההבדל ביניהם קטן כשמדובר בשתי מילים,
אבל נהיה דרמטי כשמדובר באלפי חיבורים בלולאה.

```
# חיבור ישיר
result = "Hello " + "World"

# חיבור יעיל ברשימות
words = ["AI", "is", "changing", "everything"]
result = " ".join(words)
```

join() יעילה פי כמה,
כי היא יוצרת את המחרוזת הסופית **בפעולה אחת בזיכרון**,
במקום ליצור מאות עותקים זמניים.

במילים אחרות
אם אתה מהנדס, תתרגל להשתמש ב-join כבר מהיום הראשון.

## טיפ הנדסי

כמעט כל מערכת AI תיגע בטקסטים: שמות קבצים, תגיות,
תיאורים, נתוני JSON.

היכולת שלך **לטפל במחרוזות במהירות ובזהירות** תקבע אם הקוד
שלך ישרוד ב-production או יקרוס על תו שונה אחד.

**אל תזלזל בזה.**

כתיבה נכונה של פעולות טקסט היא אחת המיומנויות השקטות שמבדילות בין סקריפטיסט למהנדס.

## f-strings לפורמט טקסט מודרני

אם אתה מגיע משפות כמו C#, Java או JavaScript,
אתה בטח רגיל לראות מחרוזות שמלאות בסימני פלוס, סוגריים וסימני אחוז.
בפייתון, כל זה כבר היסטוריה.
מאז גרסה 3.6, פייתון הציגה את אחד השיפורים הקטנים אך המשמעותיים ביותר בשפה: **f-strings**.

## למה זה חשוב?

פייתון עוסקת בקריאות.

ולפני f-strings, עיצוב טקסט היה אחד המקומות הכי פחות קריאים בקוד.
למשל, אם רצית להדפיס פרטי משתמש בשיטה הישנה, היית כותב:

```
name = "תמר"
age = 29
print("שמי {} ואני בת {}".format(name, age))
```

וזה עוד בסדר.
אבל תנסה להכניס חישוב או תנאי קטן – והכול מיד נהיה מבולגן.

f-strings פתרו את זה בדרך פייתונית טיפוסית:
במקום "לקרוא" למחרוזת מבחוץ, פשוט כותבים את הביטוי ישירות בתוכה.

## התחביר הפשוט

הוספת האות `f` לפני מחרוזת מאפשרת לשלב בתוכה ביטויים דינמיים, ממש כאילו כתבת קוד רגיל:

```
name = "תמר"
age = 29
print(f"שמי {name} ואני בת {age}")
```

הפלט:

```
שמי תמר ואני בת 29
```

בתוך הסוגריים המסולסלים `{}` אפשר לשים **כל ביטוי חוקי של פייתון:**
חישוב, קריאה לפונקציה, תנאי. הכול.

```
print(f"בעוד שנה אהיה בת {age + 1}")
```

או אפילו:

```
print(f"השם באותיות גדולות: {name.upper()}")
```

## עיצוב מספרים ותאריכים

f-strings כוללות גם תחביר מובנה לעיצוב ערכים. בלי להסתבך עם פונקציות חיצוניות.

```
pi = 3.1415926535
print(f"פאי בקירוב: {pi:.3f}")    # פלט: פאי בקירוב: 3.142
```

ניתן גם לשלוט בצורה מדויקת בעימוד וביישור של טקסט, ממש כמו ב-printf הקלאסית:

```
for n in range(1, 4):
    print(f"{n:>3} → {n*n:>5}")
```

פלט:

```
  1 →     1
  2 →     4
  3 →     9
```

החלק שבין הסוגריים המסולסלים `{}` מאפשר להגדיר איך הערך יוצג.

בדוגמה הזו:

- `:>3` אומר: תיישר לימין בתוך שדה שרוחבו שלושה תווים.
- `:>5` אומר: תיישר לימין בתוך שדה שרוחבו חמישה תווים.

אם המספר קצר מהרוחב שהוגדר, פייתון מוסיפה רווחים משמאל כך שכל הערכים יהיו מיושרים באותה עמודה.

זו דרך פשוטה להציג טקסטים או מספרים בצורה מסודרת, במיוחד כשמדפיסים טבלאות או פלט רב-שורות.

## f-strings לעומת חיבור מחרוזות

f-strings הן לא רק נוחות, הן גם **מהירות יותר**.
פייתון מבצעת את ההחלפה בזמן קומפילציה של המחרוזת,
כך שאין צורך לקרוא לפונקציות או לבנות אובייקטים זמניים.
במילים אחרות: גם קריאות, גם ביצועים.

## טיפ הנדסי

f-strings הן אחת הדוגמאות היפות ביותר לדרך שבה פייתון חושבת:
**פשטות שקופה, עם עומק אמיתי**.

הן מאפשרות לך לשלב קוד ולוגיקה בצורה שמרגישה טבעית, ומצמצמות באגים שנובעים מטעויות בפורמט מחרוזות.

בעולם שבו כמעט כל מערכת AI מפיקה טקסטים, לוגים, תוצאות או קונפיגורציות,
f-strings הם הכלי שאתה לא רוצה לוותר עליו.
קטנים, אלגנטיים, ומושלמים למהנדס שחושב בהקשרים של נתונים, לא של תחביר.

## כתיבה קומפקטית – List Comprehensions

יש רגע בכל מסע עם פייתון שבו אתה מבין שאתה יכול לכתוב בלולאה של שורה אחת, מה שבשפות אחרות היה לוקח חמש.

הרגע הזה מגיע כשאתה מגלה את **List Comprehensions**

הדרך הפייתונית לבנות רשימות בצורה תמציתית, ברורה, ומפתיעה ביעילותה.

### איך זה נראה

במקום:

```
squares = []
for n in range(5):
  squares.append(n * n)
```

אפשר פשוט לכתוב:

```
squares = [n * n for n in range(5)]
```

והתוצאה למעשה זהה.

רשימה חדשה נבנית תוך כדי מעבר על טווח הערכים.

אבל ההבדל הוא לא רק בקיצור שורות, זה שינוי צורת חשיבה.

במקום "תיצור רשימה, עבור בלולאה, הוסף איברים",

פייתון אומרת:

**"בנה רשימה של איברים שעומדים בתנאי מסוים."**

## עם תנאים

אפשר לשלב גם תנאי ישירות בתוך הביטוי:

```
evens = [n for n in range(10) if n % 2 == 0]
```

כאן נבנית רשימה של כל המספרים הזוגיים בין 0 ל-9.
הלולאה והתנאי מתמזגים לשורה אחת נקייה,
שאפשר לקרוא אותה כמו משפט רגיל:
"קח את כל n בטווח 10 אם הוא זוגי."

## דוגמה מעשית על טקסטים

```
text = "Machine learning is amazing"
words = [w.lower() for w in text.split() if len(w) > 3]
print(words)
```

פלט:

```
['machine', 'learning', 'amazing']
```

זו בדיוק החשיבה שנרצה לראות אצל מהנדס AI:

לא איך לעבור על כל מילה, אלא **איך לסנן ולעבד בקו אחד ברור**.

## Comprehensions לכל מבנה

המבנה הזה עובד לא רק עם רשימות.
פייתון מאפשרת להשתמש באותו רעיון גם עם סוגים אחרים של אוספים:

```
# Dictionary comprehension
squares = {n: n * n for n in range(5)}

# Set comprehension
squares = {n * n for n in range(5)}

# Generator comprehension
nums = (n * n for n in range(1_000_000))
```

### Set comprehension

יוצר קבוצה (set) של ערכים ייחודיים – כלומר בלי כפילויות.

### Dictionary comprehension

יוצר מילון, שבו כל איבר הוא צמד מפתח-ערך.

### Generator comprehension

עובד בדומה ל-list comprehension, אבל במקום ליצור רשימה בזיכרון, הוא "מפיק" איברים אחד-אחד בזמן ריצה.
זה שימושי במיוחד כשעובדים עם כמויות גדולות של נתונים. למשל בעיבוד רצף לוגים או ב-Data pipelines.

## טיפ הנדסי

כשהקוד שלך הופך להיות דחוס מדי, הוא מאבד קריאות אבל כשהוא כתוב נכון, comprehension הוא סימן לקוד בוגר ומדויק.

הוא חוסך משתנים זמניים, מפחית באגים, ומשקף דרך חשיבה מוצהרת:

**מה אתה רוצה להשיג, לא איך אתה משיג את זה.**

זו הסיבה שכל מתכנת פייתון מנוסה משתמש בהם בזהירות, אבל כמעט תמיד מחייך כשהוא עושה זאת.

## דוגמה מרכזית: ניקוי רשימת משפטים עם תנאים

הגיע הזמן לראות איך כל מה שלמדנו בפרק הזה מתחבר יחד למשהו אמיתי.
נבנה פונקציה קטנה שמקבלת רשימה של משפטים גולמיים, מנקה אותם מרווחים מיותרים ומחזירה רק את אלה שאינם ריקים.

במילים אחרות, ניקוי ראשוני של טקסט,
שלב שמופיע כמעט בכל מערכת AI, לפני עיבוד או ניתוח תוכן.

### הקוד

```
def clean_sentences(sentences: list[str]) -> list[str]:
    """
    מנקה רשימת משפטים:
    1. מסירה רווחים מסביב.
    2. מדלגת על משפטים ריקים.
    3. מחזירה רשימה חדשה, נקייה.
    """
    return [s.strip() for s in sentences if s.strip()]
```

פשטות קיצונית, אבל גם יעילות.
לולאה, תנאי, ופעולת טקסט – כולם משולבים בשורה אחת נקייה, בזכות **list comprehension**.

## דוגמה להרצה

```python
def clean_sentences(sentences: list[str]) -> list[str]:
    """
    מנקה רשימת משפטים:
    1. מסירה רווחים מסביב.
    2. מדלגת על משפטים ריקים.
    3. מחזירה רשימה חדשה, נקייה.
    """
    return [s.strip() for s in sentences if s.strip()]


raw = [
    "  שלום עולם!  ",
    "",
    "    זה מבחן קטן",
    "   ",
    "AI משנה הכול"
]

clean = clean_sentences(raw)
print(clean)
```

פלט:

```
['שלום עולם!', 'זה מבחן קטן', 'AI משנה הכול']
```

במספר שורות בלבד ניקינו רשימה של טקסטים,
הסרנו רווחים, סיננו שורות ריקות,
וקיבלנו רשימה נקייה שמוכנה לשלב הבא בעיבוד נתונים.

## תנאי נוסף

אם נרצה לסנן גם משפטים קצרים מדי, נוכל להוסיף תנאי נוסף:

```
def clean_sentences(sentences: list[str]) -> list[str]:
    return [s.strip() for s in sentences if s.strip() and len(s.strip()) > 5]
```

כעת כל משפט קצר מדי פשוט ייעלם מהפלט.

זו בדיוק הגמישות שהופכת את פייתון לשפה "חושבת":

אפשר להוסיף לוגיקה, מבלי לשבור את הקריאות.

## טיפ הנדסי

הפונקציה הזו אולי קטנה,

אבל מאחוריה עומדים מספר עקרונות חשובים למהנדס AI:

- **תנאי אחד ברור עדיף על מספר שורות מיותרות.**
- **לולאה קצרה ונקייה עדיפה על שרשרת if ים.**
- **List comprehension חוסך באגים ומבהיר את הכוונה.**

במערכות אמיתיות, תראה דפוס כזה שוב ושוב - פונקציות שמבצעות פעולה אחת מדויקת, נקיות, ניתנות לבדיקה, וכתובות כך שכל מתכנת בצוות יבין אותן מיד.

## סיכום: איך לחשוב "פייתונית"

ללמוד פייתון זה קל.
אבל **לחשוב פייתוני** זה כבר משהו אחר לגמרי.
זה המעבר בין כתיבת קוד שעובד לבין כתיבת קוד שמרגיש נכון.

במהלך הפרק הזה ראינו את המרכיבים הבסיסיים של השפה:
משתנים, תנאים, לולאות, חיתוכים ו-list comprehensions.
אבל יותר חשוב מהתחביר, זו **הפילוסופיה** שמאחורי כל אחד מהם.

### פשטות לפני הכול

פייתון מעודדת אותך לוותר על תחבולות ולהעדיף בהירות.
אם אפשר לכתוב דבר אחד ברור בשורה אחת, אין סיבה לפרוס אותו על שלוש.
זו שפה שבנויה על ההנחה שהקוד שלך ייקרא לא רק על-ידך, אלא על-ידי מהנדס אחר, ואם הוא יבין אותו מיד, סימן שכתבת נכון.

### תחשוב ב"מה", לא ב"איך"

בשפות אחרות אתה נוטה לתאר למחשב איך לעשות כל שלב.
בפייתון, אתה מתאר **מה אתה רוצה שיקרה**.
זו הסיבה שהשפה הזו כל כך מתאימה לעולם ה-AI:
היא מאפשרת להתמקד בתוצאה.

בעיבוד, בלוגיקה, בנתונים.
ולא בצעדים הטכניים שבדרך.

## פחות קוד, יותר משמעות

הקיצור הוא לא המטרה, הוא התוצאה.
כשקוד קצר נובע ממבנה נכון, הוא קריא יותר, קל לבדיקה, ונשבר פחות.
פייתון מאלצת אותך להיות ממוקד: כל שורה צריכה לשרת רעיון.

אין מקום לקישוטים, אבל גם אין צורך בתבניות מסובכות כדי להיראות "מתוחכם".

## הדגש ההנדסי

המהנדס שחושב "פייתונית" לא מחפש קסמים, הוא מחפש סדר.
הוא מפריד בין אחריויות, ממקם פונקציות לפי תפקידן, ובונה מודולים קטנים וברורים.
פייתון לא תעשה את זה במקומך, אבל היא נותנת לך את הכלים לכך:
קריאות, גמישות, ונגישות מיידית לכל שלב בתהליך.

לכתוב פייתון זו לא רק שאלה של תחביר, זו שאלה של **גישה**.
גישה של פשטות, אמינות, וחשיבה מוצהרת.
זו שפה שמתגמלת אותך על בהירות, לא על טריקים.
וכשמאמצים אותה כך, מגלים שכל פרויקט, קטן או ענק
יכול להפוך לקוד שקל להבין, קל לבדוק, וקל לאהוב.

# פרק 3 – מבני נתונים שימושיים

## למה מבני נתונים קריטיים ב-AI

אם בפרקים הקודמים למדנו איך פייתון חושבת,
הפרק הזה עובר לשאלה החשובה יותר. **איך היא זוכרת**.
כל מערכת חכמה, מאימון מודל שפה ועד ניתוח טקסט קצר
קמה ונופלת על הדרך שבה אנחנו **מאחסנים, מעבדים, וניגשים לנתונים**.

מבני הנתונים הם הלב הפועם של כל מערכת AI.
הם קובעים כמה מהר תשלוף תוצאה,
כמה זיכרון תבזבז בדרך,
וכמה קל יהיה לשנות את האלגוריתם בלי לפרק הכול מחדש.

פייתון אולי נראית פשוטה, אבל מתחת לפני השטח היא מסתירה **מערכת מבני נתונים מתקדמת במיוחד**.
חמשת המבנים הבסיסיים שלה:
`tuple`, `set`, `dict`, `list`, ו-`collections`

מכסים כמעט כל תרחיש אפשרי של אחסון ועיבוד מידע.

והיופי? כולם מובנים בשפה,
נגישים מיד. בלי ייבוא, בלי קונפיגורציה, בלי טקסיות מיותרת.

## מבני נתונים בעולם ה-AI

כשאנחנו בונים מערכת בינה מלאכותית,
אנחנו לא עובדים רק עם "מספרים"
אלא עם טקסטים, תגים, **ייצוגים וקטורים** (embeddings), מדדים, תוצאות.
ובכל אחד מהמקרים האלה הבחירה במבנה הנתונים הנכון יכולה להיות ההבדל בין מערכת שעובדת חלק לבין אחת שנתקעת בלופ אין-סופי.

- ניתוח טקסטים? רשימות (list) ומילונים (dict).
- ספירת תדירויות? Counter מתוך collections.
- חיפוש ייחודיות? קבוצות (set).
- שמירה על סדר הכנסה? deque או OrderedDict.

מהנדס AI מנוסה לא חושב רק על הפתרון, הוא חושב על **הייצוג**.
איך הנתונים יזרמו, איפה הם יאוחסנו,
ואיך לשמור על איזון בין ביצועים לקריאות.

## למה זה חשוב דווקא בפייתון

פייתון הפכה לשפה המובילה בעולם ה-AI לא רק בזכות הספריות שלה, אלא בגלל הדרך שבה היא מאפשרת לעבוד עם נתונים בצורה טבעית.

היכולת לעבור בין רשימה למילון, לסנן, למיין וליצור מבנים חדשים תוך שניות היא מה שמאפשר למהנדסים להתנסות, למדל ולבנות אב-טיפוס במהירות שיא.

אבל יש גם מחיר: הגמישות הזו מזמינה בלבול.
קל מאוד לבחור במבנה "שעובד". אבל לא בהכרח "נכון".

בפרק זה נלמד לזהות את ההבדלים, להבין את יתרונות כל מבנה, ולראות איך להשתמש בהם **כמו מהנדסים**, לא כמו חובבנים.

לפני שנצלול לקוד, חשוב לזכור:
פייתון היא לא רק שפה שמריצה אלגוריתמים.

היא שפה שמעצבת את צורת החשיבה שלך על נתונים.
אם תשלוט במבני הנתונים שלה,
תוכל להתמודד עם כמעט כל בעיה בעולם ה-AI
בלי לגעת בשורה אחת של NumPy או TensorFlow.

## list רשימות דינמיות ופעולות נפוצות

הרשימה (list) היא אחד הכלים הפשוטים אבל גם אחד החשובים ביותר בפייתון.

במובן מסוים, היא ה-Swiss Army Knife של עולם הנתונים:
גמישה, קלה לשימוש, ויכולה לשמש כמעט לכל צורך
מאיסוף תוצאות ועד אחסון טקסטים, ייצוגים וקטוריים
(embeddings)או מדדים ממודלים.

### איך יוצרים רשימה

```
numbers = [1, 2, 3, 4, 5]
names = ["תמר", "נועם", "תומר"]
mixed = [1, "AI", True, None]
```

רשימה יכולה להכיל ערכים מטיפוסים שונים
וזו אחת הסיבות שפייתון כל כך גמישה.
מהנדס טוב יודע להשתמש בזה בזהירות:
אם כל איבר שונה לגמרי, כנראה שהנתונים עצמם לא מאורגנים היטב.

### פעולות נפוצות

```
data = [10, 20, 30, 40]

print(len(data))     # 4 – אורך הרשימה
print(data[0])      # 10 – איבר ראשון
print(data[-1])     # 40 – איבר אחרון
```

```
print(data[1:3])    # [20, 30] – חיתוך
```

רשימות בפייתון מתנהגות כמעט כמו מערך,
אבל עם יתרון עצום: הן **דינמיות**
אפשר להוסיף, להסיר ולשנות ערכים תוך כדי ריצה.

```
data.append(50)      # מוסיף לסוף
data.insert(0, 5)    # מוסיף בהתחלה
data.remove(30)      # מסיר ערך מסוים
print(data)          # [5, 10, 20, 40, 50]
```

רשימות גם ניתנות למיון בקלות:

```
data.sort()
data.reverse()
```

והכול קורה במקום, בלי צורך להחזיר עותקים חדשים (כמו במחרוזות).

## מעבר על רשימה

```
for value in data:
  print(value)
```

אין צורך במונה, ואין צורך לבדוק את האורך
פייתון פשוט "מבינה" איך לעבור על הרשימה.

אם בכל זאת צריך גם אינדקס, משתמשים ב-enumerate:

```
for i, value in enumerate(data):
  print(f"{i}: {value}")
```

## פעולות שימושיות למהנדסי AI

רשימות משמשות כמעט בכל Pipeline של AI:

- רשימת משפטים שממתינה לעיבוד.
- רשימת תוצאות ממודל.
- רשימת קבצים בתיקייה.

שילוב קטן עם comprehension או תנאים,
ומתקבל קוד מדויק ויעיל:

```
scores = [88, 92, 75, 100, 67]
high_scores = [s for s in scores if s >= 90]
print(high_scores)  # [92, 100]
```

## שכפול רשימות

שכפול הוא נקודה רגישה בפייתון:

כשמעתיקים רשימה עם = נוצר רק **מצביע** חדש, לא עותק אמיתי.

```
a = [1, 2, 3]
b = a
b.append(4)
print(a)  # [1, 2, 3, 4]
```

שני המשתנים מצביעים על אותה רשימה!

כדי לשכפל באמת, יש להשתמש באחת מהדרכים הבאות:

```
b = a.copy()
```

```
# או
b = list(a)
# או
b = a[:]  # slicing מלא
```

## dict מילונים (key → value) ושימושים

אם הרשימה היא לב המערכת,
המילון (dict) הוא המוח שלה.
זהו מבנה הנתונים שבו פייתון באמת מצטיינת
מיפוי של מפתח לערך, כמו טבלה קטנה בזיכרון,
עם גישה מיידית לכל נתון בלי צורך לעבור על כל הרשימה.

### איך נראֶה מילון

```
person = {
  "name": "תמר",
  "age": 29,
  "is_active": True
}
```

כל ערך מאוחסן תחת מפתח ייחודי.
המפתחות במילון חייבים להיות בלתי ניתנים לשינוי (immutable).
בדרך כלל אלו מחרוזות או מספרים, אך אפשר להשתמש גם ב-
tuple (למשל, לציון מיקום או זוג ערכים).

גישה לערכים פשוטה וברורה:

```
print(person["name"])  # תמר
print(person["age"])   # 29
```

אם תנסה לגשת למפתח שלא קיים, תקבל שגיאה (KeyError).

לכן בפועל, נהוג להשתמש בפונקציה בטוחה יותר:

```
print(person.get("email", "לא צוין אימייל"))
```

אם המפתח לא קיים, מוחזר הערך ברירת-המחדל.

### הוספה, עדכון והסרה

```
person["city"] = "תל אביב"  # הוספה
person["age"] = 30          # עדכון
del person["is_active"]     # הסרה
```

פשוט, ישיר, וקריא.

אין צורך במתודות מורכבות או בקונסטרוקציות מסורבלות.

### מחיקת ערכים ממילון

לפעמים נרצה להסיר פריט ממילון קיים. מפתח שלם, או ערך מסוים בתוך מילון פנימי.

```
person = {"name": "Tomer", "age": 13, "city": "Petah Tikva"}

# מחיקה של מפתח שלם
del person["age"]

# מחיקה בטוחה – אם המפתח לא קיים לא תתקבל שגיאה
```

```
person.pop("city", None)
```

אם רוצים לרוקן את כל המילון:

```
person.clear()
```

## מחיקת ערכים ממילון מורכב

לפעמים נרצה למחוק רק שדה מתוך פריט, לפעמים פריט שלם, ולפעמים את כל הנתונים.

בדוגמה הבאה רואים את כל המצבים הנפוצים:

```
students = {
    1: {"name": "Charlie", "age": 16, "grade": 82},
    2: {"name": "Noam", "age": 15, "grade": 95},
    3: {"name": "Tomer", "age": 17, "grade": 78},
}

# מוחקים רק את הגיל של התלמיד Charlie
del students[1]["age"]

print(students)
# {1: {'name': 'Charlie', 'grade': 82},
#  2: {'name': 'Noam', 'age': 15, 'grade': 95},
#  3: {'name': 'Tomer', 'age': 17, 'grade': 78}}

# מוחקים את התלמיד עם המזהה 2
del students[2]
print(students)
```

```
# {1: {'name': 'Charlie', 'grade': 82},
#  3: {'name': 'Tomer', 'age': 17, 'grade': 78}}

# מוחקים תלמיד לפי שם (לפי ערך פנימי)
for student_id, info in list(students.items()):
    if info["name"] == "Charlie":
        del students[student_id]

print(students)
# {3: {'name': 'Tomer', 'age': 17, 'grade': 78}}

# מוחקים את כל התלמידים
students.clear()
print(students)
# {}
```

כך אפשר לשלוט בדיוק ברמה של המחיקה. משדה יחיד ועד ניקוי מוחלט של כל המידע.

במילונים גדולים, כדאי להעדיף מחיקה ממוקדת כדי לשמור על יעילות הקוד.

## מעבר על מילון

שלושה דרכים עיקריות לעבור על מילון:

```
for key in person:
    print(key)  # רק המפתחות

for value in person.values():
    print(value)  # רק הערכים

for key, value in person.items():
    print(key, "→", value)  # שניהם
```

הצורה השלישית: `items()` – היא הנפוצה ביותר,
בעיקר כשאנחנו מעבדים נתונים לצורכי לוגים, JSON או ניתוח תוצאות.

## Dict בעולם ה-AI

מילונים נמצאים בכל מקום:

- פלטים של מודלים (`{"label": "positive", "score": 0.97}`)
- קונפיגורציות של מערכות
- שליפת פרמטרים ממודלים
- אחסון תוצאות ביניים

זו דרך טבעית לתאר **נתונים מובְנים**, בלי צורך במחלקות מורכבות.

```
model_output = {
```

```
  "text": "AI is amazing",
  "tokens": 4,
  "score": 0.98
}
```

כך נראים כמעט כל הפלטים שמוחזרים
מ-OpenAI, Hugging Face, או LangChain.
המבנה הזה מאפשר גישה ברורה, המרה ל-JSON, וחיסכון בזמן פיתוח.

### פעולות שימושיות

```
data = {"a": 1, "b": 2, "c": 3}

print("a" in data)       # בדיקת קיום מפתח
print(data.keys())       # מציג את כל המפתחות
print(data.values())      # מציג את כל הערכים
print(list(data.items()))  # מציג רשימת זוגות (מפתח, ערך)
```

הפלט בפועל יהיה:

```
True
dict_keys(['a', 'b', 'c'])
dict_values([1, 2, 3])
[('a', 1), ('b', 2), ('c', 3)]
```

אפשר גם למזג שני מילונים בקלות:

```
defaults = {"lang": "he", "mode": "prod"}
```

```
custom = {"mode": "dev"}
config = {**defaults, **custom}
```

אם יש התנגשות במפתחות, הערכים מהשני גוברים.

## set קבוצות של ערכים ייחודיים

אם list היא רשימת נתונים,
ו-dict הוא מיפוי נתונים,
אז set הוא השומר בשער.
הוא לא מתעניין בסדר, רק בשאלה אחת פשוטה:
**האם הערך הזה כבר קיים?**

מבנה הנתונים set מייצג **קבוצה של ערכים ייחודיים**,
כלומר לא יכולים להיות בו כפילויות.
זה הופך אותו לכלי אידיאלי למצבים שבהם רוצים לוודא ייחודיות,
לספור סוגים שונים של פריטים,
או לבצע פעולות חיתוך ואיחוד בין קבוצות נתונים.

### יצירה ושימוש בסיסי

```
tags = {"AI", "ML", "NLP", "AI"}
print(tags)  # {'AI', 'ML', 'NLP'}
```

כמו שאפשר לראות, הערך "AI" הופיע פעמיים, אבל נשמר רק פעם אחת.
פייתון שומרת רק את הערכים הייחודיים, ללא סדר קבוע.

אפשר גם ליצור קבוצה מרשימה קיימת:

```
numbers = [1, 2, 2, 3, 3, 3]
unique_numbers = set(numbers)
print(unique_numbers)  # {1, 2, 3}
```

## פעולות קבוצתיות

Set מאפשר לבצע פעולות מתמטיות קלאסיות בקלות,
כמו איחוד, חיתוך והפרש
וזה בדיוק מה שמועיל בניתוח נתונים מורכבים.

```
a = {"AI", "ML", "Data"}
b = {"AI", "Vision", "Robotics"}

print(a | b)  # איחוד: {'AI', 'ML', 'Data', 'Vision', 'Robotics'}
print(a & b)  # חיתוך: {'AI'}
print(a - b)  # הפרש: {'ML', 'Data'}
print(a ^ b)  # ערכים ייחודיים לשני הצדדים בלבד
```

אלה פעולות רשת קלאסיות: איפה יש חפיפה, איפה לא.
בעולם ה-AI הן שימושיות במיוחד להשוואת תגים, קטגוריות או
מזהים ייחודיים.

## שימושים מעשיים

### סינון כפילויות:

```
tokens = ["ai", "ai", "is", "awesome"]
unique_tokens = list(set(tokens))
```

**בדיקה מהירה של שייכות** (מהירה בהרבה מרשימה):

```
if "ai" in unique_tokens:
  print("נמצא!")
```

**השוואת קבוצות תוצאות** ממודלים שונים:

```
model_a = {"positive", "neutral"}
model_b = {"neutral", "negative"}
overlap = model_a & model_b
print(overlap)  # {'neutral'}
```

## הבדל חשוב מול רשימות

Set **אינו שומר על סדר.**

אם הסדר חשוב, השתמש ב-list או ב-dict (שמגרסה 3.7 שומר על סדר הכנסת המפתחות).

לעומת זאת, אם העדיפות היא למהירות ולייחודיות

set ינצח בכל פעם.

## טיפ הנדסי

במערכות AI, חישובים רבים מסתמכים על זיהוי חפיפות, כפילויות וייחודיות

למשל, כמה מילים חדשות הופיעו בטקסט,
או כמה מזהים שונים עברו בתהליך מסוים.
set הוא מבנה הנתונים המושלם לזה:
קל, מהיר, ועם פעולות שמאפשרות לחשוב ברמה של קבוצות, לא של לולאות.

זו בדיוק החשיבה ההנדסית שפייתון מעודדת
לא לבדוק "אחד אחד", אלא להסתכל על התמונה הכוללת.

## tuple רצף קבוע ואי-שינוי

אם list היא רשימה גמישה ודינמית,
tuple הוא ההפך המוחלט.

**רצף קבוע, יציב, שלא ניתן לשנות לאחר שנוצר**.

זה אולי נשמע כמו מגבלה,
אבל במערכות חכמות, דווקא היכולת *לא* להשתנות היא לעיתים היתרון הכי גדול.

## איך נראה tuple

```
point = (10, 20)
print(point[0])  # 10
print(point[1])  # 20
```

התחביר כמעט זהה לרשימה. רק עם סוגריים עגולים במקום מרובעים.

מה שמייחד את tuple הוא העובדה שלא ניתן לשנות אותו:

```
point[0] = 5  # ❌ תגרום לשגיאה: 'tuple' object does not support item assignment
```

ברגע שיצרת tuple, הערכים שבו קבועים.

זו תכונה חשובה כשמדובר בנתונים שאתה **לא רוצה שישתנו בטעות**,

למשל תוצאות ביניים, קואורדינטות, או נתונים שמיועדים לשימוש חוזר.

## יצירה והמרה

```
data = (1, 2, 3)
single = (5,)  # גם כן tuple צריך פסיק בודד!
as_list = list(data)
as_tuple = tuple(as_list)
```

פסיק אחד קטן הוא מה שהופך ביטוי ל-tuple אמיתי.

בלי הפסיק, פייתון תזהה את זה כערך רגיל, לא כקבוצה.

## פירוק חכם Unpacking

אחת הסיבות ש-tuples כל כך נוחים היא היכולת לפרק אותם בקלות:

```
x, y = (10, 20)
print(x, y)  # 10 20
```

ה-tuple "נפתח" לשניים או שלוש משתנים, לפי הסדר.

וזה עובד גם בפונקציות שמחזירות כמה ערכים:

```
def get_stats():
    return (10, 5, 2)

max_val, avg, min_val = get_stats()
```

במקום להחזיר מילון, לפעמים עדיף להחזיר tuple כשהמבנה פשוט וברור.

## שימושים מעשיים ב-tuple

### תוצאה קבועה מפונקציה:

כשפונקציה מחזירה כמה ערכים, שימוש ב-tuple מבטיח שמבנה התוצאה יציב וברור:

```
def analyze(text: str) -> tuple[int, int]:
    """מחזירה (מספר מילים, מספר תווים)."""
    return len(text.split()), len(text)
```

השימוש ב-tuple מבהיר שהתוצאה **קבועה וחד-צורתית**.

כעת אפשר להשתמש בערכים כך:

```
words, chars = analyze("שלום עולם")
```

tuple מבטיח שהתוצאה תישאר תמיד באותו מבנה.

**מפתח במילון:**

מאחר ש-tuple הוא immutable, ניתן להשתמש בו כ-key במילון:

```
coords = {(10, 20): "A", (15, 25): "B"}
print(coords[(10, 20)])  # A
```

אי אפשר לעשות זאת עם list, ולכן tuple הוא מבנה אידיאלי לייצוג מיקום, צבע או כל זוג ערכים יציב.

**הגנה על נתונים:**

כשלא רוצים שאף חלק בקוד ישנה ערכים בטעות, tuple מספק שכבת הגנה טבעית (read-only):

```
rgb = (255, 128, 0)
# rgb[0] = 0  → תגרום לשגיאה
```

כך ניתן לשמור על נתונים קריטיים "נעולים".

**עבודה עם מערכים או מימדים:**

בספריות כמו NumPy או pandas, **tuple** משמשת לתיאור מימדים (dimensions) או קואורדינטות קבועות:

```
array.shape  # לדוגמה מחזיר (3, 5)
```

היא מאפשרת לייצג את מימדי הנתונים בצורה ברורה וחד-משמעית.

## העברת פרמטרים לפונקציות:

ניתן "לפתוח" tuple ישירות כפרמטרים לפונקציה:

```
def show(x, y):
    print(x, y)

point = (10, 20)
show(*point)  # 10 20
```

זוהי דרך שימושית להעביר אוספי נתונים לפונקציות בצורה אלגנטית.

## tuple לעומת list

| תכונה | list | tuple |
|---|---|---|
| **שינוי ערכים** | כן | לא |
| **גודל משתנה** | כן | לא |
| **ביצועים** | איטי יותר | מהיר יותר |
| **שימוש טיפוסי** | נתונים דינמיים | נתונים קבועים |

**ההבדל העיקרי הוא בגישה:**

רשימות נועדו לשינויים,
tuple נועד ליציבות.
במערכות AI, זה שימושי במיוחד כשמעבירים נתונים בין שלבים ב-pipeline
אפשר להיות בטוחים שאף שלב לא שינה אותם בדרך.

## טיפ הנדסי

כמעט כל פונקציה שאתה כותב יכולה להחזיר tuple קטן של ערכים
וזה לא "קיצור דרך", אלא **שיטה הנדסית** לשמור על קוד פשוט וברור.
אם המבנה צפוי, tuple עדיף על מילון.
אם אתה צריך שמות שדות, תעבור ל-dataclass או ל-TypedDict.

## collections: defaultdict, Counter, deque

עד עכשיו דיברנו על מבני הנתונים הבסיסיים של פייתון:
set, dict, list, ו-tuple.
אבל לפעמים אתה רוצה משהו קצת יותר מתוחכם
מבנה נתונים שעדיין פשוט, אבל **חוסך ממך קוד שחוזר על עצמו**.

בשביל זה קיימת הספרייה collections.
היא חלק מובנה מפייתון, ואין צורך להתקין כלום.

## שלושת הכוכבים שלה:

defaultdict, Counter, ו-deque.

כל אחד מהם נועד לפתור בעיה יומיומית אחת, בצורה אלגנטית.

## Defaultdict: מילון עם ערך ברירת מחדל

נניח שאתה רוצה לספור כמה פעמים כל תו מופיע במחרוזת:

```
text = "banana"
freq = {}

for ch in text:
    if ch not in freq:
        freq[ch] = 0
    freq[ch] += 1
```

זה עובד, אבל מכוער.

עם defaultdict, אין צורך לבדוק אם המפתח קיים:

```
from collections import defaultdict

text = "banana"
freq = defaultdict(int)

for ch in text:
    freq[ch] += 1

print(freq)
```

ברגע שפייתון רואה מפתח חדש, היא פשוט יוצרת ערך ברירת-
מחדל (במקרה הזה – 0),
ומאפשרת להמשיך כאילו הוא כבר קיים.
כך חוסכים קוד הגנתי מיותר.

## Counter ספירה חכמה

אם כל מה שאתה צריך הוא ספירה,
פייתון כבר מספקת פתרון ישיר עוד יותר:

```
from collections import Counter

words = ["ai", "is", "amazing", "ai", "is", "ai"]
count = Counter(words)
print(count)
```

פלט:

```
Counter({'ai': 3, 'is': 2, 'amazing': 1})
```

אפשר לשלב אותו עם פעולות מתקדמות:

```
print(count.most_common(1))    # [('ai', 3)]
print(count["is"])          # 2
```

Counter שומר על מבנה של מילון,
אבל מתנהג כמו כלי סטטיסטי קטן
מושלם לניתוח טקסטים, לוגים או תוצאות ממודלים.

## deque תור דו-כיווני

deque (נשמע כמו "deck") הוא רשימה מהירה במיוחד שמאפשרת להוסיף ולהסיר איברים **משני הכיוונים** ביעילות גבוהה.

```
from collections import deque

queue = deque(["task1", "task2", "task3"])
queue.append("task4")       # מוסיף לסוף
queue.appendleft("urgent")  # מוסיף להתחלה

print(queue)  # deque(['urgent', 'task1', 'task2', 'task3', 'task4'])

queue.pop()       # מסיר מהסוף
queue.popleft()   # מסיר מההתחלה
```

בניגוד ל-list, הוספה או הסרה בתחילת רשימה גדולה **לא דורשת העתקה של כל האיברים**.

במערכות שבהן יש תורים (queues) או זרימת נתונים (streams), deque הוא הבחירה הנכונה.

## למה זה חשוב ב-AI

שלושת המבנים האלה חוזרים על עצמם שוב ושוב בפרויקטי AI:

- Defaultdict – ניהול תוצאות ביניים, ניקוי נתונים, או קיבוץ לפי קטגוריות.
- Counter – ספירת מילים, טוקנים, תגיות או קטגוריות.

- Deque – אחסון נתונים זמניים בתהליכים אסינכרוניים או בזמן אמת.

הם קטנים, מהירים, ומובנים בשפה.
ואת כל מה שהם עושים היית יכול לכתוב ידנית
אבל השורה הזו מסכמת הכול:
**למה לכתוב קוד כשפייתון כבר כתבה אותו בשבילך?**
הוא כותב **פחות** קוד שעובד **חכם** יותר.

## דוגמה מרכזית: סטטיסטיקות טקסט עם dict ו-Counter

מערכות AI מתבססות על נתונים,
אבל לפני שיש מודל, יש טקסטים.
לפני, embeddings יש מילים.
ולפני למידה עמוקה, יש **סטטיסטיקות פשוטות**.

בדוגמה הזו נשתמש ב-dict וב-Counter כדי לנתח טקסט קצר:
לספור מילים, לחשב ממוצע אורך, ולמצוא את המילה הנפוצה ביותר.
זו אותה לוגיקה שמופיעה כמעט בכל שלב של עיבוד שפה טבעית (NLP).

## הקוד

```
from collections import Counter
import re

def simple_word_stats(text: str) -> dict[str, float | str]:
    """
    מחשב סטטיסטיקות בסיסיות על טקסט:
    - מספר מילים
    - מספר תווים
    - אורך ממוצע של מילה
    - המילה הנפוצה ביותר
    """
    # ניקוי בסיסי של סימני פיסוק
    clean_text = re.sub(r"[^\w\s]", "", text)
    words = clean_text.split()
    num_words = len(words)
    num_chars = len(clean_text)
    avg_length = sum(len(w) for w in words) / num_words if num_words
else 0
    most_common = Counter(words).most_common(1)[0][0] if words
else ""

    return {
        "num_words": num_words,
        "num_chars": num_chars,
        "avg_word_length": round(avg_length, 2),
        "most_common_word": most_common
    }
```

## דוגמת הרצה

```
sample = "AI is amazing. AI changes everything!"
print(simple_word_stats(sample))
```

פלט:

```
{'num_words': 6, 'num_chars': 35, 'avg_word_length': 5.0,
'most_common_word': 'AI'}
```

## הסבר קצר

- `re.sub` מנקה סימני פיסוק כדי לקבל מילים נקיות.
- `split()` מפרק את הטקסט לרשימת מילים (`list`).
- `Counter` מחשב בקלות את שכיחות כל מילה.
- הפונקציה מחזירה מילון (`dict`) שמוכן לכתיבה לקובץ JSON או לוג.

מספר שורות, אבל מאחוריהן כל החשיבה ההנדסית של פייתון:

- ניצול של מבני נתונים פשוטים במקום קוד הגנתי.
- שימוש ב-Counter במקום לבנות לולאה ידנית.
- קריאות מוחלטת, כל מה שהקוד עושה כתוב במפורש.

## סיכום: מתי להשתמש בכל מבנה נתונים

הבחירה במבנה הנתונים הנכון היא מה שמבדיל בין **קוד שעובד** לקוד שבנוי **נכון**.

בפייתון יש ארבעה כלים עיקריים, וכל אחד נועד למטרה אחרת.

list – כשצריך **סדר וגמישות.**

מתאימה לרצפים משתנים כמו משפטים, תוצאות או מדדים.

אם אתה בודק הרבה "האם הערך קיים?", עדיף לעבור ל-**set**.

tuple – כשצריך **יציבות.**

בלתי ניתן לשינוי, מושלם לערכים קבועים כמו **קואורדינטות, תוצאות או מפתחות במילון**.

set – כשצריך **ייחודיות ובדיקות מהירות.**

שומר רק ערכים ייחודיים, מעולה לסינון והשוואה בין קבוצות.

dict – כשצריך **קשרים בין נתונים.**

מיפוי של **מפתח לערך**, הבסיס ל-**JSON**, קונפיגורציות ונתונים מובנים.

collections – כשצריך **מבני נתונים מתקדמים** שמוכנים לשימוש מידי.

במקום להמציא לוגיקה משלך, תשתמש במה שפייתון כבר בנתה עבורך:

- **Defaultdict** – מילון שיודע להתמודד לבד עם ערכים חסרים.

- **Counter** – לספירה חכמה של מילים, תגים, תגובות – כל דבר.
- **Deque** – תור דו-כיווני מהיר ויציב.

שלושתם חוסכים קוד, טעויות וזמן.

## הבחירה הנכונה = קוד יציב יותר

| צורך | מבנה מתאים | מאפיין בולט |
|---|---|---|
| **סדר וערכים משתנים** | list | ניתנת לשינוי, שומרת על סדר |
| **מיפוי מהיר לפי מפתח** | dict | גישה ישירה, קריאה טבעית |
| **ייחודיות ובדיקת קיום** | set | מהירה במיוחד, בלי כפילויות |
| **נתונים קבועים מראש** | tuple | יציבות, בטיחות, ביצועים |
| **ספירה, קיבוץ או תור** | collections | פתרונות חכמים ומוכנים |

בסוף זה פשוט:
list – **סדר**, tuple – **יציבות**, set – **ייחודיות**, dict – **הקשרים**,
ו-collections – **כלים חכמים שפייתון כבר בנתה עבורך**.

כשאתה בוחר נכון – הקוד שלך נשאר קצר, ברור ועמיד.

# פרק 4 – פונקציות בפייתון

## פונקציות כיחידת בניין קריטית

כל שפת תכנות מאפשרת לכתוב פונקציות.
אבל בפייתון, פונקציות הן הרבה יותר מסתם דרך "לא לחזור על קוד"
הן **אבן הבניין המרכזית של כל מערכת יציבה**.

אם משתנים הם החומר הגולמי שלך,
הפונקציות הן המכונות שמעבדות אותו.
במערכת AI אמיתית הן מה שמפריד בין בלגן של קוד ניסיוני לבין תהליך הנדסי מדויק:
כל שלב מנוקה, נמדד, מתועד, וניתן לבדיקה.

## למה זה קריטי בפרויקטי AI

במערכות למידת מכונה ובינה מלאכותית,
הקוד מתמלא מהר מאוד בחזרות
קריאות API, ניקוי נתונים, בדיקות חריגות, מדידת זמן, שמירת תוצאות.

אם לא מפרקים את כל זה לפונקציות ברורות,
הקוד הופך לארוך, שביר, וכמעט בלתי ניתן לתחזוקה.

במקום לכתוב את הכול בלולאה אחת גדולה,
אנחנו מפרקים את המערכת **ליחידות קטנות שיש להן מטרה אחת בלבד**

פונקציות שמקבלות קלט, מבצעות פעולה מדויקת, ומחזירות תוצאה ברורה.

זו הגישה שמאפשרת:

- לבדוק כל שלב בנפרד (Unit Testing)
- למדוד ביצועים בצורה ממוקדת
- לכתוב קוד שאפשר להרחיב בלי לשבור
- ולעבוד בצוות בלי דריכה הדדית על רגליים

**פונקציה טובה היא כמו מכונה**

היא צריכה שלוש תכונות:

1. **קלט ברור:** מה היא מקבלת.
2. **פלט צפוי**: מה היא מחזירה.
3. **שום תופעות לוואי מיותרות:** היא לא משנה משתנים חיצוניים בלי סיבה.

היא לא חייבת להיות קצרה, אבל היא חייבת להיות ממוקדת.
אם פונקציה עושה "גם ניקוי, גם עיבוד, גם הדפסה",
זה סימן שאתה צריך לחלק אותה לשלוש.

## תזכורת קטנה

פייתון לא מאלצת אותך לכתוב פונקציות
אבל היא **מתגמלת אותך** כשאתה כותב אותן נכון.
קוד פונקציונלי הוא קריא יותר, קל יותר לבדיקה,
ובמערכות AI: הוא גם מאפשר לנתק רכיבים לצורך ניסויים מבלי לשבור את המערכת כולה.

במילים פשוטות:
פונקציות הן תזכורת שפיתוח טוב מתחיל בהנדסה,
לא בניסוי וטעייה.

## פרמטרים והחזרת ערכים (כולל tuple unpacking)

פונקציה היא כמו "מכונה קטנה":
אתה נותן לה קלט, היא עושה עיבוד, ומחזירה תוצאה.
אבל בפייתון, המכונה הזו גמישה בהרבה ממה שנראה במבט ראשון.
היא לא מחייבת אותך בהצהרות ארוכות,
אבל מאפשרת לך לבנות מנגנון קלט ופלט מדויק, כמעט כמו במערכת טיפוסים חזקה.

### פרמטרים בסיסיים

```
def greet(name):
    print(f"היי{name}!")
```

הקריאה פשוטה:

```
greet("תמר")
```

אבל מאחורי הפשטות הזו מסתתרת לא מעט עוצמה.
כשאתה מעביר ערך לפונקציה, פייתון לא יוצרת ממנו עותק חדש,
היא פשוט נותנת לפונקציה גישה לאותו אובייקט בזיכרון.
אם זה אובייקט שניתן לשינוי, כמו רשימה או מילון, כל שינוי שתעשה
עליו בתוך הפונקציה **ישפיע גם מחוץ לה**.
וזה מצוין כשזה מה שאתה רוצה, אבל עלול להיות כאב ראש כשלא.

כדי להימנע מהפתעות כאלה, כדאי לעבוד על **עותק חדש** כשלא
רוצים "אפקט צד".

## החזרת ערכים

פונקציה מחזירה ערך בעזרת return.

אם אין return, היא מחזירה אוטומטית None.

```
def add(a, b):
    return a + b

result = add(5, 7)
print(result)  # 12
```

אין צורך להגדיר טיפוס החזרה,

אבל מהנדסים מנוסים מוסיפים type hints כדי לשמור על סדר:

```
def add(a: int, b: int) -> int:
    return a + b
```

## החזרת כמה ערכים יחד

בניגוד לשפות אחרות שדורשות מחלקה או מבנה נתונים מיוחד, פייתון מאפשרת להחזיר מספר ערכים בבת אחת. פשוט על ידי החזרת tuple:

```
def min_max_avg(values: list[int]) -> tuple[int, int, float]:
    return min(values), max(values), sum(values) / len(values)
```

ועכשיו אפשר לפרק את התוצאה ישירות:

```
nums = [10, 5, 8, 12]
low, high, avg = min_max_avg(nums)
print(low, high, avg)  # 5 12 8.75
```

זה נקרא **tuple unpacking**,
וזו אחת הסיבות שפייתון כל כך קריאה
היא מאפשרת "להוציא מידע ממבנה" בלי תחביר כבד.

## פרמטרים בעלי שם (Keyword Arguments)

פייתון מאפשרת גם קריאה מפורשת לפי שם,
מה שהופך את הקוד לברור יותר:

```
def connect(host: str, port: int):
    print(f"Connecting to {host}:{port}...")

connect(port=8080, host="localhost")
```

כך אתה לא תלוי בסדר הפרמטרים,
והקוד שלך כמעט קורא את עצמו.

### טיפ הנדסי

בפונקציות טובות, המידע **נכנס ברור ויוצא ברור**.
בלי הפתעות, בלי תלות במשתנים גלובליים.

העיקרון פשוט: פונקציה צריכה לדעת הכול על מה שנכנס אליה.
ושום דבר על מה שקורה מחוצה לה.

במערכות AI זה קריטי: פונקציות נבדקות בנפרד, נמדדות בנפרד,
וחייבות להחזיר תוצאות צפויות גם כשהקלט משתנה.

## ערכי ברירת מחדל ו-pitfalls עם mutable defaults

פייתון מאפשרת להגדיר **ערכי ברירת מחדל** לפרמטרים,
וכך להפוך פונקציות לגמישות וידידותיות יותר:

```
def greet(name: str, greeting: str = "שלום"):
  print(f"{greeting}, {name}!")
```

כעת אפשר לקרוא לפונקציה בשתי דרכים:

```
greet("תמר")         # שלום, תמר!
greet("נועם", "היי")  # היי, נועם!
```

זה נוח, קריא, ומקצר המון קוד.

אבל מתחת לפני השטח מסתתרת אחת המלכודות הוותיקות

והמסוכנות ביותר בשפה:

ה-**mutable default trap**.

## המלכודת: ערכי ברירת מחדל נוצרים פעם אחת בלבד

פייתון מחשבת את ערכי ברירת-המחדל **רק פעם אחת, בזמן הגדרת הפונקציה**,

ולא בכל פעם שהיא נקראת.

אם הערך הוא משתנה Mutable (כמו `dict`, `list`, או `set`),

הוא יישמר בזיכרון בין קריאות לפונקציה.

```
def add_item(item, items=[]):
    items.append(item)
    return items

print(add_item("A"))  # ['A']
print(add_item("B"))  # ['A', 'B'] ← מפתיע!
```

במקום להתחיל רשימה חדשה בכל קריאה,

הפונקציה משתמשת באותה רשימה מהפעם הקודמת.

זו לא "תקלה", זה העיצוב של השפה,

אבל אם לא יודעים עליו, זה עלול ליצור באגים חמקמקים מאוד.

### הפתרון הנכון

הדרך הבטוחה היא להשתמש ב-`None` כברירת מחדל,

וליצור את האובייקט בתוך גוף הפונקציה:

```
def add_item(item, items=None):
```

```
  if items is None:
    items = []
  items.append(item)
  return items
```

כעת כל קריאה מתחילה עם רשימה חדשה:

```
print(add_item("A"))  # ['A']
print(add_item("B"))  # ['B']
```

פשוט, ברור, ובטוח.

## למה זה חשוב בפרויקטי AI

במערכות AI ו-Data Processing,

פונקציות רבות בונות מבני נתונים או אוספות תוצאות בין קריאות.

אם אתה נופל במלכודת הזו,

הנתונים שלך "נוזלים" מקריאה לקריאה

ופתאום אתה מקבל תוצאות לא צפויות בלי להבין למה.

לכן, כלל הברזל:

לעולם אל תשתמש בערך ברירת-מחדל Mutable.

תמיד העדף None ויצירה מחדש בתוך הפונקציה.

## טיפ הנדסי

מפתחים מנוסים נופלים במלכודת הזו לא בגלל חוסר ידע,

אלא כי הקוד "נראה נכון".

אבל חלק מהיותך מהנדס הוא **לזהות מצבים שנראים תמימים אך מסוכנים**.
בפייתון, זוהי אחת הדוגמאות הקלאסיות.

## ‎*args ו-‎**kwargs מתי ואיך

פייתון מתייחסת לפונקציות כאל ישויות דינמיות.
אין חובה להגדיר מראש כמה פרמטרים הפונקציה תקבל
אפשר לכתוב פונקציה שמקבלת **מספר משתנה של ארגומנטים**,
ולנהל אותם בצורה חכמה בתוך הגוף שלה.

### ‎*args ארגומנטים מיקומיים

הכוכבית היחידה * משמשת לאיסוף כל הפרמטרים **המיקומיים** (positionals) לתוך tuple.

```
def summarize(*args):
    print(args)

summarize(1, 2, 3)
# פלט: (1, 2, 3)
```

כל הערכים שהועברו נכנסים ל-args כרצף (tuple).
אפשר לעבור עליהם בלולאה, לסכום אותם, או לעבד אותם כרשימה:

```
def add_all(*numbers):
```

```
    return sum(numbers)

print(add_all(3, 5, 10))  # 18
```

אם אתה לא יודע מראש כמה פרמטרים יגיעו, זו הדרך הבטוחה לטפל בזה.

## ארגומנטים לפי שם **kwargs

שתי כוכביות (**) משמשות לאיסוף פרמטרים בשם (keyword arguments).

הם נשמרים בתוך מילון (dict) כך שקל לגשת אליהם לפי שם:

```
def describe_person(**kwargs):
    print(kwargs)

describe_person(name="תמר", age=29, city="תל אביב")
# {'name': 'תמר', 'age': 29, 'city': 'תל אביב'}
```

אפשר להשתמש בהם ישירות:

```
def describe_person(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
```

## שילוב של שניהם

פייתון מאפשרת לשלב בין שני הסוגים — קודם *args, אחר כך **kwargs:

```
def debug(*args, **kwargs):
    print("ARGS:", args)
    print("KWARGS:", kwargs)

debug(10, 20, mode="test", verbose=True)
```

פלט:

```
ARGS: (10, 20)
KWARGS: {'mode': 'test', 'verbose': True}
```

כך ניתן לכתוב פונקציות גמישות שמקבלות קלט מכל סוג כמו פונקציות לוג, דיבוג, או עיבוד נתונים גנרי.

## שימוש מעשי בפרויקטי AI עם **kwargs

במערכות AI הממשק משתנה כל הזמן. היום צריך שני פרמטרים, מחר מצטרפים עוד שלושה. **kwargs מאפשר לך לשמור על יציבות בקוד גם כשהממשק מתרחב.

### שלב 1: גרסה בסיסית וגמישה

הפונקציה יודעת מה חשוב לה, וכל השאר נכנס דרך המילון בלי לשבור כלום.

```
def run_inference(model, **config):
```

```
    # ערכי ברירת מחדל
    temperature = config.get("temperature", 0.7)
    max_tokens = config.get("max_tokens", 256)

    # משתמשים רק במה שצריך כרגע
    return model.generate(temp=temperature,
limit=max_tokens)

# דוגמת שימוש
config = {
    "temperature": 0.8,
    "max_tokens": 512,
    "top_p": 0.9,       # שדה חדש שלא בשימוש עדיין
    "logprobs": True    # שדה נוסף שלא בשימוש
}

result = run_inference(gpt_model, **config)
print(result)
# פלט לדוגמה:
# "המודל יצר תשובה עם טמפרטורה 0.8 ומגבלת 512 טוקנים"
```

הפונקציה משתמשת רק בפרמטרים שהיא מכירה. שדות חדשים לא שוברים שום דבר.

## שלב 2: מתקדמים בלי לשנות קריאות קיימות

אחרי חודש מוסיפים תמיכה ב-top_p. הקריאות נשארות זהות.

```
def run_inference(model, **config):
    temperature = config.get("temperature", 0.7)
    max_tokens  = config.get("max_tokens", 256)
    top_p       = config.get("top_p", 1.0)  # שימוש חדש

    return model.generate(
        temp=temperature,
        limit=max_tokens,
        top_p=top_p
    )

# אותה קריאה בדיוק אותה קריאה בדיוק כמו קודם
result = run_inference(gpt_model, **config)
print(result)
# פלט לדוגמה:
# "המודל יצר תשובה עם temp=0.8, limit=512, top_p=0.9"
```

זה כל הרעיון: מתחילים פשוט, מוסיפים יכולות כשצריך, והקוד שמסביב לא נשבר.

### טיפ הנדסי

*args ו-**kwargs הם לא טריק תחבירי, הם **תבנית הנדסית** שמאפשרת להרחיב מערכות מבלי לשנות ממשקים.

אבל כמו כל גמישות, צריך לדעת היכן לעצור:
אם פונקציה משתמשת ב-kwargs לכל דבר, סימן שהיא צריכה ריפקטור.

כלל זהב: השתמש ב-args* ו-kwargs** **רק כשאתה באמת צריך גמישות**,
לא כדי להסתיר מבנה לא ברור של נתונים.

## פונקציות כערכים (First-Class Citizens)

בפייתון, פונקציות מתנהגות כמו כל משתנה אחר

אפשר לשמור אותה במשתנה, להעביר אותה כפרמטר,
להחזיר אותה כפלט, ואפילו לבנות פונקציות שמייצרות פונקציות אחרות.

זו לא סתם גמישות תחבירית, זו דרך חשיבה.
במקום להעביר רק נתונים, אתה מעביר **התנהגות**.

### פונקציה שמאוחסנת במשתנה

```
def greet(name: str) -> str:
    return f"היי {name}!"

say_hi = greet
print(say_hi("נועם"))  # היי נועם!
```

כאן say_hi הוא משתנה שמצביע לפונקציה greet.
לא בוצעה קריאה לפונקציה, רק "שימור" שלה.
זה מאפשר להעביר פונקציות בדיוק כמו שמעבירים מחרוזת או רשימה.

### פונקציה כפרמטר לפונקציה אחרת

```
def apply_twice(func, value):
    return func(func(value))

def add_one(x: int) -> int:
    return x + 1

print(apply_twice(add_one, 3))  # 5
```

כאן הפונקציה apply_twice מקבלת פונקציה אחרת (add_one) ומפעילה אותה פעמיים על הערך.
אין כאן קסם, רק ניצול של העובדה שפונקציות הן אובייקטים לכל דבר.

### החזרת פונקציה מתוך פונקציה

כן, גם זה אפשרי. וזה אפילו שימושי מאוד:

```
def make_multiplier(factor: int):
    def multiply(x: int) -> int:
        return x * factor
```

```
    return multiply

double = make_multiplier(2)
print(double(5))  # 10
```

זו פונקציה שיוצרת פונקציות.

כשקוראים ל-make_multiplier(2) היא לא מחזירה מספר, אלא **פונקציה חדשה** בשם multiply שכבר "מכירה" את הערך של factor.

כל פעם שנקרא ל-double, היא תשתמש באותו ערך שנשמר.

במקרה הזה 2.

התופעה הזו נקראת **Closure**.

המשמעות היא שהפונקציה הפנימית **זוכרת** את הערכים שהיו זמינים בזמן שנוצרה, גם אחרי שהפונקציה החיצונית כבר הסתיימה.

זה שימושי במיוחד כשאתה רוצה ליצור **פונקציות מותאמות מראש**

למשל פונקציה שמחשבת מחיר עם מע"מ, מקדם הנחה, או כל קבוע אחר:

```
add_vat = make_multiplier(1.17)
discount = make_multiplier(0.9)

print(add_vat(100))  # 117.0
```

```
print(discount(200))  # 180.0
```

כל אחת מהן "זוכרת" את המקדם שלה,

וזה מה שהופך את ה-Closure לכלי כל כך אלגנטי לבניית לוגיקה חכמה ופשוטה.

## למה זה חשוב בפרויקטי AI

במערכות AI אנחנו מריצים קוד שחוזר על עצמו
ניקוי טקסט, מדידה, לוגים, עיבוד פלטים, ועוד.
כשפונקציות הן אובייקטים, אפשר לבנות **תהליכים גנריים**,
שמקבלים "התנהגות" כפרמטר, בלי לשכפל לוגיקה.

```
def process_text(text: str, transform):
   clean = text.strip().lower()
   return transform(clean)

result = process_text("  AI IS AMAZING  ", lambda t:
t.replace("ai", "ML"))
print(result)  # ml is amazing
```

כך פונקציה אחת יכולה לטפל בכל תרחיש
רק מחליפים את ההתנהגות שמועברת לה.

## טיפ הנדסי

במערכות בוגרות, פונקציות אינן רק "פעולות", הן **ממשקים קטנים**.

היכולת להעביר, לשמור ולהחזיר פונקציות
מאפשרת לבנות מערכות מודולריות, עם אחריות ברורה ויכולת הרחבה אינסופית.

פייתון נותנת לך את זה ישר מהקופסה
והיא עושה את זה בפשטות שמרבית השפות הוותיקות עדיין מקנאות בה.

## lambda פונקציות אנונימיות ושימוש עם map/filter

עד עכשיו כל פונקציה שהגדרנו קיבלה שם.
אבל לפעמים אתה צריך פונקציה קטנה, רגעית
כזו שנועדה לפעולה אחת בלבד,
ואין סיבה "להכביד" עליה בהגדרה מלאה.

כאן נכנסת לתמונה המילה הקטנה **lambda**
פונקציה אנונימית (כלומר בלי שם),
שאפשר להגדיר במקום שבו היא נדרשת.

## התחביר

```
lambda פרמטרים: ביטוי
```

הביטוי היחיד שאחרי הנקודתיים הוא מה שהפונקציה מחזירה.
אין צורך במילת מפתח return.

לדוגמה:

```
square = lambda x: x ** 2
print(square(5))  # 25
```

זה שווה ערך לכתיבה:

```
def square(x):
    return x ** 2
```

אבל הרבה יותר קצרה כשמדובר בפעולה פשוטה.

## שימושים קלאסיים

פונקציות lambda הופכות לחזקות במיוחד כשהן משולבות עם
כלים כמו map, filter ו-sorted,
פונקציות שמקבלות פונקציה אחרת כפרמטר.

```
numbers = [1, 2, 3, 4, 5]

# העלאה בריבוע של כל המספרים
squares = list(map(lambda x: x ** 2, numbers))

# סינון מספרים זוגיים
evens = list(filter(lambda x: x % 2 == 0, numbers))

print(squares)  # [1, 4, 9, 16, 25]
print(evens)    # [2, 4]
```

במקום להגדיר פונקציה חיצונית בשם,
אנחנו פשוט "שולפים" פונקציה קטנה לתוך השורה.

### שימוש עם sorted

פייתון מאפשרת גם למיין לפי ביטוי מסוים באמצעות key:

```
words = ["AI", "python", "Machine", "deep"]
words.sort(key=lambda w: len(w))
print(words)  # ['AI', 'deep', 'python', 'Machine']
```

ה-lambda כאן משמשת כתנאי מיון זמני,
בלי להגדיר פונקציה נפרדת.

### למה זה חשוב בפרויקטי AI

כשאתה עובד עם נתונים, מודלים או preprocessing,
יש אינספור מצבים שבהם אתה צריך פונקציה רגעית:
להמיר טקסטים, לנקות נתונים, לסנן תוצאות.

lambda מאפשרת לכתוב את זה **ישירות בהקשר**
כך שהקוד נשאר קומפקטי וברור.

לדוגמה:

```
texts = [" AI ", "Data ", " science  "]
cleaned = list(map(lambda t: t.strip().lower(), texts))
print(cleaned)  # ['ai', 'data', 'science']
```

זו בדיוק הגישה הפייתונית: לא לפזר פונקציות מיותרות, אלא לכתוב רק מה שצריך.

**טיפ הנדסי**

למרות הפיתוי, אל תשתמש ב-lambda כדי לכתוב לוגיקה מורכבת.
היא נועדה לפעולות קצרות וברורות בלבד.

כלל אצבע פשוט:

אם הפונקציה שלך לא נכנסת לשורה אחת, תן לה שם אמיתי.

lambda היא כמו ביטוי רגולרי: נהדרת כשמשתמשים בה במידה.

## Scope: LEGB (Local, Enclosing, Global, Built-in)

בכל פעם שפייתון רואה שם של משתנה,
היא צריכה לדעת מאיפה להביא את הערך שלו.
האם הוא הוגדר בתוך הפונקציה? מחוצה לה? אולי בכלל מילה שמורה של השפה?

כדי להחליט, פייתון עוברת בסדר קבוע
מנגנון חיפוש שנקרא **LEGB**:

**L – Local** (בתוך הפונקציה)
**E – Enclosing** (בתוך פונקציה עוטפת)

**G – Global** (ברמה הגלובלית של הקובץ)

**B - Built-in** (הגדרות של פייתון עצמה, כמו len או print)

## Local - תחום מקומי

זהו התחום הצר ביותר, משתנים שהוגדרו **בתוך פונקציה**.

```python
def show():
    msg = "Hello"
    return msg

# קריאה לפונקציה
print(show())  # Hello
```

**הסבר קצר:**

הפונקציה show() **לא** מדפיסה בעצמה, אלא *מחזירה* את הערך "Hello".

הקריאה ל-print(show()) מדפיסה את מה שהפונקציה החזירה.

זו דרך נקייה וברורה יותר לכתוב פונקציות.

הן עושות חישוב ומחזירות תוצאה,

וההדפסה מתבצעת רק מחוץ להן.

## Enclosing - תחום עוטף

כשיש פונקציות בתוך פונקציות,

הפונקציה הפנימית יכולה לגשת למשתנים של הפונקציה העוטפת.

```
def outer():
    name = "תמר"

    def inner():
        return "שלום " + name

    return inner()

# קריאה לפונקציה
print(outer())  # שלום תמר
```

הפונקציה `inner()` לא מגדירה את `name` בעצמה,
אלא משתמשת בערך שהוגדר ב-scope העוטף שלה (`outer`).

### משתנים גלובליים - Global

אם המשתנה לא נמצא לא בתחום המקומי (Local) ולא בתחום העוטף (Enclosing), פייתון תבדוק אם הוא הוגדר ברמה הגלובלית של הקובץ.

```
count = 0

def increment():
    global count
    count += 1

# קריאה לפונקציה
increment()
print(count)  # 1
```

המילה global אומרת לפייתון:
"אל תיצור משתנה חדש מקומי, השתמש בזה שמוגדר מחוץ לפונקציה."

## Built-in משתנים פנימיים של השפה

אם פייתון לא מוצאת את המשתנה באף אחד מהשלבים הקודמים, היא בודקת האם זהו שם שמובנה בשפה עצמה:

```
print(len("AI"))  # len הוא Built-in
```

מילים כמו range, sum, len, ו-print
הן חלק מה-Built-ins של פייתון.
אפשר "לדרוס" אותן בטעות, ולכן כדאי להימנע משמות כאלה בקוד שלך:

```
sum = 42  # המקורית לא זמינה sum עכשיו הפונקציה
```

## דוגמה שמאחדת הכול

```
x = "global"

def outer():
  x = "enclosing"
  def inner():
    x = "local"
    print(x)
```

```
  inner()

outer()
```

פלט:

```
local
```

אם נמחוק את ההגדרה המקומית (x = "local"),
פייתון תשתמש בזה מה-Enclosing.
אם גם זה לא קיים, היא תעבור ל-Global,
ואם גם שם לא תמצא, תבדוק ב-Built-ins.

זו בדיוק שרשרת ה-LEGB.

## טיפ הנדסי

הבנת Scope היא קריטית כשכותבים מערכות מורכבות במיוחד ב-AI, שם קוד רץ במקביל, בפונקציות פנימיות וב-callbacks.

זכור את הכלל הזה:

"השתמש בערכים מקומיים, העבר משתנים גלובליים כפרמטרים, ואל תיגע בפונקציות או באובייקטים מובנים (Built-ins), אלא אם אתה באמת יודע מה אתה עושה."

Scope ברור = קוד צפוי = פחות באגים.

## Docstrings תיעוד קוד מקצועי

כל מתכנת יודע שצריך "להשאיר הערות בקוד",
אבל מתכנתים מצוינים יודעים שצריך **לתעד את ההתנהגות, לא את התחביר**.

כאן נכנסים לתמונה **Docstrings**
תיעוד מובנה שמאפשר להבין מה הפונקציה עושה,
איך משתמשים בה, ומה היא מחזירה
ישירות מהקוד, בלי לפתוח קובץ נפרד.

## מה זה Docstring

Docstring הוא מחרוזת טקסט שנמצאת מיד אחרי הגדרת פונקציה, מחלקה או מודול.
הוא כתוב בין שלושה גרשיים (""") ומשמש כפירוט רשמי לתיעוד.

```
def add(a: int, b: int) -> int:
    """
    מחזיר את סכום שני המספרים.
    :param a: המספר הראשון
    :param b: המספר השני
    :return: סכום שני הערכים
    """
    return a + b
```

זה לא סתם הערה.

פייתון ממש **שומרת** את ה-Docstring כחלק מהאובייקט עצמו.

```
print(add.__doc__)
```

פלט:

```
מחזיר את סכום שני המספרים.
:param a: המספר הראשון
:param b: המספר השני
:return: סכום שני הערכים
```

כלומר, זהו תיעוד חי, לא רק טקסט.

**למה זה חשוב במערכות AI**

במערכות AI יש עשרות פונקציות קטנות
ניקוי, ניתוח, מדידה, ולוגים.
הן מתחברות אחת לשנייה כמו רכיבים במעגל חשמלי.
אם כל פונקציה מתועדת היטב,
אפשר להבין מה כל שלב עושה גם בלי לפתוח את הקוד הפנימי.

זה לא רק נוחות, זו **אחריות הנדסית**.
כשמישהו אחר (או אתה בעוד חודשיים) ניגש לקוד,
ה-Docstring הוא ההבדל בין "איך זה עובד?" לבין "מדהים, זה ברור!".

## תקני תיעוד נפוצים

יש מספר סגנונות מקובלים. הנה שניים עיקריים:

### 1. סגנון Google

```
def load_dataset(path: str) -> list[str]:
    """
    טוען קובץ טקסט ומחזיר רשימת שורות.

    Args:
        path (str): הנתיב לקובץ.

    Returns:
        list[str]: רשימת שורות מהקובץ.
    """
```

### 2. סגנון reStructuredText (בשימוש נרחב ב-Sphinx)

```
def tokenize(text: str) -> list[str]:
    """
    מבצע פיצול טקסט למילים.

    :param text: מחרוזת לניתוח
    :type text: str
    :return: רשימת מילים
    :rtype: list[str]
    """
```

אין סגנון אחד "נכון"
העיקר שתהיה עקביות בכל הפרויקט.

## תיעוד גם לפונקציות פנימיות

גם פונקציות עזר קצרות ראויות ל-Docstring.
לא בגלל שמישהו אחר יקרא אותן,
אלא כדי לעזור לך להבין מה חשבת כשכתבת אותן.

```
def normalize(text: str) -> str:
    """מסיר רווחים מיותרים וממיר לאותיות קטנות."""
    return text.strip().lower()
```

שורה אחת מספיקה כשאין מורכבות.

## טיפ הנדסי

הכלל פשוט: "אם פונקציה שווה בשביל לכתוב אותה, היא שווה בשביל לתעד אותה."

Docstrings אינם קישוט, הם חוזה.
הם מגדירים מה הפונקציה מבטיחה לעשות ומה לא.
וכשכותבים קוד שצריך לעבור ביקורת, תחזוקה, או עבודה בצוות
זה ההבדל בין קוד "שנשבר" לקוד שחי שנים.

## Best Practices שמות, פיצול, אחריות יחידה

אחרי שהבנו איך פונקציות נוצרות, נקראות, מחזירות ערכים ומטפלות בפרמטרים

נשארה השאלה החשובה באמת:

**איך לכתוב פונקציות שנשארות קריאות וברורות גם עוד חצי שנה?**

## שמות: הפונקציה מדברת בעד עצמה

שמות פונקציות טובים הם כמו תיעוד חכם

אם אתה צריך לקרוא את ה-Docstring כדי להבין מה הפונקציה עושה,

כנראה שהשם שלה לא מספיק טוב.

## כללים פשוטים:

- שם פונקציה = **פועל** + תיאור הפעולה.
לדוגמה: `calculate_accuracy`, `clean_text`, `load_data`.
- לא לקצר מילים סתם: `calc_acc` אולי חוסך תווים, אבל גוזל קריאות.
- עדיף להיות ברור מאשר מתוחכם.

- אל תשתמש באותיות בודדות (כמו p, f או x) כשמות משתנים, אלא אם מדובר בהקשר מתמטי מובהק.

שם טוב הוא חוזה: הוא מבטיח מה שהפונקציה עושה, ולא יותר.

## פונקציה = פעולה אחת בלבד

אם פונקציה עושה יותר מדי, היא מאבדת את היכולת להיבדק, להבין ולתחזק.
העיקרון הזה נקרא: **Single Responsibility Principle (SRP)**
עיקרון הליבה של כל תכנות מודרני.

כל פונקציה צריכה:

1. לעשות דבר אחד בלבד.
2. לעשות אותו היטב.
3. להיקרא בהתאם למה שהיא עושה.

```
def load_and_clean_data(path):  # ❌ עושה יותר מדי
    ...
```

עדיף:

```
def load_data(path):
    ...
def clean_data(data):
    ...
```

זה אולי יותר שורות קוד, אבל הרבה פחות כאב ראש.

## פיצול חכם עדיף על אופטימיזציה מוקדמת

מתכנתים מנוסים נוטים "לדחוס" קוד כדי לשפר ביצועים.
אבל כמעט תמיד קריאות חשובה יותר ממהירות.
אם פונקציה נהיית ארוכה מדי, זה סימן שהיא צריכה להתפצל.

פייתון מעודדת קריאות על פני תחכום.
ולכן משפט אחד נכון שווה יותר מעשרה "טריקים פייתוניים".

## פונקציות קצרות לא חייבות להיות טיפשיות

יש מי שחושב שפונקציות קצרות = פונקציות "לא שוות".
בפועל, ההפך הוא הנכון:
פונקציות קטנות מאפשרות:

- בדיקות ממוקדות (unit tests)
- שימוש חוזר בקוד
- לוגים מדויקים
- וניטור מדויק של תקלות

אין גבול תחתון לאורך פונקציה,
אבל כלל אצבע: אם צריך לגלול כדי להבין אותה, היא ארוכה מדי.

## טיפ הנדסי

הבדל קטן בין מתכנת טוב למהנדס מצוין הוא **תחושת הסדר**.
מהנדס לא רק כותב קוד שעובד, אלא קוד שקל להבין, לבדוק ולשפר.

זכור את שלושת ה-S:

1. **Simple:** פשוט להבנה.
2. **Specific:** עושה דבר אחד ברור.
3. **Self-describing:** מדברת בעד עצמה.

פונקציה טובה היא כמו משפט טוב
לא צריך להסביר אותה פעמיים.

## דוגמה מרכזית: utility functions לעיבוד טקסט

בפרויקטים של בינה מלאכותית, עיבוד טקסט הוא אחד השלבים הבסיסיים והנפוצים ביותר.
לפני שמודל לומד משהו, מישהו צריך לנקות את הנתונים, להפריד מילים, ולמדוד תוצאות.

במקום קוד מפוזר, נהוג לרכז פונקציות עזר (utilities) שמבצעות פעולות קטנות, עקביות וברורות.

נבנה כאן גרסה פשוטה של **text_utils.py**,
שתוכל להשתלב אחר כך בפרויקט ה-mini_text_analyzer שלך.

### normalize ניקוי בסיסי

```
def normalize(text: str) -> str:
    """
    מנקה טקסט מרווחים מיותרים וממיר לאותיות קטנות.
    """
    return text.strip().lower()
```

שורה אחת, אבל חשובה:
היא מבטיחה שכל שלב אחר יעבוד על נתונים עקביים.
גם מודלים מתקדמים ייכשלו אם הקלט לא יישר קו.

### tokenize פיצול למילים

```
import re

def tokenize(text: str) -> list[str]:
    """
    מפצל טקסט למילים תוך הסרת סימני פיסוק.
    """

    text = re.sub(r"[^\w\s-]", " ", text)  # משאיר מקף
    return [w for w in text.split() if w]
```

זו דרך פשוטה ויעילה לבצע tokenization ראשוני לא מושלם כמו מודלי NLP, אבל מספיק לרוב התרחישים המוקדמים.

## חישוב מדדים בסיסיים  word_stats

```
from collections import Counter

def word_stats(words: list[str]) -> dict[str, int | str | float]:
    """
    מחשב מדדים בסיסיים על רשימת מילים.
    """
    total = len(words)
    if total == 0:
        return {"num_words": 0, "avg_length": 0, "most_common":
""}
    avg_length = sum(len(w) for w in words) / total
    most_common = Counter(words).most_common(1)[0][0]
    return {
        "num_words": total,
        "avg_length": round(avg_length, 2),
        "most_common": most_common
    }
```

כאן אנחנו משלבים כמה עקרונות:

- פיצול אחריות: פונקציה אחת מחשבת, אחרת מנתחת.
- שימוש ב-Counter במקום כתיבת לולאה.
- החזרת מילון ברור עם שדות קבועים, נוח לכתיבה לקובץ או לוג.

## הרכבה של מספר פונקציות  analyze_text

```
def analyze_text(text: str) -> dict[str, int | str | float]:
```

```
    """
    מפעיל את כל שלבי הניתוח על טקסט גולמי.
    """
    clean = normalize(text)
    words = tokenize(clean)
    return word_stats(words)
```

זו כבר פונקציה “על” שמדגימה **הרכבה נכונה של פונקציות קטנות**.

היא עושה דבר אחד בלבד

קוראת לפונקציות אחרות במבנה ברור, בלי להתעסק בפרטים שלהן.

## דוגמת הרצה

```
if __name__ == "__main__":
    sample = "פייתון היא שפה קלה, מהירה ונוחה ה-AI. פייתון מדהים של פייתון."
    print(analyze_text(sample))
```

פלט לדוגמה:

```
{'num_words': 8, 'avg_length': 4.5, 'most_common': 'פייתון'}
```

## טיפ הנדסי

שים לב כמה הקוד קריא:

- כל פונקציה עושה פעולה אחת בלבד.
- השמות שלהן ברורים ומדברים בעד עצמם.
- אפשר לבדוק כל פונקציה בנפרד (unit test).
- והכול מתחבר לפונקציה אחת פשוטה: analyze_text.

כך בונים **תשתית פונקציונלית אמינה**
שאפשר להרחיב, לעטוף בלוגים, להוסיף מדידות,
ובסוף גם לשלב במערכת אמיתית לעיבוד שפה.

## סיכום: מה מתכנת מנוסה צריך לזכור

פונקציות הן הלב הפועם של פייתון.
הן לא רק מקצרות קוד, אלא **מייצרות ארכיטקטורה**.
כל פונקציה שאתה כותב היא לבנה קטנה במכונה גדולה
וכשבונות נכון את הלבנים האלה, כל המערכת נראית אחרת לגמרי.

## עיקרי הדברים שחשוב לזכור

### קלט ברור, פלט צפוי

פונקציה טובה מתנהגת כמו חוזה.

אם היא מקבלת ערכים, היא לא משנה אותם בחוץ.
אם היא מחזירה ערכים, הם תמיד באותו מבנה.

## אל תשתמש בערכי ברירת מחדל Mutable

זוהי אחת המלכודות הכי ותיקות בפייתון.
תמיד העדף None והתחל ערכים מחדש בתוך הפונקציה.

## תעד הכל עם Docstrings

לא בשביל "הבודק", אלא בשביל עצמך בעוד חצי שנה.
ה-Docstring הוא ההסבר שאתה כותב למי שיבוא אחריך, גם אם זה אתה.

## פונקציה = פעולה אחת בלבד

אם פונקציה עושה יותר מדבר אחד
חלק אותה.
היא תישאר קריאה, נבדקת ונשלטת.

## פונקציות הן אובייקטים

תוכל להעביר אותן, לשמור אותן, וליצור מהן פונקציות חדשות.
זה כלי עוצמתי! השתמש בו באחריות, לא מתוך גימיק.

## קוד טוב הוא קוד פשוט

אם אתה מתלבט אם לקצר או להשאיר קריאות.

תמיד בחר בקריאות.
מכונות קוראות מהר, אבל בני אדם מתחזקים קוד שנים.

## סיכום תמציתי

| עיקרון | מטרה |
|---|---|
| **פשטות** | קריאות והבנה מיידית |
| **אחריות יחידה** | תחזוקה ובדיקות קלות |
| **תיעוד** | שיתוף ידע ושימוש חוזר |
| **חזרתיות נמוכה** | פחות שגיאות |
| **גמישות מבוקרת** | הרחבה בלי לשבור |

## מילה אישית לסיום

במערכות AI, הפונקציות שלך הן ה-DNA של המערכת.
הן אלו שמחברות בין הנתונים, האלגוריתמים וההיגיון העסקי.
ככל שתכתוב פונקציות ברורות, קטנות ומתועדות היטב
כך תוכל לגדול מהר יותר בלי לאבד שליטה.

פונקציה טובה היא לא רק מה שרץ
היא מה שאתה שמח לקרוא גם אחרי שנה.

**מאת: תומר קדם**

# פרק 5 – מודולים, חבילות וארגון פרויקט

## למה לא קובץ אחד גדול

כל מתכנת מתחיל את דרכו עם קובץ יחיד main.py, לפעמים app.py, ובימים עמוסים במיוחד אפילו script_final_v2_fixed.py.
זה עובד מצוין כל עוד מדובר בניסוי קטן.

אבל אז זה קורה: הקובץ גדל למאות שורות, הפונקציות מסתבכות, ואתה כבר לא בטוח איפה נמצאת הפונקציה שמחשבת את ה-accuracy.

בנקודה הזו אתה מגלה את ההבדל בין קוד שעובד, לבין מערכת שניתנת לניהול.
כשהכול נמצא בקובץ אחד:

- אין הפרדה בין שלבים שונים בקוד.
- כל שינוי קטן עלול לשבור אזורים אחרים.
- אי אפשר לבדוק רכיב אחד בלי להריץ את הכול.
- ובעיקר, קשה מאוד לעבוד בצוות.

הפתרון הוא לא "פחות קוד", אלא **קוד מחולק נכון**.
זו בדיוק הסיבה שפייתון בנויה סביב מודולים וחבילות וזו הדרך שלה לארגן מחשבה הנדסית.

## Imports בסיסיים (import, from, alias)

**מודול** הוא פשוט קובץ פייתון (.py) שמכיל קוד.

פונקציות, מחלקות, משתנים או קבועים שאפשר להשתמש בהם גם ממקומות אחרים בקוד שלך. ובמקום לכתוב את הכול שוב ושוב בכל קובץ, אתה **מייבא** את מה שאתה צריך.
זה מה שהופך את פייתון לשפה כל כך נוחה לבניית מערכות גדולות: כל קובץ הוא יחידה עצמאית שאפשר לשתף, לבדוק ולהרכיב ממנה מערכת שלמה.

### import

ייבוא של מודול שלם:

```
import math

print(math.sqrt(16))  # 4.0
```

כאן אתה אומר לפייתון: "תטען את הקובץ math.py (מהספרייה הסטנדרטית), ואני אשתמש בפונקציות שלו דרך השם math."

### from ... import

ייבוא של חלק מסוים בלבד:

```
from math import sqrt

print(sqrt(25))  # 5.0
```

חוסך הקלדה, אבל עדיף להשתמש בזה רק כשבאמת יש צורך.
כדי לשמור על קריאות ולדעת מאיפה הגיע כל שם.

### שם מקוצר - alias

כמעט כל מתכנת מכיר את זה:

```
import numpy as np
import pandas as pd
```

זהו קיצור מקובל שמקל על הקריאה והופך את הקוד לאחיד בין צוותים.

### למה זה חשוב בפרויקטי AI

במערכות עיבוד נתונים, יש עשרות מודולים קטנים:
ניקוי טקסט, קריאה ממקורות, חישוב מדדים, שמירת תוצאות, לוגים, ועוד.
מנגנון import מאפשר **להרכיב מהם מערכת אחת נקייה**, בלי כפילויות או תלות הדדית מיותרת.

```
from text.cleaner import normalize
from text.tokenizer import tokenize
from text.stats import word_stats

def analyze(text):
    return word_stats(tokenize(normalize(text)))
```

כל שורה ברורה, כל רכיב ממוקד. והכול משתלב בהרמוניה.

## מבנה תיקיות מומלץ לפרויקט Production

כשהפרויקט גדל, חשוב לדעת איפה כל דבר ממוקם.
קוד נקי מתחיל ממבנה תיקיות הגיוני.
כזה שקל להבין גם חצי שנה אחרי שכתבת אותו.

### package – איך פייתון מזהה חבילה

חבילה (package) היא פשוט תיקייה שיש בה קובץ בשם `__init__.py`

הקובץ הזה אומר לפייתון: "זו חבילה, לא סתם תיקייה."
בתוכו אפשר להגדיר אילו מודולים יהיו זמינים למי שמייבא את החבילה.

### מבנה מומלץ לפרויקט

זה מבנה פרויקט בסיסי שמתאים גם לפרודקשן:

```
my_project/
 ├── src/
 │    └── my_package/
 │        ├── __init__.py
 │        ├── core.py
 │        └── utils.py
 ├── scripts/
 │    └── run_demo.py
 ├── tests/
 │    └── test_core.py
 └── requirements.txt
```

בתיקייה src נמצא קוד הספרייה שלך.
בתוך scripts תשמור קבצי הרצה או דוגמאות,
וב-tests, בדיקות יחידה.

## דוגמת קבצים קצרה

src/my_package/core.py

```
def tokenize(text: str) -> list[str]:
    return text.split()

def count_tokens(text: str) -> int:
    return len(tokenize(text))
```

src/my_package/utils.py

```
def normalize(text: str) -> str:
    return " ".join(text.split()).strip()
```

src/my_package/__init__.py

```
# מה ייחשף כשעושים import my_package
from .core import tokenize, count_tokens
from .utils import normalize

__all__ = ["tokenize", "count_tokens", "normalize"]
__version__ = "0.1.0"
```

## קובץ הרצה חיצוני

`scripts/run.py`

```
import sys
sys.path.append("src")  # בסביבת פיתוח פשוטה. בהתקנה אמיתית
לא צריך את זה.

import my_package as mp

text = "hello   world"
print(mp.normalize(text))      # hello world
print(mp.tokenize(text))       # ['hello', 'world']
print(mp.count_tokens(text))    # 2
```

## למה זו הגישה הנכונה בפרויקטים גדולים

- הקוד שלך מופרד מהרצה, מבדיקות ומהתלויות.
- אפשר להוסיף מודולים חדשים בלי לשנות קוד קיים.
- ייבוא עובד בצורה אחידה וברורה.
- מבנה `/src` מונע התנגשויות בין קבצים מקומיים לחבילות חיצוניות.

## Imports יחסיים מול מוחלטים

כשפרויקט מתחיל לגדול, אתה כבר לא מייבא רק מתוך ספריות סטנדרטיות, אלא גם בין מודולים שכתבת בעצמך.
כאן חשוב להבין את ההבדל בין **ייבוא מוחלט** ל-**ייבוא יחסי**.

### ייבוא מוחלט (Absolute Import)

זו הדרך הברורה והעדיפה ברוב המקרים:
פשוט לייבא לפי שם החבילה המלא מהשורש של הפרויקט.

```
# מתוך src/my_package/text/cleaner.py
from my_package.utils import normalize
```

ייבוא מוחלט ברור לכל מי שקורא את הקוד, גם מחוץ לפרויקט.
הוא עובד מצוין כשיש לך סביבת הרצה יציבה (כמו התקנה ב-venv או מבנה /src מסודר).

### ייבוא יחסי (Relative Import)

שימושי כשאתה עובד בתוך אותה חבילה ומעדיף לקצר כתיבה:

```
# מתוך src/my_package/text/tokenizer.py
from ..utils import normalize
```

שני הנקודות (`..`) אומרות "עלה תיקייה אחת למעלה."
אפשר להשתמש גם ב-`.` (תיקייה נוכחית) או ביותר מנקודה אחת לפי הצורך.

## אז מתי להשתמש במה?

- **בפרויקטים קטנים או בסקריפטים פנימיים:**
  אפשר להסתפק בייבוא יחסי.
- **בפרויקטים גדולים, חבילות או קוד פתוח:**
  עדיף תמיד ייבוא מוחלט.

ייבוא מוחלט מקל על קריאות, בדיקות ותחזוקה,
בעוד שייבוא יחסי מתאים בעיקר לשלב הפיתוח המוקדם כשהכול עדיין בתיקייה אחת.

## Docstrings למודולים: תיעוד ברמת הקובץ

מטרת ה-docstring ברמת מודול היא לתת לקורא שלך כיוון מיידי: מה הקובץ עושה, איך משתמשים בו, ואיזה חלקים נחשבים API ציבורי.

### איך זה נראה במודול אמיתי

```
"""
text tools
כלי עזר לעיבוד טקסט: ניקוי, טוקניזציה וספירת טוקנים.

שימוש בסיסי:
  from my_package.text_tools import normalize, tokenize,
count_tokens
  s = normalize(" Hello   world ")
  words = tokenize(s)        # ['Hello', 'world']
```

```
    n = count_tokens(s)        # 2

מודולים קשורים: my_package.utils
"""

from __future__ import annotations

__all__ = ["normalize", "tokenize", "count_tokens"]
__version__ = "0.2.0"

def normalize(text: str) -> str:
    """מחזיר טקסט מרווח ונקי."""
    return " ".join(text.split()).strip()

def tokenize(text: str) -> list[str]:
    """מפצל טקסט למילים על בסיס רווחים."""
    return text.split()

def count_tokens(text: str) -> int:
    """סופר כמה טוקנים יש בטקסט אחרי ניקוי."""
    return len(tokenize(normalize(text)))
```

מה חשוב לשים בדוקסטרינג של מודול:

- תיאור קצר וברור של מטרת המודול.
- דוגמת שימוש של שתי שורות שמראה את ה-import והקריאה לפונקציות.

- אזכור מודולים קשורים אם יש.
- אם יש API ציבורי, סמן גם ב-`__all__` כדי להבהיר מה חשוף.

## איך קוראים את זה בזמן אמת

```
import my_package.text_tools as tt

print(tt.__doc__[:120], "...")  # מציג את תחילת הדוקסטרינג של המודול
help(tt)                        # תצוגה מלאה עם פונקציות והסברים
```

## סגנון תיעוד קצר ואחיד

בחר סגנון אחד לפונקציות ולמחלקות והיצמד אליו:

- תיאור במשפט אחד.
- פרמטרים עיקריים ופלט בשורה או שתיים.
- אם יש התנהגות מיוחדת, שורה קצרה על חריגים או קצה.

למשל, סגנון תמציתי:

```
def summarize(text: str, max_tokens: int = 64) -> str:
    """
    יוצר תקציר קצר לטקסט.

    text: הטקסט המקורי.
    max_tokens: אורך תקציר מרבי.
    מחזיר: מחרוזת עם תקציר.
    """
    ...
```

טיפ קטן לפרויקטים גדולים:

- שמרו דוקסטרינגים קצרים. פרטים ארוכים עוברים ל-README או למסמך API נפרד.
- עדכנו דוגמת שימוש כשמשנים חתימה. דוגמה לא מעודכנת מבלבלת יותר מחוסר דוגמה.

## Best Practices: שמות, אחריות וסדר imports

ככל שהפרויקט גדל, סדר וקריאות הופכים לא פחות חשובים מביצועים.

### שמות ברורים

- קובץ או מודול:
  בשם קטן וברור, `text_utils.py`, לא `myTextFunctions.py`.
- פונקציות:
  פועל שמתאר פעולה, כמו `normalize_text` או `count_tokens`.
- משתנים:
  קצרים אבל משמעותיים `config`, `tokens`, `text`.
- קבועים:
  באותיות גדולות (`UPPER_CASE`).

## אחריות אחת לכל מודול

כל קובץ אמור לעשות דבר אחד ברור.
אם אתה מוצא את עצמך גולל 300 שורות שמערבבות לוגיקות שונות.

זה סימן לשהגיע הזמן לפיצול.

```
# text_clean.py
def normalize(text: str) -> str:
    return " ".join(text.split()).strip()

# text_tokenize.py
from .text_clean import normalize

def tokenize(text: str) -> list[str]:
    return normalize(text).split()
```

## סדר imports

סדר קבוע הופך את הקוד למובן גם בלי לחשוב.

1. ספריות סטנדרטיות
2. ספריות צד שלישי
3. מודולים פנימיים שלך

```
from pathlib import Path
```

```
import pandas as pd
from my_package.text_tokenize import tokenize
```

## ממשק ציבורי ברור

אם זו חבילה, חשוב להגדיר מה נחשף החוצה.

```
# __init__.py
from .text_clean import normalize
from .text_tokenize import tokenize

__all__ = ["normalize", "tokenize"]
```

## כלל אצבע פשוט

אם שם הקובץ מסביר את מטרתו,
אם אתה יכול למחוק פונקציה בלי לשבור את השאר,
ואם ה-imports נקיים וברורים

אז אתה כבר **עובד נכון**.

## הפרדה בין מודול להרצה: `__name__ == "__main__"`

בפייתון, כל קובץ הוא גם **מודול** וגם **תוכנית בפני עצמה**.
כשאתה מריץ קובץ ישירות, משתנה פנימי בשם `__name__` מקבל

את הערך "__main__".
אבל כשקובץ מיובא כמודול ממקום אחר, הוא יקבל את שמו האמיתי, לדוגמה "text_utils".

וזה בדיוק מה שמאפשר להפריד בין **קוד להרצה** לבין **קוד לשימוש חוזר**.

### דוגמה פשוטה

```
# text_utils.py
def normalize(text: str) -> str:
    return " ".join(text.split()).strip()

if __name__ == "__main__":
    sample = "   שלום   עולם   "
    print(normalize(sample))  # שלום עולם
```

כשאתה מריץ את הקובץ ישירות (python text_utils.py),
פייתון תבצע גם את החלק שבתוך ה-if.
אבל אם תייבא את הקובץ ממקום אחר:

```
from text_utils import normalize
```

הקטע שבתוך ה-if **לא ירוץ** בכלל.

## למה זה חשוב

כי ככה אתה יכול לבדוק קובץ בעצמך,
מבלי שהוא יפריע לקוד שמייבא אותו אחר כך.
זה אחד הטריקים הכי פשוטים שהופכים סקריפט לספרייה אמיתית.

## טיפ קטן

אם יש לך מודול עם קוד בדיקה פנימי, השאר אותו תמיד תחת `if __name__ == "__main__":`
ולא סתם בסוף הקובץ.
כך הוא נשאר להרצה עצמאית בלי להשפיע על שאר המערכת.

## Utility modules: איחוד פונקציות עזר

מטרת מודול עזר היא לרכז פונקציות קטנות שחוזרות על עצמן, בלי להפוך לפח אשפה של הפרויקט.

## מתי ליצור מודול עזר

- כשרואים את אותה פעולה בקבצים שונים של הפרויקט.
- כשהפונקציות קצרות, טהורות, ולא תלויות בהקשר ספציפי.
- כשהן שימושיות בכמה מודולים שונים.

## איך לארגן

עדיף להחזיק כמה מודולי עזר קטנים לפי תחום, ולא קובץ ענק בשם utils.py.

- `string_utils.py` לטקסט
- `io_utils.py` לקריאה וכתיבה
- `time_utils.py` לזמנים ותאריכים

### דוגמה קצרה

`src/my_package/string_utils.py`

```
def normalize_spaces(s: str) -> str:
    return " ".join(s.split()).strip()

def safe_lower(s: str | None) -> str:
    return (s or "").lower()
```

שימוש מתוך מודול אחר:

```
# src/my_package/text/cleaner.py
from my_package.string_utils import normalize_spaces,
safe_lower

def normalize(text: str) -> str:
    text = normalize_spaces(text)
    return safe_lower(text)
```

## כללי עבודה פשוטים

- פונקציות עזר עדיף שיהיו טהורות. קלט בפנים, פלט החוצה.
- אם פונקציה נוגעת בקבצים או בסביבה, ציינו זאת בשם או בדוקסטרינג.
- אם מודול עזר גדל מדי, פצלו אותו לפי תחומים. קל יותר לתחזק ולייבא.

## בדיקה מהירה

אם קשה לתת שם ברור למודול העזר, או שהוא מתחיל להכיל את הכול מהכול, זה סימן לפיצול.

## דוגמה מרכזית: פרויקט mini_text_analyzer מחולק למודולים

אחרי שהבנו איך מחלקים קוד למודולים וחבילות, הגיע הזמן לראות איך זה נראה בפרויקט אמיתי.

הדוגמה הבאה מציגה גרסה פשוטה של כלי לעיבוד טקסטים **mini_text**.

הרעיון הוא לא רק לפצל קבצים, אלא לבנות מבנה שמאפשר להתרחב בלי לגעת בלוגיקה קיימת.

```
mini_text/
 ├── src/
 │    └── mini_text/
```

```
│       ├── __init__.py
│       ├── clean.py
│       ├── tokenize.py
│       └── stats.py
└── scripts/
    └── run_demo.py
```

## clean.py

```python
def normalize(text: str) -> str:
    """מסיר רווחים מיותרים וניקוי בסיסי."""
    return " ".join(text.split()).strip()
```

## tokenize.py

```python
from mini_text.clean import normalize

def tokenize(text: str) -> list[str]:
    """מפצל טקסט למילים אחרי ניקוי."""
    return normalize(text).split()
```

## stats.py

```python
from mini_text.tokenize import tokenize

def count_tokens(text: str) -> int:
    """סופר את מספר המילים בטקסט."""
    return len(tokenize(text))
```

## init.py

```
from .clean import normalize
from .tokenize import tokenize
from .stats import count_tokens

__all__ = ["normalize", "tokenize", "count_tokens"]
```

## run_demo.py

```
import sys
sys.path.append("src")

from mini_text import normalize, tokenize, count_tokens

text = "  שלום-AI   לעידן   ה "
print(normalize(text))       # שלום-AI לעידן ה
print(tokenize(text))        # ['שלום-AI', 'לעידן', 'ה']
print(count_tokens(text))    # 3
```

## למה זה עובד טוב

- כל קובץ מטפל בנושא אחד בלבד.
- אין תלות מעגלית – כל מודול יודע בדיוק על מי הוא נשען.
- אפשר להוסיף פונקציה חדשה (למשל detect_language) בלי לגעת בקוד קיים.
- קריא גם למי שנכנס לפרויקט בפעם הראשונה.

מאת: תומר קדם

## סיכום: איך ארגון נכון מקל על הרחבה

כשפרויקט **באמת מסודר**, כל שינוי קטן הוא לא מלחמה. אתה פשוט יודע איפה לגעת ומה להשאיר בשקט.
זה ההבדל בין קוד שנשען על קורי עכביש לבין מערכת שאפשר **לסמוך עליה**.
אז מה לקחת מהפרק?

- אל תשאיר את הכול בקובץ אחד. כל **מודול מטפל בנושא אחד ברור**.
- השתמש ב-**imports מוחלטים** כברירת מחדל. זה הופך את הקוד ליציב וקל לקריאה.
- שמור על **מבנה תיקיות קבוע** פרויקטים מסודרים מתחילים ב-/src.
- כתוב **docstring קצר וברור** בתחילת כל קובץ. משפט אחד שמסביר מה הוא עושה מספיק.
- **שמות פשוטים, פונקציות מדויקות, אחריות אחת לכל קובץ.**
- הפרד בין ספרייה להרצה בעזרת `if __name__ == "__main__":`
- רכז פונקציות כלליות במודולי עזר קטנים, לא בקובץ "ענק, ואת הכול".

ולפני שאתה עושה merge או שולח PR, תעבור בראש על שלושת השאלות האלו:

1. אני מבין מיד **מה כל קובץ עושה?**
2. אני יודע **מאיפה כל import מגיע?**
3. אם אמחק פונקציה, ברור לי **מה יישבר?**

אם ענית "כן" על שלושתן.

הקוד שלך כבר נראה כמו של מתכנת שמבין **הנדסה, לא רק סינטקס**.

הרחבות חדשות ייכנסו חלק, באגים יהיו קלים יותר לאיתור, והקוד שלך יהיה נעים גם לעיניים של מי שיבוא אחריך.

# פרק 6 – סביבות עבודה ותלויות

## למה סביבה וירטואלית חיונית

פייתון היא שפה עם אקו-סיסטם עצום.
פרויקט אחד דורש numpy ו-pandas, אחר רוצה tensorflow,
ושלישי מתעקש על גרסה ישנה של fastapi.

כשאתה מתקין את הכול על אותה מערכת, הספריות מתחילות לריב ביניהן.
מה שעבד אתמול. היום קורס, וכל ניסיון לשחזר גרסאות הופך לסיוט.

כאן נכנסת הסביבה הווירטואלית (Virtual Environment).
היא יוצרת "בועה" קטנה וסגורה,
שבה מותקנות רק החבילות שהפרויקט שלך באמת צריך.

אפשר לחשוב עליה כעל **מיכל מבודד**
כל פרויקט חי בעולם משלו, בלי לגעת במערכת הראשית ובלי להפריע לאחרים.

בפרויקט AI זה קריטי במיוחד,
כי חבילות כמו torch, transformers או openai תלויות בגרסאות מדויקות מאוד.
אם תערבב ביניהן. שום מודל לא ירוץ כמו שצריך.

לכן, **כלל הברזל:** "לפני שאתה כותב שורת קוד אחת, תיצור סביבה."

## venv: יצירה והפעלה (Windows / Linux / Mac)

ברירת המחדל של כל פרויקט פייתון רציני, היא להתחיל בסביבה נקייה.

הדרך הפשוטה ביותר לעשות זאת היא בעזרת **venv**,

כלי שמובנה בפייתון כברירת מחדל, בלי צורך בהתקנה נוספת.

### יצירת סביבה וירטואלית

בתוך תיקיית הפרויקט שלך (למשל mini_text_analyzer), הרץ:

```
python -m venv .venv
```

זה ייצור תיקייה בשם venv. ובה כל מה שצריך:

- עותק מבודד של פייתון,
- תיקיית ספריות (site-packages),
- וסקריפט הפעלה.

### הפעלה

**Windows (PowerShell):**

```
.\.venv\Scripts\activate
```

**Mac / Linux:**

```
source .venv/bin/activate
```

לאחר ההפעלה, תראה בתחילת השורה את שם הסביבה:

```
(.venv) D:\Projects\mini_text_analyzer>
```

מכאן, כל חבילה שתתקין תישמר בתוך venv. בלבד, לא תשפיע על מערכת ההפעלה שלך.

## יציאה מהסביבה

כדי לחזור למצב רגיל:

```
deactivate
```

## טיפ קטן

אם אתה משתמש ב-VS Code, ברגע שתזהה את venv.,
העורך יציע אוטומטית לבחור בה כפייתון הפעיל שלך.
לחץ על "Select Interpreter", בחר את venv., וזהו
העורך, המסוף וה-IntelliSense יפעלו בסביבה הנכונה.

## pip: התקנת חבילות ו-requirements.txt

ברגע שהסביבה הווירטואלית פעילה, אפשר להתחיל להכניס לתוכה את כל מה שהפרויקט שלך צריך.
הכלי שמנהל את זה הוא **pip**, מנהל החבילות הרשמי של פייתון.

## התקנת חבילות

נניח שאתה רוצה להשתמש ב-NumPy וב-FastAPI:

```
pip install numpy fastapi
```

pip יוריד את הגרסאות האחרונות של הספריות האלו מהמאגר הרשמי (PyPI) וישמור אותן בתוך הסביבה שלך (venv.).

## בדיקת מה מותקן

```
pip list
```

תקבל רשימה של כל הספריות והגרסאות שהותקנו.
זה בדיוק המידע שתרצה לשתף עם חבר צוות או לשחזר במחשב אחר.

## שמירת התלויות בקובץ

כדי לתעד את כל מה שהתקנת, צור קובץ בשם requirements.txt:

```
pip freeze > requirements.txt
```

תוכן הקובץ ייראה כך:

```
fastapi==0.115.0
numpy==2.1.1
```

זהו צילום מצב מדויק של הסביבה שלך.

### התקנה מסביבה קיימת

אם מישהו שולח לך פרויקט עם קובץ כזה, תוכל לשחזר אותו בפקודה אחת:

```
pip install -r requirements.txt
```

כל הספריות הנכונות, בדיוק באותן גרסאות, יותקנו לתוך ה-venv שלך.

### טיפ מעשי

אל תערוך ידנית את `requirements.txt`.
הוא נוצר אוטומטית מ-pip freeze, וכך שומרים על עקביות בין מפתחים ומכונות שונות.

### Poetry: הכלי המודרני לניהול תלויות ו-סביבות

בעשור האחרון, קהילת פייתון חיפשה פתרון אלגנטי יותר לניהול תלויות. השילוב של `pip` עם `requirements.txt` עובד, אבל הוא גולמי: הוא לא יודע לנהל גרסאות חכמות, לא עוזר בארגון הפרויקט, ולא נוח במיוחד כשעובדים בצוותים גדולים.

כאן נכנס **Poetry**, כלי מודרני שמטפל בכל מחזור החיים של פרויקט פייתון:

- יצירת סביבה וירטואלית.
- ניהול תלויות עם פתרון קונפליקטים אוטומטי.
- נעילת גרסאות לשחזור יציב.
- בניית חבילות להפצה.

**Poetry** מרכז במקום אחד את כל מה שקשור לתלויות, גרסאות, סביבות, חבילה ופקודות ריצה.

הוא מחליף שימוש מפוזר ב-venv + pip + requirements.txt בקובץ יחיד בשם pyproject.toml וקובץ נעילה poetry.lock.

## התקנה מהירה

```
# Windows (PowerShell)
(Invoke-WebRequest -Uri https://install.python-poetry.org -UseBasicParsing).Content | py -

# Mac/Linux
curl -sSL https://install.python-poetry.org | python3 -
```

בדיקה:

```
poetry --version
```

יצירת פרויקט או אימוץ פרויקט קיים

```
# בתוך תיקיית הפרויקט
poetry init    # עונה על שאלות ומייצר pyproject.toml
# או, להוסיף תלות ראשונה וזה ייצור קובץ לבד:
poetry add requests
```

Poetry ייצור גם סביבה וירטואלית אוטומטית עבור הפרויקט.

לראות איזו:

```
poetry env info
```

לבחור גרסת פייתון ספציפית:

```
poetry env use python3.12
```

התקנת תלויות והפעלת הסביבה

```
poetry install   # מתקין הכל לפי pyproject.toml ו-poetry.lock
poetry shell     # נכנס לסביבה הווירטואלית של הפרויקט
```

אפשר גם להריץ פקודות בלי להיכנס ל-shell:

```
poetry run python -m mini_text_analyzer
```

הוספה והסרה של חבילות

```
poetry add fastapi "numpy>=2.0"
poetry remove fastapi
```

תלויות פיתוח בלבד:

```
poetry add --group dev pytest black
# התקנת כל הקבוצות:
poetry install --with dev
# או רק הפרודקשן:
poetry install --without dev
```

## קבצי הניהול

- pyproject.toml מצהיר מה הפרויקט צריך: שם, גרסת פייתון, תלויות, קבוצות, סקריפטים.
- poetry.lock נועל גרסאות מדויקות כדי שכולם יקבלו את אותה סביבה.

נעילת גרסאות ידנית:

```
poetry lock
```

ייצוא לקובץ requirements.txt כשצריך לכלים שאינם מכירים Poetry:

```
poetry export -f requirements.txt --output requirements.txt
```

דוגמת pyproject.toml מינימלית

```
[tool.poetry]
name = "mini_text_analyzer"
version = "0.1.0"
description = "ניקוי וניתוח טקסט קצר"
authors = ["Your Name <you@example.com>"]
readme = "README.md"
packages = [{ include = "mini_text_analyzer" }]

[tool.poetry.dependencies]
python = "^3.12"
numpy = "^2.1"
fastapi = "^0.115"

[tool.poetry.group.dev.dependencies]
pytest = "^8.3"
black = "^24.8"

[tool.poetry.scripts]
```

```
mini-text = "mini_text_analyzer.__main__:main"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

כעת אפשר להריץ:

```
poetry run mini-text
```

## מתי Poetry עדיף

- צוותים, CI/CD, והרבה תלויות שצריך לנהל בעקביות.
- פרויקטים שעתידים להפוך לחבילה או שירות מתמשך.
- כשצריך נעילת גרסאות קפדנית ופרופילים שונים לפיתוח מול פרודקשן.

## pyproject.toml מול requirements.txt: מתי לבחור במה

נראה כאילו שני הקבצים האלה עושים אותו דבר
רשימת חבילות ותלויות.
אבל האמת? הם מייצגים **שתי פילוסופיות שונות** של ניהול פרויקט.

## הצד השמרני, requirements.txt

requirements.txt הוא כמו צילום מצב של סביבה חיה.
הוא לא "מסביר" מה אתה רוצה, אלא **מה בדיוק רץ עכשיו אצלך**.

זה כאילו אתה אומר למחשב:

“אל תשאל שאלות. קח בדיוק את זה, אותן גרסאות, אל תשנה כלום.”

וזה נפלא כשאתה רוצה לשחזר סביבת עבודה במדויק
למשל על שרת Production או ב-Dockerfile:

```
pip install -r requirements.txt
```

אבל זו רשימה עיוורת.
היא לא יודעת מי תלוי במי, לא יודעת למה בחרת את הגרסאות האלה,
והיא תצבור רעש עם הזמן כמו "urllib3==1.26.3" שאף אחד לא התקין ישירות.

כלומר, היא טובה **לצילום מצב**, אבל לא **להצהרת כוונות**.

## הצד המודרני, pyproject.toml

פה מגיעה הגישה החדשה:
במקום לתעד מה קורה עכשיו, אתה מתאר **מה צריך לקרות**.

pyproject.toml הוא לא רק רשימה של חבילות,
הוא חוזה שלם שמספר:

- מה גרסת פייתון,
- מה שם הפרויקט,

- מי כתב אותו,
- מהן תלויות ה-runtime וה-dev,
- ואפילו איך להריץ אותו.

כלומר: לא **רק מה יש עכשיו**, אלא **מה צריך להיות תמיד**.

הקובץ הזה מאפשר לצוותים לעבוד יחד בלי לדרוך אחד לשני על הסביבה,
ומאפשר לכלי CI/CD לבנות את הפרויקט באופן דטרמיניסטי לחלוטין.

דוגמה:

```
[tool.poetry]
name = "mini_text_analyzer"
version = "0.2.0"
description = "מנוע עיבוד טקסט קטן לפרויקטי AI"
authors = ["Tomer Kedem <tomer@example.com>"]

[tool.poetry.dependencies]
python = "^3.12"
fastapi = "^0.115"
numpy = "^2.1"
openai = "^1.40"

[tool.poetry.group.dev.dependencies]
pytest = "^8.3"
```

```
black = "^24.8"
```

ברגע שתריץ:

```
poetry install
```

תיווצר לך סביבה וירטואלית נקייה עם כל החבילות המדויקות, וקובץ נוסף בשם `poetry.lock` ינעל את הגרסאות בפועל.

## תכל'ס – איך לבחור

- **אם אתה מפתח לבד**, או רק מנסה רעיון `requirements.txt` יספיק.
- **אם אתה עובד בצוות**, או מתכנן להריץ את הקוד ב-CI/CD לך על `pyproject.toml`.
- **אם אתה אוהב סדר**, ולא רוצה להילחם בתלויות שבורות Poetry ישמור עליך.

אבל הכי חשוב להבין: `requirements.txt` מתעד את העבר `pyproject.toml` מגדיר את העתיד.

## מעבר חלק בין העולמות

אפשר לחיות בשני הצדדים בלי מאבקי שליטה:

אם אתה עובד עם Poetry ורוצה לייצא קובץ קלאסי:

```
poetry export -f requirements.txt -o requirements.txt
```

אם אתה עובר מפרויקט ישן עם requirements.txt אל Poetry:

```
poetry init
poetry add $(cat requirements.txt)
```

בכמה פקודות, והפרויקט שלך קפץ עשור קדימה.

## נעילת גרסאות: למה ואיך עושים את זה נכון

בעולם ה-AI, עדכון קטן בגרסה עלול לשבור מודל שלם.

נעילת גרסאות שומרת על סביבה עקבית. כך שכל מפתח, שרת או Pipeline יריצו בדיוק את אותן חבילות.

## pip-tools – הדרך הפשוטה

כלי קטן מעל pip שמייצר **קובץ נעילה אמיתי**.

במקום לנהל requirements.txt ידנית, כותבים קובץ קליל בשם requirements.in:

```
fastapi>=0.115,<0.116
numpy>=2.1,<2.2
```

ואז מריצים:

```
pip install pip-tools
pip-compile
```

זה ייצור קובץ requirements.txt עם גרסאות מדויקות.

להחלת הסביבה בפועל:

```
pip-sync
```

תוצאה: כל מי שמריץ את הפקודות האלו יקבל בדיוק אותה סביבת עבודה.

## Poetry – הדרך המודרנית

Poetry עושה את אותו עיקרון, אבל אוטומטית.

כשמריצים:

```
poetry install
```

הוא קורא את ההצהרות שב-pyproject.toml

ויוצר קובץ נעילה (poetry.lock) עם גרסאות מדויקות.

אם משהו מתעדכן:

```
poetry lock
```

כך אתה שומר על סביבה יציבה גם כשמוסיפים חבילות חדשות.

## מתי להשתמש במה

- **pip-tools**:

  כשאתה בפרויקט קטן או עובד בסביבה ישנה שכבר מבוססת pip.

- **Poetry**:

  כשמדובר בפרויקט צוותי, מערכת גדולה או CI/CD.

שניהם שומרים על כלל הזהב:

"אותו קוד צריך לרוץ באותה צורה, בכל מקום."

## דוגמה מרכזית: סביבת פיתוח ל-mini_text_analyzer

נבנה סביבת עבודה אמיתית ונקייה לפרויקט שלנו: mini_text_analyzer.

הרעיון: ליצור סביבה מבודדת, להתקין רק את מה שצריך, ולהבטיח שכל מי שיפתח את הפרויקט יקבל בדיוק את אותן תלויות.

### שלב א – יצירת הסביבה

```
python -m venv .venv
# הפעלה:
# Windows
.\.venv\Scripts\activate
# Linux / Mac
source .venv/bin/activate
```

### שלב ב – התקנת הספריות

```
pip install numpy
pip freeze > requirements.txt
```

כך נוצר קובץ requirements.txt שמגדיר את גרסאות הספריות.

### שלב ג – מבנה הפרויקט

```
mini_text_analyzer/
 ├─── mini_text_analyzer/
 │    ├─── __init__.py
 │    ├─── __main__.py
 │    ├─── core/
```

```
│     │     └──── analyzer.py
│     └──── utils/
│        └──── text_tools.py
└──── requirements.txt
```

## שלב ד – הרצה

mini_text_analyzer/__main__.py

```python
from mini_text_analyzer.core.analyzer import analyze

def main() -> None:
    text = "פייתון היא שפה נהדרת לעיבוד טקסטים חכמים"
    print(analyze(text))

if __name__ == "__main__":
    main()
```

הרצה:

```
python -m mini_text_analyzer
```

פלט:

```
{'num_words': 6, 'avg_length': 5.5, 'most_common': 'פייתון'}
```

## טיפ – באותה מידה עם Poetry

אפשר להשיג בדיוק אותו דבר גם כך:

```
poetry init
poetry add numpy
poetry run python -m mini_text_analyzer
```

Poetry ינהל עבורך את כל ה-venv, ה-requirements והנעילה, בלי שתצטרך לחשוב עליהם בכלל.

## Best Practices: סביבה לכל פרויקט ו-gitignore.

## למה זה חשוב

פרויקטי AI נוטים להתנפח בתלויות, בקבצי מודל ובניסויים. כמה כללים פשוטים שומרים על פרויקט נקי, משוחזר וקל לעבודה בצוות.

## סביבה נפרדת לכל פרויקט

- לכל תיקייה יש `venv` משלה או Poetry משלה. לא מערבבים.
- מצהירים גרסת פייתון מפורשת: ב-`Poetry python = "^3.12"`, וב-README עבור `venv`.

## מה נכנס ל-git ומה לא

- **מכניסים**:

  קוד מקור, `pyproject.toml` ו-`poetry.lock` או `requirements.txt`, קונפיגים, סקריפטים, דוקומנטציה.

- **לא מכניסים:**

  תיקיית .venv, פלטים זמניים, קבצי מודל כבדים, קבצי נתונים פרטיים, קבצי מערכת.

## .gitignore מינימלי מומלץ

```
# סביבה וירטואלית
.venv/
venv/

# ארטיפקטים של פייתון
__pycache__/
*.pyc

# פלטי בנייה ואריזה
build/
dist/
*.egg-info/

# נתונים זמניים ולוגים
*.log
outputs/
.cache/

# סודות וקונפיג מקומי
.env
.env.*
secrets/
```

## סודות וקונפיג

- שומרים מפתחות API בקובץ .env מקומי שלא נכנס ל-git.
- טוענים אותם בקוד עם python-dotenv או דרך משתני סביבה של המערכת.

דוגמה:

```
# .env (לא ב-git)
OPENAI_API_KEY=sk-...
ENV=dev
```

```
# טעינה בקוד
from os import getenv
from dotenv import load_dotenv

load_dotenv()
api_key = getenv("OPENAI_API_KEY", "")
```

## עקביות בין מכונות

- אם עובדים עם pip: מנהלים requirements.in וקומפילציה ל-requirements.txt בעזרת pip-tools.
- אם עובדים עם Poetry: מתחייבים ל-poetry.lock ומריצים poetry install בכל קלון.
- לא מתקינים ידנית חבילות בלי לעדכן את קבצי הנעילה.

## טיפים קטנים לעבודה חכמה ב-VS Code

כדי לשמור על פרויקט נקי ועקבי גם כשעובדים בצוות, שווה להגדיר כמה דברים כבר בהתחלה:

- **בחירת Interpreter נכון**
  ב-VS Code פתח את ה-Command Palette וחפש:
  `Python: Select Interpreter`
  בחר את הסביבה הווירטואלית שלך (`.venv` או Poetry).
  זה מבטיח שכל אחד יריץ את הקוד באותה גרסה של הספריות.
- **שמירה על סגנון קוד אחיד**
  הפעל פורמט אוטומטי (למשל Black) ולינטינג (כמו Ruff או Pylint).
  זה חוסך עשרות שינויים מיותרים ב-git על רווחים וסוגריים.

- **תיעוד מהיר למפתחים חדשים**
ב-README של הפרויקט כתוב בקצרה איך מקימים את הסביבה:

```
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

כך כל מפתח חדש יכול להריץ את הפרויקט תוך דקות.

## מודלים ונתונים כבדים

- קבצי מודל וסטים גדולים נשמרים מחוץ ל-git. השתמשו ב-DVC, בשרתי אובייקטים, או באחסון ענן.
- מגדירים נתיבי ברירת מחדל בתצורה, לא קשיח בקוד.

## בדיקות וחבילות מערכת

- קובעים פרופילי התקנה: dev מול prod ב-Poetry או קובץ requirements-dev.txt נוסף במסלול pip.
- מתעדים תלות מערכתית שאינה פייתון (למשל CUDA, poppler) ב-README ובקובץ התקנה של Docker כשיש.

## סיכום: איזה כלי מתאים לפרויקטי AI

### קוד קטן, צוות קטן – venv ו-pip

כשעובדים לבד או בפרויקט קצר, אין צורך במערכת כבדה.
python -m venv .venv + pip install יעשו את העבודה.
תעד את החבילות ב-requirements.txt ותקבל סביבה שניתנת לשחזור בקלות.

### צוותים ומערכות חיות – Poetry

כשיש כמה מפתחים, בדיקות CI/CD, או פרויקט מתמשך Poetry מנצח.
הוא שומר על עקביות בין מכונות, מנהל קבוצות תלות (dev / prod)
ומספק קובץ נעילה (poetry.lock) שמונע הפתעות.
עכשיו, זה הסטנדרט ברוב צוותי ה-AI.

## מתי לשלב כלים נוספים

- **pip-tools**: מי שרוצה להישאר עם pip אבל לקבל נעילה אמיתית.
- **Docker**: כשצריך לבנות סביבה ניידת לשרתים או להרצה בענן.

- **conda / mamba**: במערכות מדעיות או ML כבד עם תלות ב-C/CUDA.

## השורה התחתונה

בחר בכלי שיתאים לגודל שלך היום,
אבל שיאפשר לך לגדול מחר.

- לפרויקטים קלים: venv + pip.
- לפרויקטים אמיתיים: **Poetry**.
- לפרויקטים עצומים: **Poetry** בתוך **Docker**.

וכמו תמיד בפייתון
"לא משנה כמה מהר התקנת, משנה שתוכל לשחזר את זה באותה קלות".

# פרק 7 – קבצים, נתיבים וקונפיגורציה

## למה עבודה עם קבצים קריטית ב-AI

כל פרויקט בינה מלאכותית, מתישהו, מוצא את עצמו מוקף בקבצים.

קובצי dataset עצומים, קובצי JSON עם הגדרות, checkpoints של מודלים, לוגים, CSV, קבצי תוצאות, גרפים, ועוד אלפי יצירות קטנות שמרכיבות את המערכת שלך.

אם אינך יודע לנהל אותם בצורה מסודרת, המחשב שלך יהפוך ל-AI בגרסה לא יציבה במיוחד כזה שמדבר עם עצמו בתיקיית downloads.
עבודה נכונה עם קבצים אינה רק טכנית, היא מבטאת **תודעה הנדסית**.
מפתח טוב יודע שהקוד שלו רץ בסביבות שונות: לוקאלית, בענן, במכונת לינוקס של צוות אחר או בתוך Docker.

לכן עליו להקפיד על **נתיבים חוצי מערכת הפעלה, קידוד אחיד (UTF-8), בדיקות קיום קבצים,** ו-**קונפיגורציה חיצונית** שמאפשרת לשנות פרמטרים בלי לגעת בשורה אחת של קוד.

היכולת הזו. להפריד בין לוגיקה, נתונים וקונפיגורציה.

היא מה שמבדיל בין "סקריפט שעובד אצלי"
לבין **מערכת שניתן להפעיל בכל מקום**.

## pathlib: הדרך המודרנית לעבוד עם נתיבים

פעם עבדנו עם מודול `os.path`. היום, הספרייה `pathlib` היא הדרך הפייתונית, הקריאה והנכונה לעבוד עם נתיבים.

היא אובייקטית, תומכת במערכות הפעלה שונות, וכוללת כמעט כל מה שנצטרך, מיצירת תיקיות ועד חיפוש קבצים לפי תבנית.

```
from pathlib import Path

# יצירת אובייקט נתיב לתיקייה הנוכחית
base_dir = Path(__file__).parent

# בניית נתיב חוצה מערכת הפעלה
data_path = base_dir / "data" / "dataset.csv"

# יצירת תיקייה אם לא קיימת
data_path.parent.mkdir(parents=True, exist_ok=True)

print(f"נתיב מלא: {data_path.resolve()}")
```

שימו לב: השימוש ב-/ בתוך Path אינו חיבור מחרוזות, אלא פעולה חכמה שמבינה את מבנה הנתיבים בכל מערכת הפעלה (Windows, Linux, macOS).

חיפוש קבצים הוא פשוט להפליא:

```
# איתור כל קבצי ה-JSON בתיקיית config
for file in base_dir.glob("config/*.json"):
    print(file.name)
```

וכדי לזהות את שורש הפרויקט (root):

ניתן לעבור כלפי מעלה עד שמזהים קובץ מובהק כמו pyproject.toml או .git.

```
from pathlib import Path

def find_project_root() -> Path:
    current = Path(__file__).resolve()
    for parent in current.parents:
        if (parent / ".git").exists() or (parent /
"pyproject.toml").exists():
            return parent
    raise RuntimeError("לא נמצא שורש הפרויקט")

root = find_project_root()
print(f"שורש הפרויקט: {root}")
```

## קריאה וכתיבה של טקסט (UTF-8 תמיד)

קידוד טקסט הוא אחד ממוקדי הכאב הגדולים במערכות רב-לשוניות.

בפרויקטים מודרניים, ובמיוחד בעבודה עם עברית, חובה להשתמש **תמיד** בקידוד UTF-8.

```
from pathlib import Path

text_file = Path("data/notes.txt")

# כתיבה
text_file.write_text("שלום עולם! זו שורה בעברית.", encoding="utf-8")

# קריאה
content = text_file.read_text(encoding="utf-8")
print(content)
```

השימוש ב-"encoding=utf-8" אינו מותרות, הוא ביטוח מפני תקלות מסתוריות של תווים משובשים בקונסול או בהעלאה לשרת.

## עבודה עם JSON

קובצי JSON משמשים כמעט לכל דבר: תצורה, נתונים, הגדרות, מודלים וכו'.

בפייתון נשתמש במודול json, אך נוסיף טיפ חשוב אחד.

בעת שמירה, נקבע ensure_ascii=False כדי לא לשבור טקסט בעברית.

```
import json
from pathlib import Path

config_path = Path("config/model.json")

config = {
    "model": "gpt-mini",
    "language": "עברית",
    "max_tokens": 512
}

# כתיבה
with config_path.open("w", encoding="utf-8") as f:
    json.dump(config, f, ensure_ascii=False, indent=2)

# קריאה
with config_path.open("r", encoding="utf-8") as f:
    loaded = json.load(f)

print(loaded)
```

## עבודה עם CSV

קובצי CSV הם עדיין דרך פופולרית להעביר datasets.
פייתון מאפשרת גישה אליהם גם דרך csv.DictReader וגם באמצעות pandas לעיבוד מתקדם.

### קריאה באמצעות csv.DictReader:

```
import csv
from pathlib import Path

path = Path("data/users.csv")

with path.open("r", encoding="utf-8") as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(row["name"], row["email"])
```

### קריאה באמצעות pandas:

```
import pandas as pd

df = pd.read_csv("data/users.csv", encoding="utf-8")
print(df.head())

# סינון וכתיבה מחדש
df = df[df["active"] == True]
```

```
df.to_csv("data/active_users.csv", index=False, encoding="utf-8")
```

בעולם של AI, קובצי CSV עלולים להיות כבדים ואיטיים.
הפתרון הנפוץ הוא להשתמש בפורמטים בינאריים כמו **Parquet** או **Feather** שמאפשרים טעינה מהירה פי כמה:

```
import pandas as pd

df = pd.read_csv("data/users.csv")
df.to_parquet("data/users.parquet", index=False)

# טעינה מהירה יותר
df2 = pd.read_parquet("data/users.parquet")
```

פורמטים אלו נתמכים ישירות ב-pandas ומומלצים מאוד לעבודה עם datasets גדולים בענן.

## קונפיגורציה חיצונית (JSON/YAML)

אף אחד לא רוצה לפתוח קוד ולשנות שם API Key או מיקום Dataset.
כל ערך כזה צריך לשבת בקובץ קונפיגורציה חיצוני, JSON או YAML.

```
import json
```

```
from pathlib import Path

config_path = Path("config/app.json")

def load_config() -> dict:
    if not config_path.exists():
        raise FileNotFoundError("קובץ קונפיגורציה חסר")
    return json.loads(config_path.read_text(encoding="utf-8"))

cfg = load_config()
print(f"מפתח API: {cfg['api_key']}")
```

אם מעדיפים YAML (קריא יותר לאנשים),
ניתן להשתמש ב-PyYAML:

```
import yaml

with open("config/app.yaml", "r", encoding="utf-8") as f:
    cfg = yaml.safe_load(f)
```

הרעיון פשוט: **אין לשנות קוד כדי לשנות התנהגות**.

**לעיתים** נרצה להחזיק כמה גרסאות של קונפיגורציה.
אחת לפיתוח, אחת לבדיקה ואחת ל-Production.

אפשר לעשות זאת בקלות בעזרת משתנה סביבה פשוט:

```
import os, json
from pathlib import Path
```

```
env = os.environ.get("APP_ENV", "dev")
config_path = Path(f"config/config.{env}.json")

config = json.loads(config_path.read_text(encoding="utf-8"))
print(f"נטענה קונפיגורציה לסביבה: {env}")
```

## os.environ – משתני סביבה ו-dotenv

קבצי קונפיגורציה נוחים, אך לעיתים הם כוללים מידע רגיש.
כגון: סיסמאות או מפתחות API.

לכן נעדיף לשמור פרטים כאלה במשתני סביבה (os.environ).

```
import os

api_key = os.environ.get("API_KEY")
if not api_key:
    raise RuntimeError("חסר משתנה סביבה API_KEY")
```

כדי לנהל משתנים כאלה בסביבה מקומית, נשתמש בקובץ .env
יחד עם הספרייה python-dotenv:

```
from dotenv import load_dotenv
load_dotenv()  # טוען את הקובץ .env לשימוש

db_user = os.environ["DB_USER"]
db_pass = os.environ["DB_PASS"]
```

קובץ .env ייראה כך:

```
DB_USER=tomer
```

```
DB_PASS=1234secure
API_KEY=abcd-efgh
```

והוא **לעולם לא נכנס ל-git!** (הוסיפו .env ל-.gitignore).

## דוגמה מרכזית: קריאת Dataset, ניקוי ושמירה

נניח שיש לנו קובץ CSV עם שמות משתמשים, אימיילים וסטטוס.
נרצה לנקות אותו ולשמור גרסה נקייה.

```
import pandas as pd
from pathlib import Path


# יצירת תיקייה וקובץ לדוגמה
base_dir = Path(__file__).parent
data_dir = base_dir / "data"
data_dir.mkdir(parents=True, exist_ok=True)

# יצירת קובץ users_raw.csv לדוגמה
input_path = data_dir / "users_raw.csv"
data = {
    "email": ["example1@gmail.com", "example2@gmail.com",
None, "example1@gmail.com"],
    "name": ["Alice", "Bob", "Charlie", "Alice"]
}
df = pd.DataFrame(data)
df.to_csv(input_path, index=False, encoding="utf-8")
```

```
print(f"נוצר קובץ לדוגמה בנתיב: {input_path.resolve()}")

def clean_dataset(file_path: Path) -> pd.DataFrame:
    """קורא dataset ומחזיר גרסה נקייה"""
    df = pd.read_csv(file_path, encoding="utf-8")
    df = df.dropna(subset=["email"])        # מסיר שורות ללא אימייל
    df["email"] = df["email"].str.lower()   # מאחד רישיות
    df = df.drop_duplicates(subset=["email"]) # מסיר כפילויות
    return df

base_dir = Path(__file__).parent
input_path = base_dir / "data/users_raw.csv"
output_path = base_dir / "data/users_clean.csv"

cleaned = clean_dataset(input_path)
cleaned.to_csv(output_path, index=False, encoding="utf-8")

print(f"נשמר קובץ נקי בנתיב: {output_path.resolve()}")
```

הדוגמה הזו ממחישה את היסוד של עבודה נקייה עם נתונים: נתיבים ברורים, קידוד אחיד, שליטה בתוצאות.

## טוען אוניברסלי לפי סוג הקובץ

לעיתים נרצה פונקציה אחת שתדע להתמודד עם כל סוגי הקבצים הנפוצים (JSON, YAML, CSV) באופן אחיד.

```python
import json, yaml, pandas as pd
from pathlib import Path

def load_file(path: Path):
  if path.suffix == ".json":
    return json.loads(path.read_text(encoding="utf-8"))
  if path.suffix in (".yml", ".yaml"):
    return yaml.safe_load(path.read_text(encoding="utf-8"))
  if path.suffix == ".csv":
    return pd.read_csv(path, encoding="utf-8")
  raise ValueError(f"סוג קובץ לא נתמך: {path.suffix}")
```

כך ניתן לעבוד עם כל סוגי הקבצים באותה דרך.
גישה נקייה, גנרית ואידיאלית למערכות AI מרובות מקורות.

## Best Practices

- **השתמשו תמיד ב-UTF-8** אל תסמכו על ברירת המחדל של מערכת ההפעלה.
- **בדקו קיום קבצים** (`Path.exists()`) לפני קריאה.
- **הפרידו בין קוד לקונפיגורציה** אל תשנו קוד כדי לשנות הגדרות.
- **הימנעו מנתיבים קשיחים** השתמשו ב-`Path(__file__)` וב-`/`.
- **הוסיפו לוגים בעת קריאה וכתיבה של קבצים** כדי לדעת מה נכשל ומתי.
- **אל תשמרו סיסמאות בקוד** רק בקובץ `.env` או במשתני סביבה.

## סיכום – למה קונפיגורציה נכונה חוסכת כאב ראש

מתכנתים צעירים מתלהבים מקוד רץ.
מתכנתים מנוסים מתלהבים מקוד שניתן לפרוס, להפעיל, ולתחזק.
ניהול נכון של קבצים, נתיבים וקונפיגורציה הוא הצעד הראשון במעבר ממפתח "שעובד אצלי" למפתח **שעובד בכל מקום**.

כשכל הנתיבים נבנים נכון, כל הקבצים נכתבים ב-UTF-8,
וההגדרות יושבות מחוץ לקוד

אתה ישן טוב יותר בלילה, גם כשה-AI שלך רץ על שרת בצד השני של העולם.

# פרק 8 – חריגות, לוגים ואבחון

## למה טיפול בשגיאות הוא קריטי ב-AI

מערכת מבוססת AI שונה ממערכת רגילה בכך שהיא לעולם אינה יודעת הכול.
היא לומדת, משערת, מנחשת, ולפעמים פשוט טועה.
אבל יש גורם אחד שאסור לו לטעות, **המהנדס** שבונה אותה.

ולכן, טיפול בשגיאות ולוגים הוא לא רק נושא טכני, זו **שכבת ביטחון מנטלית**: מה יקרה כשהכול ישתבש?

**מפתח בלי טיפול שגיאות, זה כמו טייס בלי מכשור.**

## המבנה הבסיסי – try / except / else / finally

כל שפת תכנות מציעה דרך להתמודד עם שגיאות.
בפייתון, זה נעשה באמצעות בלוק try/except, עם שני תוספים חשובים: else ו-finally.

```
def load_model(path: str):
    try:
        print("טוען מודל...")
        with open(path, "rb") as f:
            model = f.read()
    except FileNotFoundError:
        print("❌ הקובץ לא נמצא.")
```

```
except Exception as e:
    print(f"⚠ שגיאה לא צפויה: {e}")
else:
    print("✅ המודל נטען בהצלחה.")
finally:
    print("ניקוי משאבים...")
```

**הסדר חשוב:**

- Try – קוד שעלול להיכשל.
- Except – טיפול בשגיאות ידועות או כלליות.
- Else – קוד שרץ רק אם **לא** הייתה שגיאה.
- Finally – קוד שרץ תמיד, גם במקרה של כישלון (לניקוי משאבים, סגירת חיבורים וכו').

במערכות AI אמיתיות נשתמש כמעט תמיד בכל הארבעה. במיוחד כשיש קריאות API, קריאה מקבצים או טעינת מודלים.

## יצירת חריגות מותאמות (Custom Exceptions)

ככל שהמערכת שלך גדלה, תרצה לדעת לא רק ש"הייתה שגיאה" אלא **איזה סוג שגיאה** ולמה.

במקום לזרוק Exception כללי, ניצור חריגות מותאמות משלנו.

```
class ModelNotFoundError(Exception):
    """נזרק כאשר קובץ המודל חסר."""
```

```
    pass

class InvalidDatasetError(Exception):
    """נזרקת כאשר מבנה הנתונים שגוי."""
    pass


def load_dataset(path: str):
    if not Path(path).exists():
        raise InvalidDatasetError(f"הקובץ {path} לא קיים.")
```

יתרון עצום של גישה זו הוא יכולת טיפול ממוקדת:

```
try:
    load_dataset("data/train.csv")
except InvalidDatasetError as e:
    logger.error(f"שגיאה בטעינת הנתונים: {e}")
```

כך אפשר להבדיל בין "בעיה בנתונים" לבין "בעיה ברשת", בין "מודל חסר" ל"טוקנים שנגמרו".

## logging בסיסי – רמות INFO/WARNING/ERROR

קריאות print הן כמו הודעות בוואטסאפ, הן זמניות ונעלמות. לוגים, לעומת זאת, הם **היסטוריה רשמית של מה שהתרחש**.

```
import logging

logging.basicConfig(
```

```
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(message)s",
    encoding="utf-8"
)

logging.info("המערכת הופעלה")
logging.warning("המודל איטי מהרגיל")
logging.error("נכשלה טעינת dataset")
```

רמות הלוגינג:

- **DEBUG** – למידע מפורט על זרימת הקוד.
- **INFO** – לאירועים רגילים.
- **WARNING** – בעיה לא קריטית.
- **ERROR** – תקלה חמורה אך ניתנת להתאוששות.
- **CRITICAL** – כשצריך לעצור הכול ולקרוא למתכנת באמצע הלילה.

במקום print, השתמש תמיד ב-logger. הוא יודע לרשום לקבצים, ל-stdout, ל-syslog, ולשירותים כמו ELK או Datadog.

## correlation ID-ו Structured Logging – extra dict

כשיש לך עשרות microservices, מאות משתמשים ומיליארדי טוקנים, לוגים רגילים כבר לא מספיקים.

Structured Logging מאפשר להוסיף **שדות קבועים** לכל הודעה, כך שמערכות ניתוח לוגים (כמו Kibana או Grafana) יוכלו לפלטר, לקבץ ולזהות בעיות במהירות.

```
import logging
import uuid

logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO, format="%(asctime)s
[%(levelname)s] %(message)s")

def process_request(user_id: str):
    correlation_id = str(uuid.uuid4())  # מזהה ייחודי לבקשה
    try:
        logger.info("מתחיל עיבוד בקשה", extra={"correlation_id":
correlation_id, "user_id": user_id})
        # קוד עיבוד...
        raise ValueError("שגיאה מדומה")
    except Exception as e:
        logger.error("כישלון בעיבוד", extra={"error": str(e),
"correlation_id": correlation_id})
```

עקרון **Correlation ID** (מזהה מתאם) נועד לאפשר מעקב אחרי כל בקשה או תהליך לאורך כל שלבי המערכת.

לכל בקשה מוקצה מזהה ייחודי, וכל הלוגים שנוצרים במהלכה כוללים את אותו מזהה. כך שניתן לעקוב אחר הזרימה שלה מתחילתה ועד סופה, גם במערכות מבוזרות או בפייפליין מורכב.

## דוגמה מרכזית: עטיפת pipeline עם לוגים וחריגות

נראה עכשיו איך משלבים הכול יחד במערכת אחת שעושה ניקוי נתונים
ו-AI Inference.

```
import logging
import pandas as pd
from pathlib import Path

logger = logging.getLogger("pipeline")
logging.basicConfig(level=logging.INFO, format="%(asctime)s
[%(levelname)s] %(message)s", encoding="utf-8")

class PipelineError(Exception):
    pass

def load_data(path: Path) -> pd.DataFrame:
    if not path.exists():
```

```python
        raise FileNotFoundError(f"לא נמצא הקובץ {path}")
    df = pd.read_csv(path, encoding="utf-8")
    if df.empty:
        raise PipelineError("ה-dataset ריק")
    return df

def run_inference(df: pd.DataFrame):
    if "text" not in df.columns:
        raise PipelineError("עמודה 'text' חסרה ב-dataset")
    df["length"] = df["text"].str.len()
    return df

def main():
    try:
        logger.info("🚀 התחלת pipeline")
        data = load_data(Path("data/input.csv"))
        result = run_inference(data)
        result.to_csv("data/output.csv", index=False,
encoding="utf-8")
        logger.info("✅ pipeline הושלם בהצלחה")
    except PipelineError as e:
        logger.error(f"❌ שגיאה בתהליך: {e}")
    except Exception as e:
        logger.exception(f"⚠ שגיאה כללית: {e}")
    finally:
        logger.info("🔚 סיום pipeline")
```

```python
if __name__ == "__main__":
    main()
```

דוגמה זו משקפת את המציאות: טעינה, עיבוד ושמירה של נתונים, עם טיפול בחריגות ולוגים. הכל במהלך אחד מסודר וברור.

## Best Practices

- **אל תבלעו חריגות**
  `except Exception: pass` הוא אויב. עדיף לכתוב לוג ולטפל.
- **השתמשו ב-logger במקום print**
  כדי לשלוט ברמות, לנתב ולשמור היסטוריה.
- **אל תחשפו מידע רגיש בלוגים**
  (סיסמאות, API keys).
- **תעדו חריגות במבנה עקבי**
  סוג, זמן, מזהה בקשה.
- **הוסיפו correlation ID**
  לכל תהליך ארוך או בקשה חיצונית.
- **תנו שמות חריגה משמעותיים**
  לא `CustomError`, אלא `ModelNotLoadedError`.
- **שמרו לוגים לקובץ נפרד בכל מודול חשוב**
  (למשל `api.log`, `pipeline.log`).

## סיכום – לוגים טובים הם העיניים של המערכת

בעולם שבו המידע זורם במהירות והמערכות מבוזרות, לוגים הם הדרך היחידה להבין מה באמת קרה.
חריגות אומרות לנו **מה נכשל**, ולוגים מספרים **איך זה קרה**.

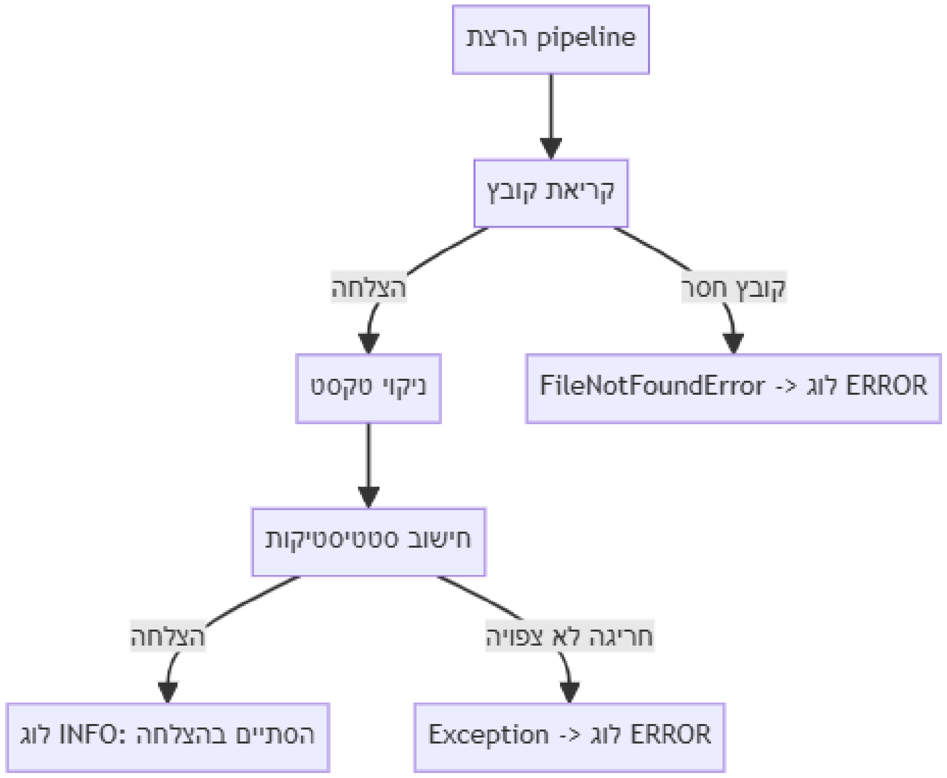
במערכת AI חכמה, לא מספיק לדעת לטפל בשגיאה.

צריך לדעת **לזהות אותה בזמן, להבין את ההקשר, ולהמשיך לעבוד**.

לוגים טובים הם לא רעש, הם מצפן.

וכשמגיע הבאג הראשון בפרודקשן, הם יהיו הקול השפוי היחיד שיספר לך את האמת.

## תרשים זרימה של טיפול בשגיאות



# פרק 9 – תכנות מונחה עצמים בסיסי

## למה OOP חשוב גם כשיש functions

בעידן של פונקציות, Pipelines ו-AI Agents,
קל לשכוח את יסודות ה-OOP.

תכנות מונחה עצמים (Object Oriented Programming) אינו רק סגנון ישן, אלא דרך **לארגן מחשבה הנדסית**.

כשמערכת גדלה, מתחילים להופיע נתונים שקשורים זה בזה, פעולות שחוזרות על עצמן, וקוד שצריך "זהות" משל עצמו.

פה בדיוק נכנסת המחלקה: היא מאפשרת **לאגד נתונים (state) והתנהגות (behavior)** לאובייקט אחד מסודר.

במערכות AI, זה חיוני.

לדוגמה, מודל למידת מכונה הוא אובייקט עם פרמטרים, פונקציות עיבוד, וסטטוס של אימון .
Tokenizer הוא אובייקט עם state של מילון ומאפייני ניקוי טקסט.

אפילו pipeline של עיבוד נתונים הוא מחלקה שמאגדת כמה שלבים תחת זהות אחת.

אז גם כשיש לך פונקציות מצוינות

כשאתה רואה **ישות עם התנהגות ומידע** כנראה הגיע הזמן להפוך אותה למחלקה.

## הגדרת מחלקה בסיסית (init, self)

בפייתון, מחלקה נוצרת בעזרת מילת המפתח class.
המתודה __init__ היא הפונקציה שמופעלת בכל פעם שנוצר מופע חדש של המחלקה, ו-self מייצגת את האובייקט הנוכחי.

```
class Counter:
    """מונה פשוט שמספור שלו נשמר בזיכרון."""

    def __init__(self, start: int = 0):
        self.value = start

    def increment(self, step: int = 1) -> None:
        self.value += step

    def get(self) -> int:
        return self.value

c = Counter(10)
c.increment()
print(c.get())  # 11
```

שימו לב:

- כל מתודה מקבלת כפרמטר ראשון את self.
- השדות של המחלקה מאוחסנים כמאפיינים (self.value).
- כל מופע (instance) שומר state משלו.

## מתודות מיוחדות (str, repr, len)

בפייתון קיימות מתודות "קסם" (magic methods) שמאפשרות לשלוט בהתנהגות ברירת המחדל של המחלקה:

```
class Vector:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

    def __str__(self) -> str:
        return f"({self.x}, {self.y})"

    def __len__(self) -> int:
        return abs(self.x) + abs(self.y)

v = Vector(3, -4)
print(v)      # (3, -4)
print(len(v)) # 7
```

- `__str__` מגדיר איך האובייקט יוצג למשתמש.
- `__repr__` מגדיר ייצוג מפורט למפתחים (בד"כ דומה).
- `__len__` מאפשר להשתמש ב- `len(obj)` כאילו זה רשימה.

# מתודות של מופע – גישה למשתני מחלקה

## מתודות מופע מול משתני מחלקה

בכל מחלקה אפשר להגדיר שני סוגים של משתנים:

**משתני מחלקה** ו-**משתני מופע**.

- משתני מחלקה שייכים למחלקה עצמה, וכל המופעים חולקים אותם.
- משתני מופע שייכים רק לאובייקט שנוצר בפועל.

לכל מחלקה יכולים להיות

**משתני מחלקה (class attributes)** ערכים שחלים על כל המופעים.

```
class Tokenizer:
  language = "hebrew"  # משתנה מחלקה - שייך לכולם

  def __init__(self, text: str):
    self.text = text  # שייך רק למופע הנוכחי

  def tokens(self) -> list[str]:
    return self.text.split()

t1 = Tokenizer("שלום עולם")
t2 = Tokenizer("מה נשמע")

print(t1.language, t2.language)  # hebrew hebrew
```

```
t2.language = "english"
print(t1.language, t2.language)  # hebrew english
```

כששינית את t2.language, בעצם יצרת **עותק חדש** של המשתנה בתוך המופע t2.
המשתנה של המחלקה עצמה (Tokenizer.language) לא השתנה.

אם תרצה לשנות את הערך עבור כל המחלקה, כתוב:

```
Tokenizer.language = "english"
```

או עשה זאת מתוך מתודת מחלקה (@classmethod).

## @classmethod ו-@staticmethod

### למה בכלל צריך @classmethod

במחלקה רגילה יש מתודות שפועלות על מופע (instance).

כלומר, הן מקבלות את self, ועובדות על הנתונים של אותו מופע:

```
class User:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f" היי{self.name}!")
```

```
u = User("תומר")
u.greet()  # היי תומר!
```

אבל לפעמים יש פעולות:

- שלא קשורות למופע מסוים
- או שקשורות לכל המחלקה כולה

כאן נכנסים שני הדקורטורים החשובים האלה.

## @classmethod — מתודה שפועלת על המחלקה עצמה

@classmethod מקבלת את **המחלקה (cls)** במקום מופע (self).

היא שימושית כשצריך לעבוד על משתנים **ששייכים למחלקה כולה**, או כשצריך לבנות מופעים בדרך שונה.

**דוגמה 1 – ספירת מופעים**

```
class Model:
    instances = 0

    def __init__(self):
        Model.instances += 1

    @classmethod
    def how_many(cls):
        return cls.instances



m1 = Model()
```

```
m2 = Model()
print(Model.how_many())  # 2
```

כאן how_many לא תלויה במופע מסוים

היא מדווחת כמה מופעים נוצרו עד עכשיו.

פייתון מעבירה אליה את המחלקה עצמה (cls),

כך שאפשר לגשת ל-cls.instances.

## דוגמה 2 – מפעל יצירה (Factory Method)

```
class User:
    def __init__(self, name, is_admin=False):
        self.name = name
        self.is_admin = is_admin

    @classmethod
    def admin(cls, name):
        return cls(name, is_admin=True)


u1 = User("Dana")
u2 = User.admin("Tomer")

print(u1.is_admin)  # False
print(u2.is_admin)  # True
```

כך אפשר ליצור משתמש אדמין ישירות,

בלי לכתוב User("Tomer", True) הקוד ברור יותר וקריא.

## @staticmethod — פונקציה כללית שנמצאת במחלקה רק בשביל סדר

@staticmethod לא מקבלת לא self ולא cls.
זו סתם פונקציה "צמודה" למחלקה מבחינה לוגית—
כלומר, היא קשורה לנושא של המחלקה, אבל לא תלויה בה.

### דוגמה

```
class MathUtils:
  @staticmethod
  def add(a, b):
    return a + b
```

## למה בכלל צריך @staticmethod

תחשוב על זה כך:
לפעמים יש לך פונקציה שעוזרת למתודות אחרות במחלקה,
אבל **היא לא צריכה לדעת שום דבר על המחלקה או על המופע**.

אם תשאיר אותה מחוץ למחלקה, היא תאבד את ההקשר הלוגי שלה.
אבל אם תשים אותה בתוך המחלקה, היא נשארת קרובה ונגישה,
וזה עושה סדר.

## דוגמה מוחשית

נגיד יש לך מחלקה שמייצגת הזמנה:

```
class Order:
  def __init__(self, items):
    self.items = items

  def total(self):
    return sum(self.items)

  @staticmethod
  def apply_vat(amount):
    return amount * 1.17
```

עכשיו נוכל להשתמש בה ככה:

```
order = Order([10, 20, 30])
subtotal = order.total()
total_with_vat = Order.apply_vat(subtotal)
print(total_with_vat)  # 70.2
```

שים לב:

- apply_vat לא צריכה גישה לא ל-self ולא ל-cls.
- היא פשוט **פונקציה עזר שקשורה לנושא של המחלקה**, אז נוח לשים אותה שם, כדי שכל מה שקשור ל-Order יהיה מרוכז באותו מקום.

## זה עניין של עיצוב (Design)

אם היית שם את `apply_vat` כפונקציה חיצונית:

```
def apply_vat(amount): ...
```

היא הייתה עובדת בדיוק אותו דבר

אבל היא הייתה "תלושה" מהקוד הלוגי של ההזמנות.

`@staticmethod` עוזרת לשמור על ארגון טוב בקוד:

פונקציות שקשורות לנושא מסוים נשארות יחד,

גם אם הן לא תלויות במחלקה.

## איך להחליט מתי להשתמש בה

שאל את עצמך:

האם הפונקציה הזו קשורה לוגית למחלקה,

אבל לא צריכה מידע ממנה?

אם התשובה כן, זו מתודה סטטית קלאסית.

## @dataclass – פחות boilerplate, יותר קריאות

כשאתה מגדיר מחלקה פשוטה. למשל בשביל לייצג משתמש, לקוח, או מוצר. אתה כמעט תמיד כותב את אותן שלוש מתודות שוב ושוב:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        return f"Person(name={self.name}, age={self.age})"

    def __eq__(self, other):
        return self.name == other.name and self.age == other.age
```

זה מלא חזרתיות. אין פה לוגיקה חכמה, אלא רק קוד טכני שחוזר על עצמו.

### מה @dataclass עושה בשבילך

`@dataclass` אומר לפייתון:

"תעשי בשבילי את כל הדברים הסטנדרטיים האלה, אני רק אגדיר את השדות."

```
from dataclasses import dataclass

@dataclass
class Person:
```

```
    name: str
    age: int
```

וזהו.

פייתון מייצרת לך אוטומטית:

- `__init__` בנאי שמכניס את כל הערכים למופע
- `__repr__` הדפסה יפה וברורה
- `__eq__` השוואה בין אובייקטים לפי הערכים שלהם

כך זה עובד בפועל

```
p1 = Person("Dana", 30)
p2 = Person("Dana", 30)
p3 = Person("Noam", 12)

print(p1)       # Person(name='Dana', age=30)
print(p1 == p2)  # True  (אותם ערכים)
print(p1 == p3)  # False
```

בלי שכתבת אף אחת מהמתודות האלה בעצמך.

פשוט, נקי, וקל לתחזוקה.

## מתי זה שימושי במיוחד

- כשיש לך **אובייקטים שהם בעיקר "נתונים"**, לא לוגיקה כבדה.
  למשל: `Config`, `Point`, `Order`, `Book`, `User`.
- כשאתה כותב **מודלים או מבני נתונים** לקוד אחר.

במיוחד בפרויקטים של ML, APIs או בדיקות.

- כשאתה רוצה **קוד קצר וברור**, במיוחד בקבצים עם הרבה הגדרות מחלקה קטנות.

## תוספות חכמות (כשמתקדמים)

`frozen=True` – הופך את האובייקט לבלתי ניתן לשינוי (immutable).

לדוגמה:

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Point:
    x: int
    y: int

p = Point(3, 4)
print(p)        # Point(x=3, y=4)

# זה עובד רגיל — אתה יכול לקרוא לערכים
print(p.x + p.y)  # 7

# אבל אם תנסה לשנות ערך:
p.x = 10        # ❌ זה יגרום לשגיאה בזמן ריצה
```

הפלט בפועל יהיה:

```
Point(x=3, y=4)
7
Traceback (most recent call last):
  ...
dataclasses.FrozenInstanceError: cannot assign to field 'x'
```

## הסבר קצר

כשאתה מוסיף `frozen=True`,
פייתון מונעת כל שינוי בשדות אחרי שהאובייקט נוצר.
זה שימושי כשאתה רוצה לוודא שאובייקט **ישאר קבוע**
למשל נקודה במרחב, מזהה משתמש, או קונפיגורציה של מערכת.

אם תסיר את `frozen=True`, הקוד יעבוד רגיל:

```
@dataclass
class Point:
    x: int
    y: int

p = Point(3, 4)
p.x = 10   # ✅ עובד, כי המחלקה לא קפואה
print(p)   # Point(x=10, y=4)
```

`order=True` – מאפשר להשוות בין מופעים לפי סדר (<, >).

```
from dataclasses import dataclass

@dataclass(order=True)
class Product:
    price: float
    name: str


p1 = Product(29.90, "Notebook")
p2 = Product(9.90, "Pencil")
p3 = Product(99.00, "Backpack")

print(p1 > p2)   # True  (29.9 > 9.9)
print(p1 < p3)   # True  (29.9 < 99.0)
print(sorted([p1, p2, p3]))
```

הפלט יהיה:

```
True
True
[Product(price=9.9, name='Pencil'), Product(price=29.9,
name='Notebook'), Product(price=99.0, name='Backpack')]
```

כשאתה כותב `@dataclass(order=True)`, פייתון יוצרת עבורך אוטומטית את כל המתודות להשוואה:

- `__lt__` קטן מ- <
- `__le__` קטן או שווה <=
- `__gt__` גדול מ- >
- `__ge__` גדול או שווה >=

ברירת המחדל היא שההשוואה מתבצעת **לפי סדר השדות** שהגדרת.

בדוגמה שלנו, קודם לפי price,  ואם יש שוויון, לפי name.

## טיפ שימושי

אם אתה רוצה להשוות לפי שדה אחד בלבד (למשל רק לפי מחיר), תוכל להשתמש בפרמטר `field(compare=False)` על שדות שלא צריכים להשתתף בהשוואה:

```
from dataclasses import dataclass, field

@dataclass(order=True)
class Product:
   price: float
   name: str = field(compare=False)
```

ככה ההשוואה תתבסס רק על המחיר, בלי להתחשב בשם בכלל.

מאת: תומר קדם

## לסיכום

| פרמטר | משמעות |
|---|---|
| **frozen=True** | מונע שינוי בשדות אחרי יצירה (immutable) |
| **order=True** | מוסיף תמיכה בהשוואה וסידור לפי ערכים |
| **compare=False** | מוציא שדה מסוים מהשוואה |

## קומפוזיציה מול ירושה – מתי מה

ירושה (Inheritance) מאפשרת להרחיב מחלקה קיימת, אבל ב-AI ובפרויקטים מודרניים משתמשים בה בזהירות.

קומפוזיציה (Composition), שילוב של אובייקטים אחרים, לרוב עדיפה.

```
class Cleaner:
    def clean(self, text: str) -> str:
        return text.lower().strip()

class Tokenizer:
    def tokenize(self, text: str) -> list[str]:
        return text.split()

class TextProcessor:
    def __init__(self):
```

```python
        self.cleaner = Cleaner()
        self.tokenizer = Tokenizer()

    def process(self, text: str) -> list[str]:
        cleaned = self.cleaner.clean(text)
        return self.tokenizer.tokenize(cleaned)

tp = TextProcessor()
print(tp.process("   שלום עולם"))  # ['שלום', 'עולם']
```

במקום "להיות" Cleaner, TextProcessor רק **משתמש** בו, וזה הרבה יותר גמיש ובטוח.

## דוגמה מרכזית: מחלקת TextCleaner עם API נקי

ניצור מחלקה אחת שימושית לעיבוד טקסטים, עם state נקי ו-API פשוט.

```
from dataclasses import dataclass
import re

@dataclass
class TextCleaner:
    lower: bool = True
    remove_punct: bool = True

    def clean(self, text: str) -> str:
        """מנקה טקסט לפי ההגדרות."""
        result = text
        if self.lower:
            result = result.lower()
        if self.remove_punct:
            result = re.sub(r"[^\w\s]", "", result)
        return result.strip()

# שימוש:
cleaner = TextCleaner(lower=True)
print(cleaner.clean("שלום, עולם!!!"))  # שלום עולם
```

היתרונות ברורים:

- קוד קריא.
- התנהגות ניתנת לשינוי באמצעות פרמטרים.
- ניתן לשלב אותה ב-pipeline מבלי לשכתב פונקציות.

מאת: תומר קדם

## Best Practices

- **PascalCase**
  לשמות מחלקות (למשל TextCleaner, לא text_cleaner).
- **לא להגזים בירושות**
  העדף קומפוזיציה.
- **אל תיצור מחלקות סתמיות**
  אם אין state, עדיף פונקציה.
- **השתמש ב- @dataclass**
  כשמדובר באובייקט פשוט.
- **שמור על SRP (Single Responsibility Principle)**
  מחלקה אחת, תפקיד אחד.
- **אל תסתיר נתונים סתם**
  פייתון סומכת על מפתחים בוגרים, לא על אינקפסולציה כפויה.

## סיכום – OOP בזהירות, לא הכל צריך להיות מחלקה

תכנות מונחה עצמים בפייתון הוא כלי, לא דת.
כשהוא בשימוש נכון – הוא מבהיר, מאחד ומונע כפילויות.
כשהוא בשימוש יתר – הוא מוסיף שכבות מיותרות ומסבך את הקוד.

מחלקה טובה צריכה להיות **ישות בעלת זהות ומשמעות** לא עטיפה אקראית לפונקציות.
בעידן של פונקציות חכמות ו-AI Agents, דווקא הבנה טובה של עקרונות OOP בסיסיים מאפשרת לכתוב קוד שקל להבין, להרחיב ולתחזק, גם במערכות מורכבות ומשתנות.

# פרק 10 – טיפוסיות סטטית עם typing

## למה type hints בשפה דינמית

פייתון נולדה כשפה דינמית. אין צורך להגדיר מראש מהו סוג הערך, והקוד פשוט "רץ".

זה נוח, זריז, וגורם למתכנת להרגיש חופשי.

עד היום שבו המערכת גדלה, ומישהו אחר מנסה להבין למה פונקציה שמחזירה מחרוזת לפעמים מחזירה גם None.

Type hints אינם באים לשנות את פייתון לשפה סטטית.
הם באים לתת **שכבת משמעות** לקוד.

מעין מסמך חי שמספר למתכנתים (ולכלים)
מה בדיוק הפונקציה מצפה לקבל ומה היא מחזירה.

בעידן שבו קוד AI מורכב מצינורות נתונים, מודלים, ופונקציות שמדברות זו עם זו,
רמזי טיפוס (type hints) הם כמו תמרורים בכביש: לא עוצרים את התנועה,
אבל מונעים תאונות.

## type hints בסיסיים (int, str, list, None)

רמזי טיפוס הם פשוט הערות על חתימת הפונקציה, בעזרת תחביר מובנה.

```
def add(a: int, b: int) -> int:
    return a + b

def greet(name: str) -> str:
    return f"שלום{name}"

def maybe_divide(x: float, y: float) -> float | None:
    if y == 0:
        return None
    return x / y
```

- `a: int` פירושו שהפרמטר `a` אמור להיות מספר שלם.
- `-> int` פירושו שהפונקציה מחזירה שלם.
- שימוש ב-`|` `None` (או `Optional`) מציין שהפונקציה עלולה להחזיר None.

הטיפוסים לא נבדקים בזמן ריצה, פייתון לא תזרוק חריגה אם שלחת מחרוזת, אבל כלים חיצוניים (כמו mypy או Pyright) כן יזהו את הבעיה בזמן פיתוח.

## טיפוסים מורכבים Union, Optional, Literal

לפעמים פונקציה יכולה להחזיר **יותר מסוג אחד של ערך**.
כדי שפייתון (והעורך שלך) ידעו את זה מראש, נשתמש בהערות טיפוס(type hints)
מהמודול typing.

### Union – כשיש כמה אפשרויות

Union אומר: הפונקציה יכולה להחזיר אחד מכמה סוגים אפשריים.

```
from typing import Union

def parse_number(s: str) -> Union[int, float, None]:
    try:
        return int(s)
    except ValueError:
        try:
            return float(s)
        except ValueError:
            return None
```

כאן הפונקציה יכולה להחזיר int, או float, או None.

לדוגמה:

```
print(parse_number("42"))     # 42 (int)
print(parse_number("3.14"))   # 3.14 (float)
print(parse_number("hello"))  # None
```

## Optional – קיצור ל-Union עם None

אם אחד מהטיפוסים האפשריים הוא None, אפשר לכתוב Optional במקום Union[..., None].

```
from typing import Optional

def find_user(id: int) -> Optional[str]:
    """מחזיר שם משתמש או None אם לא נמצא."""
    return None
```

זה בעצם אותו דבר כמו:

```
Union[str, None]
```

רק קצר וברור יותר.

```
set_mode("train")  # תקין
set_mode("debug")  # ❌ טעות – לא אחד המצבים המותרים
```

## Literal – ערכים קבועים בלבד

Literal מאפשר להגביל פרמטר לערכים ספציפיים בלבד.
זה שימושי במיוחד כשיש פונקציה שפועלת בכמה **מצבים קבועים** מראש.

```
from typing import Literal

def set_mode(mode: Literal["train", "test", "eval"]) -> None:
    print(f"Running in {mode} mode")
```

כאן הערך של mode חייב להיות אחד משלושת המצבים האלה בלבד.

### למה זה חשוב

- זה עוזר לעורכים (כמו VS Code או PyCharm) להציע ערכים נכונים בלבד.
- זה מונע באגים טיפשיים עוד לפני שהקוד רץ.
- זה גם תיעוד מצוין, כל מי שקורא את הפונקציה מבין מיד מה הערכים האפשריים ומה היא מחזירה.

## עבודה עם מבני נתונים (List[str], Dict[str, int])

כדי לתאר אוספים של טיפוסים נשתמש בפרמטרים גנריים:

```
from typing import List, Dict

names: List[str] = ["דנה", "משה", "רותם"]
ages: Dict[str, int] = {"דנה": 32, "משה": 40}
```

או ישירות בתוך פונקציה:

```
def average_length(words: List[str]) -> float:
    total = sum(len(w) for w in words)
    return total / len(words)
```

כך כלים כמו mypy יכולים לדעת ש-words חייב להיות רשימה של מחרוזות,
ושגיאה תתגלה מוקדם יותר, עוד לפני שהקוד רץ.

## TypedDict – מבנה דמוי אובייקט מוגדר

כשאנחנו עובדים עם dictionaries שמתארים מבנה קבוע (למשל תגית של dataset),
נרצה לתעד בדיוק אילו שדות קיימים ואילו סוגים הם מחזיקים.

```
from typing import TypedDict

class User(TypedDict):
    name: str
    age: int
    active: bool

def describe_user(user: User) -> str:
    return f"{user['name']} ({user['age']}) - {'פעיל' if user['active']
else 'לא פעיל'}"

data: User = {"name": "דנה", "age": 30, "active": True}
print(describe_user(data))
```

TypedDict מאפשר לתעד מבני נתונים שנראים כמו JSON או מילונים ומונע בלבול בין מפתחות חסרים לסוגים לא נכונים.

## Protocols ו-Duck Typing – ממשקים ללא ירושה

בפייתון אין "ממשקים" רשמיים כמו ב-Java או #C.
לא צריך להצהיר שמחלקה **יורשת ממחלקת-אם מסוימת** כדי שתוכל לעבוד עם אחרת.
במקום זה, פייתון פועלת לפי עיקרון שנקרא **Duck Typing**.

## מה זה Duck Typing

הרעיון פשוט מאוד:
אם זה *מתנהג* כמו ברווז, אז מבחינת פייתון זה ברווז.

במילים אחרות, לא משנה מאיזו מחלקה האובייקט הגיע,
כל עוד יש לו את המתודות שהקוד שלך מצפה להן, זה מספיק.

```
class Dog:
    def speak(self):
        print("Woof!")

class RobotDog:
    def speak(self):
        print("BEEP-WOOF!")

def make_it_speak(dog):
    dog.speak()

make_it_speak(Dog())       # Woof!
make_it_speak(RobotDog())  # BEEP-WOOF!
```

שתי המחלקות לא יורשות זו מזו,
אבל שתיהן מתנהגות "כמו כלב" – יש להן את אותה מתודה speak.
וזה כל מה שפייתון צריכה כדי שזה יעבוד.

### אז איפה הבעיה?

בפרויקטים קטנים זה נחמד,
אבל בקוד גדול קשה לדעת מראש מה בדיוק נדרש מכל אובייקט.

אם מישהו יעביר לפונקציה אובייקט שאין לו את המתודה המתאימה,
פייתון תגלה את זה רק בזמן ריצה, ותזרוק שגיאה.

כאן נכנס לתמונה `Protocol`.

## Protocol – דרך לתעד איך אובייקט אמור להתנהג

`Protocol` מאפשר להגדיר **באופן פורמלי**
מה מצופה ממחלקה שתעבוד עם הקוד שלך
בלי לחייב אותה לרשת ממנה שום דבר.

```
from typing import Protocol

class Cleaner(Protocol):
    def clean(self, text: str) -> str:
        ...
```

זה אומר: כל אובייקט שיש לו מתודה בשם clean, שמקבלת מחרוזת ומחזירה מחרוזת, נחשב מתאים ל-Cleaner.

דוגמה:

```
class LowerCleaner:
    def clean(self, text: str) -> str:
        return text.lower()

def process_text(c: Cleaner, s: str) -> str:
    return c.clean(s)

processor = LowerCleaner()
print(process_text(processor, "HELLO"))  # hello
```

שים לב:

- LowerCleaner **לא יורשת** מ-Cleaner.
- ובכל זאת, היא **עומדת באותו חוזה** יש לה את המתודה הנדרשת.
- לכן הקוד עובד, והעורכים וכלי הבדיקה מזהים שזה תקין.

## למה זה טוב

- זה נותן לך **תיעוד ברור**: מה הפונקציה מצפה לקבל.
- כלי ניתוח סטטי (כמו mypy) **יכולים לבדוק אם מחלקה באמת עומדת בדרישות האלה**.

- אתה מקבל את הגמישות של פייתון, אבל גם את הביטחון של בדיקות טיפוסים סטטיות, כמו בשפות קשיחות יותר.

### בדיקה עם mypy

כדי לבדוק את הקוד שלך, התקן את הכלי `mypy`:

התקנה:

```
pip install mypy
```

הרצה:

```
mypy src/
```

אם תכתוב פונקציה שמוגדרת להחזיר `str` אבל תחזיר בפועל `int`,
או תעביר אובייקט שאין לו את המתודה הדרושה
`mypy` יתריע על כך עוד לפני ההרצה.

בדיקה סטטית אחת ביום שווה שעות של Debugging בהמשך.

## דוגמה מרכזית:

## type hints מלאים ל-mini_text_analyzer

נחזור לפרויקט הקטן שלנו mini_text_analyzer: ניקוי טקסט וחישוב סטטיסטיקות.

נראה איך נראה אותו קוד, רק עם טיפוסיות מלאה:

```
from pathlib import Path
from typing import List, Dict
import json

def read_text(path: Path) -> str:
    """קורא קובץ טקסט ומחזיר את תוכנו."""
    return path.read_text(encoding="utf-8")

def tokenize(text: str) -> List[str]:
    """מפרק טקסט למילים."""
    return text.split()

def word_stats(tokens: List[str]) -> Dict[str, int | float]:
    """מחזיר סטטיסטיקות בסיסיות על רשימת מילים."""
    lengths = [len(t) for t in tokens]
    return {
        "num_words": len(tokens),
        "avg_length": sum(lengths) / len(lengths) if lengths else 0,
    }
```

```
def save_json(path: Path, data: Dict[str, int | float]) -> None:
    """שומר נתונים כ-JSON."""
    path.write_text(json.dumps(data, ensure_ascii=False, indent=2), encoding="utf-8")

def main() -> None:
    text = read_text(Path("data/input.txt"))
    tokens = tokenize(text)
    stats = word_stats(tokens)
    save_json(Path("data/stats.json"), stats)
```

בכל מקום שבו הקוד "דיבר בעמימות", כעת יש הגדרה ברורה. גם המפתח הבא יוכל להבין בדיוק אילו סוגים נכנסים ויוצאים מכל פונקציה.

## Best Practices

- הוסף טיפוסים לכל פונקציה ציבורית (public).
- השתמש ב-Optional או | None כשיש החזרה מותנית.
- תעד גם משתנים גלובליים או חשובים (df: pd.DataFrame).
- השתמש ב-mypy או ב-pyright כחלק מה-CI שלך.
- אל תשתמש ב-Any בלי סיבה – הוא מבטל את היתרון.
- אל תעמיס Type Hints מיותרים בקוד פנימי קצר – שמור על קריאות.

## סיכום – איך type hints מגדילים אמינות

רמזי טיפוס לא משנים את פייתון, אבל הם משנים את **הדרך שבה אנו חושבים עליה**.

הם יוצרים חוזה ברור בין פונקציות, מסייעים ל-IDE להשלים קוד בצורה מדויקת,
ומאפשרים למערכות גדולות, כמו פרויקטי AI מבוזרים לשמור על עקביות גם כשהן מתפתחות במהירות.

פייתון תישאר שפה דינמית, אבל עם typing נכון היא הופכת לשפה **אמינה, מתועדת ומובנת הרבה יותר**.

# פרק 11 – דקורטורים, Context Managers ו-dataclass

## כלים שמקצרים ומפשטים קוד

פייתון נבנתה מתוך תפיסה של **קריאות, פשטות ואלגנטיות**.
אבל ככל שהמערכת גדלה, גם בקוד היפה ביותר מופיעות חזרות, טיפול בשגיאות שחוזר על עצמו, וניהול משאבים שדורש סדר ומשמעת.

בדיוק בשביל זה קיימים הכלים הבאים:

- **דקורטורים (Decorators)**
  פונקציות שעוטפות פונקציות אחרות, מוסיפות להן יכולת או לוגיקה, בלי לגעת בקוד הפנימי.
- **Context Managers**
  מבנים שמנהלים משאבים (כגון: קבצים, חיבורים, או טרנזקציות) בצורה בטוחה עם תחביר פשוט.
- **Dataclasses**
  דרך תמציתית להגדיר מחלקות נתונים מבלי לחזור על boilerplateאינסופי.

אלה לא רק כלים "חכמים", הם הדרך הפייתונית לכתוב קוד **נקי, חכם וקצר**, מבלי לוותר על קריאות.

## דקורטורים – פונקציה עוטפת פונקציה (@decorator)

לפעמים נרצה **להוסיף התנהגות** לפונקציה קיימת
בלי לגעת בקוד שלה.

למשל:
להדפיס הודעות, למדוד זמן ריצה או לבדוק הרשאות.
במקום לשכפל את אותה לוגיקה שוב ושוב,
משתמשים ב-**דקורטור**.

### הרעיון פשוט

דקורטור הוא **פונקציה שעוטפת פונקציה אחרת**.
מוסיפה לה קוד לפני ואחרי,
ומחזירה פונקציה חדשה.

דוגמה

```
def logger(func):
  def wrapper(*args, **kwargs):
    print(f"קורא ל-{func.__name__} עם{args}")
    result = func(*args, **kwargs)
    print(f"{func.__name__} החזירה{result}")
    return result
  return wrapper
```

כאן:

- `logger` היא הדקורטור.

- wrapper היא הפונקציה החדשה שעוטפת את הפונקציה המקורית.

## איך משתמשים בזה

במקום לכתוב:

```
add = logger(add)
```

אפשר פשוט להשתמש בתחביר הקצר של דקורטורים:

```
@logger
def add(a: int, b: int) -> int:
    return a + b

add(3, 5)
```

הסימן @logger אומר לפייתון:

"לפני שאת שומרת את הפונקציה add, תעבירי עליה דרך logger."

מאחורי הקלעים, פייתון עושה בעצם:

```
add = logger(add)
```

## מה באמת קורה בזמן הריצה

כשאנחנו קוראים:

```
add(3, 5)
```

פייתון לא מריצה את add המקורית,

אלא את הפונקציה wrapper שחוזרת מהדקורטור.

וזה רצף הפעולות בפועל:

1. השורה הראשונה ב-wrapper רצה:

```
add -קורא ל עם (3, 5)
```

2. עכשיו wrapper מריצה את add(a, b) המקורית ומקבלת את הערך 8.
3. אחר כך מגיעה ההדפסה השנייה:

```
add החזירה 8
```

4. לבסוף, wrapper מחזירה את הערך 8.

## הפלט האמיתי שיודפס למסך

```
add -קורא ל עם (3, 5)
add החזירה 8
```

## למה זה טוב

- מוסיף פונקציונליות בלי לשנות את הקוד המקורי.
- מאפשר שימוש חוזר בלוגיקה דומה (כמו לוגים, מדידת זמן, הרשאות וכו').
- שומר על קוד נקי ומודולרי.

## דקורטורים עם פרמטרים (@repeat(n=3))

אם נרצה שדקורטור יקבל פרמטרים, נעטוף אותו בשכבה נוספת:

```
from functools import wraps

def repeat(n: int):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for i in range(n):
                print(f"קריאה #{i+1}")
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(3)
def say_hello():
    print("שלום!")

say_hello()
```

שימוש ב-@wraps שומר על שם הפונקציה וה-docstring המקוריים,

כדי שלא נאבד מידע חשוב על הפונקציה המקורית.

## שימושים נפוצים (@lru_cache, @measure_time)

פייתון כוללת דקורטורים מובנים ושימושיים מאוד, ביניהם:

```
from functools import lru_cache
import time

@lru_cache(maxsize=100)
def fibonacci(n: int) -> int:
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# דוגמה נוספת – מדידת זמן ריצה
def measure_time(func):
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        duration = time.perf_counter() - start
        print(f"{func.__name__} רצה במשך{duration:.4f} שניות")
        return result
    return wrapper

@measure_time
def slow_sum():
    time.sleep(1)
    return sum(range(100000))

slow_sum()
```

@lru_cache מאיץ חישובים יקרים על ידי שמירת תוצאות.
@measure_time הוא דוגמה לדקורטור מותאם אישית שיכול לעזור לנתח ביצועים.

## with – Context Managers ו-enter/exit

כשתהליך דורש פתיחה וסגירה של משאב (כגון: קובץ, חיבור רשת או טרנזקציה),
Context Manager מאפשר לו להתנהל אוטומטית:

```
class FileHandler:
    def __init__(self, path: str):
        self.path = path
        self.file = None

    def __enter__(self):
        self.file = open(self.path, "w", encoding="utf-8")
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.file.close()

with FileHandler("data/output.txt") as f:
    f.write("שלום עולם!")
```

כשה-with מסתיים, המתודה __exit__ מופעלת תמיד, גם אם נזרקה חריגה.

אין צורך לזכור לסגור קובץ או לשחרר משאב, זה מתבצע אוטומטית.

### contextlib – הפשטה עם contextmanager@

אם אין צורך במחלקה שלמה, אפשר להשתמש בדקורטור @contextmanager
כדי לכתוב Context Manager קצר וברור יותר.

```
from contextlib import contextmanager

@contextmanager
def open_utf8(path: str, mode: str = "r"):
  f = open(path, mode, encoding="utf-8")
  try:
    yield f
  finally:
    f.close()

with open_utf8("data/test.txt", "w") as f:
  f.write("💡 טקסט בעברית באיכות גבוהה")
```

הקוד הזה עושה בדיוק אותו דבר, אבל בלי לכתוב מחלקה.
זהו פתרון אלגנטי למקרי ניהול משאבים פשוטים.

### @dataclass – קיצור למחלקות נתונים

הדקורטור @dataclass (שהופיע כבר בפרקים הקודמים) הוא למעשה דוגמה מובהקת לשימוש בדקורטור ברמת מחלקה.

```
from dataclasses import dataclass

@dataclass
class Point:
    x: float
    y: float

p = Point(1.0, 2.5)
print(p)  # Point(x=1.0, y=2.5)
```

הוא יוצר אוטומטית את כל מה שצריך:

`__init__`, `__repr__`, `__eq__`

ומאפשר לכתוב מחלקות פשוטות, נקיות וברורות. בלי קוד מיותר.

## דוגמה מרכזית: measure_time + @dataclass Result

נשלב בין שני עולמות: נמדוד זמן ריצה של תהליך, ונשמור את התוצאה באובייקט נתונים נוח.

```
import time
from dataclasses import dataclass
from typing import Any, Callable

def measure_time(func: Callable[..., Any]):
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
```

```python
        result = func(*args, **kwargs)
        duration = time.perf_counter() - start
        return Result(func.__name__, duration, result)
    return wrapper

@dataclass
class Result:
    name: str
    duration: float
    output: Any

@measure_time
def heavy_computation(n: int) -> int:
    time.sleep(0.8)
    return sum(i * i for i in range(n))

res = heavy_computation(100_000)
print(res)
# Result(name='heavy_computation', duration=0.8012,
output=333328333350000)
```

שימו לב כמה קריא הקוד:

- הדקורטור מודד את הזמן.
- ה-dataclass מאגד את כל הנתונים.
- והפונקציה עצמה נשארת נקייה ופשוטה.

## Best Practices

- השתמשו בדקורטורים **רק כשיש ערך ברור** לא כדי להרשים את הצוות.
- השתמשו ב-`@wraps` כדי לשמור על מטא-מידע של הפונקציה המקורית.
- השתמשו ב-Context Manager לכל משאב שדורש סגירה בטוחה.
- העדיפו `@contextmanager` על מחלקה מלאה במקרים פשוטים.
- אל תגזימו בשימוש ב-dataclass השתמשו, רק כשמדובר בנתונים פשוטים ללא לוגיקה מורכבת.

## סיכום – איך הכלים האלה הופכים קוד לפייתוני

דקורטורים, Context Managers ו-dataclass אינם קסמים, הם פשוט תחביר שמאפשר **לחשוב ברמה גבוהה יותר**.
הם מסירים חזרות, מנהלים משאבים בביטחון, ומשאירים את הלוגיקה העסקית ממוקדת.

כשמשתמשים בהם נכון, הקוד שלך נראה פחות כמו רצף של פעולות,

ויותר כמו **שפה טבעית שמתארת את כוונתך**.
וזה, בדיוק הרגע שבו פייתון מפסיקה להיות רק שפה,
והופכת לכלי ביטוי הנדסי אמיתי.

## pytest: התקנה, מבנה קבצים, assert בסיסי

Pytest הוא הסטנדרט בפייתון לבדיקות יחידה (unit tests).
קל להשתמש בו, אינו דורש מחלקות או boilerplate, ויודע לזהות
אוטומטית כל קובץ שמתחיל ב-_test.

התקנה:

```
pip install pytest
```

מבנה תיקיות טיפוסי:

```
project/
├──── src/
│     └──── mini_text_analyzer/
│           ├──── __init__.py
│           └──── text_utils.py
└──── tests/
      └──── test_text_utils.py
```

בדיקה פשוטה:

```
from mini_text_analyzer.text_utils import tokenize

def test_tokenize_basic():
    text = "שלום עולם"
    tokens = tokenize(text)
    assert tokens == ["שלום", "עולם"]
```

הרצה:

```
pytest -v
```

אם הבדיקה נכשלת, `pytest` יציג בדיוק איזו השוואה נכשלה, בלי לוגים מיותרים.

# פרק 12 – בדיקות אוטומטיות וארגונומיה למפתח

## למה בדיקות קריטיות בפרויקטי AI

בעולם של AI, כל שינוי קטן

ספרייה חדשה, פרמטר נוסף, או מודל משודרג.

עלול לשנות תוצאות בצורה לא צפויה.

ולכן, בניגוד לקוד "רגיל", כאן **בדיקות אינן מותרות**, הן **חומת הגנה הכרחית**.

מודלי שפה, Pipelines לעיבוד נתונים, ותהליכי אימון, כולם מלאים באלמנטים הסתברותיים.

אי-אפשר להבטיח תוצאה זהה בכל הרצה, אבל כן ניתן לוודא שהמערכת מתנהגת כראוי, שומרת על מבנה תקין, ומגיבה נכון לשגיאות.

בדיקות טובות בפרויקטי AI לא נמדדות רק ב-"עבר/נכשל", אלא גם ביכולת שלהן **לאתר התנהגויות לא צפויות מוקדם**, ולאפשר למפתח לעבוד בביטחון.

זו לא עוד משימה, זו **שיטת חשיבה הנדסית**.

## Fixtures: הכנה משותפת לבדיקות

Fixtures הן פונקציות שמכינות נתונים או סביבה לבדיקה, ומוחזרות לבדיקה אוטומטית לפי שם.

```
import pytest
from mini_text_analyzer.text_utils import tokenize

@pytest.fixture
def sample_text() -> str:
    return "פייתון היא שפה מדהימה"

def test_tokenize_with_fixture(sample_text):
    tokens = tokenize(sample_text)
    assert len(tokens) == 4
```

אפשר להגדיר Fixtures כלליים בקובץ `conftest.py` כדי לשתף אותם בכל הפרויקט.
הם מעולים להכנות חוזרות כמו פתיחת קובץ, יצירת אובייקט API, או ניקוי נתונים.

## בדיקות חריגות (pytest.raises)

נרצה לוודא שגם במקרים חריגים הפונקציה מתנהגת כמצופה. כלומר, זורקת את החריגה הנכונה.

```
import pytest
from mini_text_analyzer.io_utils import read_json

```

```
def test_read_json_not_found():
    with pytest.raises(FileNotFoundError):
        read_json("data/missing.json")
```

בדיקה כזו אינה נועדה "להפיל" את הקוד, אלא לוודא שהתנהגות השגיאה צפויה, מתועדת וניתנת ללכידה.

### Mocking – סימולציה של API חיצוני

לא תמיד נרצה לגשת לשירות חיצוני אמיתי בזמן הבדיקות (כמו OpenAI API או Google Cloud).
במקום זאת, נשתמש ב-Mock, אובייקט שמדמה התנהגות אמיתית.

```
from unittest.mock import patch
from mini_text_analyzer.llm_client import query_model

@patch("mini_text_analyzer.llm_client.send_request")
def test_query_model(mock_send):
    mock_send.return_value = {"text": "שלום עולם"}
    result = query_model("hi")
    assert "שלום" in result
```

כך אנו בודקים את הלוגיקה שלנו בלי תלות ברשת או ב-API אמיתי.
גישה זו חיונית במיוחד במערכות מבוססות AI שבהן הגישה החיצונית איטית או עולה כסף.

## כלים משלימים (black, ruff, pre-commit)

בדיקות הן לא רק "בדיקת תוצאה", הן חלק מתרבות של **איכות קוד ונוחות פיתוח**.

שימוש בכלים חכמים שמקלים על המפתח, שומרים על אחידות בקוד, ומונעים בעיות עוד לפני שהן קורות.

כלים משלימים יכולים להפוך את הסביבה שלך לחכמה ואחידה:

- **Black**: מעצב קוד אוטומטית לפי תקן אחיד.
- **Ruff**: מנתח סגנון (lint) ומזהה בעיות בזמן כתיבה.
- **pre-commit**: מריץ בדיקות לפני כל commit כדי למנוע טעויות מראש.

דוגמה לקובץ .pre-commit-config.yaml:

```
repos:
 - repo: https://github.com/psf/black
   rev: 24.4.0
   hooks:
     - id: black
 - repo: https://github.com/astral-sh/ruff-pre-commit
   rev: v0.6.3
   hooks:
     - id: ruff
 - repo: https://github.com/pre-commit/mirrors-mypy
   rev: v1.10.0
```

```
  hooks:
    - id: mypy
```

## דוגמה מרכזית: בדיקות ל-tokenize ו-clean

נבנה בדיקות אמיתיות לשתי פונקציות מהפרקים הקודמים:

```python
from mini_text_analyzer.text_utils import tokenize, normalize

def test_tokenize_simple():
    text = "שלום עולם"
    result = tokenize(text)
    assert result == ["שלום", "עולם"]

def test_normalize_lowercase():
    text = "   פייתון"
    result = normalize(text)
    assert result == "פייתון"
```

אפשר גם לבדוק קלט בעייתי:

```python
import pytest

def test_tokenize_empty():
    assert tokenize("") == []

def test_normalize_non_string():
    with pytest.raises(AttributeError):
        normalize(None)
```

בדיקות קטנות, ממוקדות וברורות, הרבה יותר יעילות מבדיקה אחת ענקית שמנסה לבדוק את הכול.

## שילוב ב-CI/CD (GitHub Actions)

בדיקות אוטומטיות הופכות משמעותיות באמת כשהן רצות לבד, בכל פעם שמישהו מבצע push או pull request.

דוגמה פשוטה ל-GitHub Actions:

```yaml
name: Tests

on: [push, pull_request]

jobs:
 test:
   runs-on: ubuntu-latest
   steps:
    - uses: actions/checkout@v4
    - name: Set up Python
      uses: actions/setup-python@v5
      with:
       python-version: '3.12'
    - run: pip install -r requirements.txt
    - run: pytest -v
```

כך כל שינוי בקוד עובר בדיקה אוטומטית, בלי שמפתח צריך לזכור להריץ משהו ידנית.

## Best Practices

- כתוב בדיקות **קטנות, ממוקדות וברורות**.
- לפחות בדיקה אחת, עבור כל פונקציה ציבורית.
- ודא שכל קוד שגיאה נבדק עם `pytest.raises`.
- השתמש ב-Fixtures למידע שחוזר על עצמו.
- הפרד בין **Unit Tests** (בודקים פונקציה אחת) לבין **Integration Tests** (בודקים תהליך מלא).
- ודא שהבדיקות שלך **חיוביות** (מה אמור לעבוד) וגם **שליליות** (מה אמור להיכשל).
- שמור על זמן ריצה קצר, כך תוכל להריץ בדיקות לעיתים קרובות.

## סיכום – בדיקות הן חלק מהפיתוח, לא "עוד משימה"

בדיקות טובות לא נועדו להרשים את הבודקים, אלא להגן עליך, המפתח.
במערכות AI שבהן הכל משתנה מהר, רק בדיקות עקביות שומרות על יציבות.

כשבדיקות רצות בכל commit, כש-pre-commit שומר על סגנון, וכש-CI בודק הכול אוטומטית.

אתה לא רק כותב קוד, אתה **בונה מערכת אמינה**.

בדיקות אינן "שלב הסיום", אלא **היסוד של תהליך הפיתוח עצמו**.
ומי שמבין את זה, כבר מתכנת ברמה של מהנדס.

# פרק 13 – ביצועים, זיכרון וקצת NumPy

## למה ביצועים חשובים גם בפייתון

פייתון נחשבת לשפה איטית רק בעיני מי שמודד אותה **באופן שגוי**.
היא לא נועדה לנצח בתחרות על **האלגוריתם המהיר ביותר**,
אלא לאפשר למפתח **חופש לחשוב, להתנסות ולבנות במהירות**.

אלא שבעולם ה-**AI**, שום דבר כבר איננו קטן.
כל טקסט עשוי להכיל **ג׳יגה-בייטים של מידע**,
כל **מערך נתונים (dataset)** מתנהג כמו יקום שלם,
ומודל אחד תמים לכאורה עלול לצרוך **עשרות ג׳יגה-בייטים של זיכרון**.
לא דרוש הרבה כדי שמחשב נייד יתחיל להשמיע **קולות של מנוע סילון** באמצע האימון.

במילים אחרות: הבעיה אינה טמונה בפייתון עצמה,
אלא **באופן שבו משתמשים בה**.
כאשר מתייחסים אליה כשפה סקריפטית בלבד, היא מתנהגת בהתאם;
אך כשמשלבים בה את **הכלים הנכונים**

Generators, NumPy, ו-Profilers

היא הופכת ל-**כלי הנדסי של ממש**.

מטרת הפרק הזה איננה "לסחוט עוד שלושה אחוזים של מהירות",
אלא להבין **היכן באמת מתבזבז הזמן**,
מה גורם **לעומס על הזיכרון**,
ואיך לגרום לפייתון לעבוד **בצורה חכמה ויעילה יותר**.

בסופו של דבר, ביצועים אינם רק עניין של מספרים
הם ההבדל בין **קוד שמגיב מיידית**,
לבין **קוד שמזכיר לך שהמאוורר במחשב עדיין עובד**.

## מדידת זמן ריצה – כי ניחושים לא משפרים ביצועים

לפני שמייעלים קוד, צריך לדעת **מה באמת איטי**.
המודול `timeit` נועד בדיוק לזה.
למדוד זמן ריצה בצורה אמינה, נקייה ובלתי תלוית-מערכת.

```
import timeit

code = "result = [x**2 for x in range(10_000)]"
print(timeit.timeit(code, number=100))
```

`timeit` מריץ את הקוד שוב ושוב ומחזיר **ממוצע זמן ריצה מדויק**.
כך ניתן להשוות בין גרסאות של פונקציה ולראות איזו מהן **באמת מהירה יותר**, לא לפי תחושת בטן, אלא לפי נתונים.

דוגמה קטנה:

```
setup = "nums = list(range(1000))"
```

```
v1 = "sum([x**2 for x in nums])"
v2 = "sum(x**2 for x in nums)"  # Generator, לא יוצר רשימה שלמה בזיכרון

print(timeit.timeit(v1, setup=setup, number=1000))
print(timeit.timeit(v2, setup=setup, number=1000))
```

במבחן אמיתי, תופתע לגלות שהגרסה הקצרה יותר **לא רק נקייה וברורה יותר**, אלא גם **מהירה וחסכונית בזיכרון**
בזכות השימוש ב-**Generator Expression**,
שמחשב ערכים תוך כדי תנועה במקום ליצור רשימה שלמה מראש.

מדידת זמן ריצה היא לא רק שלב לפני אופטימיזציה, היא כלי למחשבה.
ברגע שמודדים באופן עקבי, מתחילים לזהות תבניות:
אילו פעולות באמת יקרות, ואילו רק נראות כך.
מפתח שמודד, כותב קוד מדויק יותר, ולא מהיר "במקרה".

## ניתוח פרופיל: לזהות את צוואר הבקבוק

לפעמים הקוד כולו מרגיש "איטי", אבל האמת היא שפונקציה אחת גונבת את כל הזמן.
כדי למצוא אותה, יש את `cProfile`:

```
import cProfile
```

```
def compute():
    return sum(i * i for i in range(100_000))

cProfile.run("compute()")
```

הפלט יגיד לך כמה פעמים כל פונקציה נקראה וכמה זמן היא לקחה.

אם אחת מהן אחראית ל-80% מהזמן, מצאת את הבעיה.

רוצה לראות את זה יפה? התקן `snakeviz`:

```
pip install snakeviz
python -m cProfile -o out.prof myscript.py
snakeviz out.prof
```

ותקבל גרף צבעוני של זמן הריצה שלך.

## עבודה חכמה עם זיכרון – לא כל דבר צריך רשימה

לפעמים הבעיה אינה המהירות, אלא **הזיכרון**.

אפשר לכתוב פונקציה שרצה מהר, אבל אם היא יוצרת ברשימה אחת מיליון איברים,

המחשב שלך עלול "להתנפח" עד כדי האטה או קריסה.

כאן נכנסים לתמונה **Generators**

פונקציות שמחזירות ערכים **אחד-אחד**, לפי הצורך,

במקום לבנות רשימה שלמה מראש.

```
def squares():
    for i in range(1_000_000):
        yield i ** 2

for n in squares():
    if n > 100:
        break
```

הפונקציה הזו לא שומרת מיליון ערכים בזיכרון.

היא פשוט מחשבת כל ערך כשצריך אותו

וזה חוסך המון זיכרון, בלי לפגוע בפשטות הקוד.

אפשר לחשוב על Generator כעל **צינור שמזרים נתונים**,

במקום **דלי שמחזיק הכול מראש**.

ובעולם ה-**AI**, זו לא רק יעילות, זו לעיתים **תנאי הכרחי**:

כך אפשר לקרוא datasets עצומים ולנתח אותם בהדרגה,

מבלי להעמיס את כל המידע על הזיכרון בבת אחת.

## NumPy בפועל – מערכים ופעולות וקטוריות חכמות

אם אתה עובד עם מספרים, טבלאות או מטריצות. תכיר את **NumPy**.

זו הספרייה שהפכה את פייתון משפה נוחה אך איטית לשפה שיכולה לרוץ כמעט כמו C.

איך היא עושה את זה?

פשוט:

במקום לשמור רשימת אובייקטים בזיכרון, כמו שפייתון רגילה עושה, NumPy שומרת **בלוק אחד צפוף של נתונים** (ב-C).

כשאתה מבצע חישוב, היא מפעילה את הפעולה על כל הבלוק בבת אחת

בלי לולאה אחת בפייתון.

```
import numpy as np

a = np.arange(1_000_000)
b = a * 2  # פעולה וקטורית – פי עשרות מהירה מלולאה רגילה
```

במקום לעבור איבר-איבר, המעבד מבצע את כל ההכפלה "בבאטץ' אחד".

זו בדיוק **וקטוריזציה (Vectorization)**

היכולת לבצע פעולה אחת על מערך שלם,

באמצעות הוראות מעבד שמטפלות בכמה ערכים בו-זמנית.

## דוגמה

```
import numpy as np

x = np.array([1, 2, 3])
y = np.array([4, 5, 6])

print(x + y)      # [5 7 9]
print(x * y)      # [ 4 10 18]
print(np.mean(x)) # 2.0

m = np.random.rand(3, 3)
print(np.linalg.inv(m))  # הופכי מטריצה
```

במבט ראשון זה נראה כמו קוד רגיל.

חיבור רשימות, כפל איברים, חישוב ממוצע.

אבל מתחת לפני השטח, NumPy **לא רצה בלולאה אחת**.

היא שומרת את כל הנתונים במערך צפוף (array) בזיכרון,

ומעבירה את הפעולה כולה לקוד C יעיל,

שמבצע אותה על כל הנתונים יחד, בלי הפרשנות האיטית של פייתון.

כך זה נראה בגרסה "רגילה" לעומת גרסה וקטורית:

```
# גרסה רגילה
result = []
for i in range(len(x)):
```

```
    result.append(x[i] + y[i])

# גרסה וקטורית (NumPy)
result = x + y
```

התוצאה זהה, אבל זמן הריצה שונה לגמרי.

## למה זה כל כך מהיר

במילים פשוטות:

- **רשימות פייתון:** אוסף של אובייקטים נפרדים, כל אחד במקום אחר בזיכרון.
- **מערכי NumPy:** בלוק רציף וצפוף של נתונים שמיוצגים כמספרים "טהורים".

המעבד יודע לגשת ישירות לבלוק הזה ולעבד אותו בפעולה אחת (SIMD),

מה שמדלג על כל ה-overhead של פייתון

ובמקרים רבים, מביא למהירות גבוהה פי 50–100.

זה לא טריק, זו **ארכיטקטורה חכמה**:

לתת לשפת פייתון את המוח של C, בלי לאבד את הפשטות של פייתון.

## דוגמה מרכזית: תדירויות מילים (נאיבי מול NumPy)

ניקח תרגיל קלאסי: ספירת תדירויות מילים.

### גישה נאיבית:

```
from collections import Counter

def word_freq_naive(words: list[str]) -> dict[str, int]:
    return dict(Counter(words))
```

### גישה וקטורית עם NumPy:

```
import numpy as np

def word_freq_numpy(words: list[str]) -> dict[str, int]:
    arr = np.array(words)
    unique, counts = np.unique(arr, return_counts=True)
    return dict(zip(unique, counts))
```

ב-dataset קטן זה לא משנה.

אבל כשיש לך מיליון מילים,

הגרסה של NumPy תרוץ פי 5–10 מהר יותר,

בלי לשנות שורה של לוגיקה.

## Best Practices

- **מדוד לפני שאתה משפר** בלי נתונים, זו רק אינטואיציה.
- **השתמש ב-NumPy** לכל חישוב מתמטי רציני.
- **השתמש ב-generators** כשקוראים קבצים או עובדים עם זרמים גדולים.
- **הימנע מהעתקות מיותרות של נתונים.**
- **תעדף וקטוריזציה** על פני לולאות.
- **תעדף פשטות על פני "אופטימיזציה חכמה"** רק אם יש בעיה אמיתית, פותרים אותה.

## סיכום – איך להגיע לביצועים גבוהים בפייתון

פייתון לא נועדה לנצח בתחרות על המהירות הגולמית.
אבל עם קצת הבנה של הכלים הנכונים, היא מסוגלת לרוץ מהר, ממש מהר.

מדוד עם timeit, חפור עם cProfile,
חסוך בזיכרון עם generators, והאץ כל חישוב עם NumPy.

התוצאה: קוד נקי, קריא, ועם ביצועים שמפתיעים גם את הספקנים הכי גדולים.
כי בסוף, לא מדובר בלהיות "מהיר", אלא בלהיות **יעיל**.
וזה בדיוק מה שמבדיל בין מתכנת לפייתוניסט אמיתי.

מאת: תומר קדם

# פרק 14 – Pandas למהנדסי AI

## למה Pandas (ולא “רק” NumPy או dict)

בוא נדבר רגע על המקום שבו רוב מהנדסי ה-AI נופלים בהתחלה:
הם מתחילים לנתח נתונים עם **NumPy** או אפילו עם רשימות ו-dicts של פייתון, וזה עובד מצוין... עד שזה כבר לא.

במבט ראשון, נראה ש-NumPy נותן לך הכול:
מערכים מהירים, פעולות וקטוריות, חישוב מטריצות, ויעילות כמעט כמו בקוד C.
אבל כשאתה עובד עם **נתונים אמיתיים**, לא מטריצות סטריליות אלא קבצי CSV, טקסטים, מזהים, ערכים חסרים, תאריכים, סוגים שונים של עמודות פתאום זה מתחיל להרגיש כמו לאפות עוגה עם מברג.

ופה נכנסת **Pandas**.

## Pandas = השכבה האנושית של הנתונים

אפשר לחשוב על Pandas כעל **עטיפה אינטואיטיבית ל-NumPy**, שמבינה איך מתכנתים באמת עובדים עם מידע.

במקום להתעסק במיקומים ובממדים, אתה עובד עם **שמות עמודות** ו**שורות** ממש כמו בגיליון Excel, רק עם כוח של קוד.

```
import pandas as pd
```

```python
# קריאת קובץ CSV ישירות ל-DataFrame
df = pd.read_csv("data/users.csv")

# מבט ראשון על הנתונים
print(df.head())

# סינון לפי תנאי לוגי
active = df[df["is_active"] == True]

# ספירת משתמשים פעילים
print("משתמשים פעילים:", len(active))
```

בשלוש שורות אתה עושה מה שב-NumPy היה דורש מערך דו-ממדי, חישוב אינדקסים, והמרת טיפוסים.
ופה בדיוק הכוח: **Pandas נבנתה סביב מודל החשיבה של מהנדס הנתונים**, לא סביב מבני הזיכרון של המעבד.

## למה לא להסתפק ב-dict או list?

רשימות ו-dicts הם מושלמים כשמדובר באובייקטים בודדים או אוספים קטנים.
אבל ברגע שהנתונים שלך מגיעים **ממקור חיצוני** (כמו CSV או JSON) ואתה רוצה:

- למיין לפי עמודה
- לסנן לפי תנאי

- לחשב ממוצעים או סטיות תקן
- לאחד datasets שונים
- להתמודד עם NaN או סוגי נתונים מעורבים

אתה מגלה מהר מאוד ש-dict הוא לא טבלה, הוא מבוך.

ניקח דוגמה קטנה:

```python
users = [
    {"name": "דנה", "age": 29, "city": "תל אביב"},
    {"name": "רועי", "age": 34, "city": "חיפה"},
    {"name": "נועה", "age": None, "city": "ירושלים"},
]

# לחשב ממוצע גיל?
ages = [u["age"] for u in users if u["age"] is not None]
print(sum(ages) / len(ages))
```

לעומת זאת, ב-Pandas:

```python
import pandas as pd

df = pd.DataFrame(users)
print(df["age"].mean())
```

שורה אחת, בלי לולאות, בלי בדיקות, בלי טעויות.

והכי חשוב, אותה פקודה תעבוד גם על **מיליון שורות**, עם אותה יעילות וקריאות.

## Pandas בעולם ה-AI

בעולם של AI, **Pandas היא נקודת המעבר בין העולם הגולמי לבין העולם הלמידה.**

היא הגשר שבין "קובץ לא מנוקה" לבין "מערך אימון מוכן להזנה למודל".

בין אם אתה עוסק ב-NLP, בראייה ממוחשבת או ב-RAG, תצטרך בשלב כלשהו:

- לקרוא קבצי טקסט או מטא-דטה
- לנקות ולסנן נתונים
- למזג מידע ממקורות שונים
- לייצא את התוצאה ל-Parquet או ל-JSON מוכן לאימון

כל זה קורה ב-Pandas.

## Series ו-DataFrame: הבסיס

ב-Pandas קיימים שני מבני נתונים בלבד שצריך להבין לעומק:
**Series** ו-**DataFrame**.
השניים האלו הם הליבה של כל מניפולציה על נתונים.

### Series

Series מייצגת עמודה בודדת עם אינדקס.
היא דומה לרשימה, אבל שומרת הקשר בין מפתח לערך.

```
import pandas as pd

ages = pd.Series([29, 34, 41], index=["דנה", "רועי", "הילה"])
```

אפשר לבצע עליה פעולות מתמטיות ולוגיות ישירות:

```
ages.mean()      # ממוצע
ages[ages > 30]   # סינון לפי תנאי
```

במונחים של NumPy, זו עטיפה וקטורית עם שמות.
במונחים של מפתח, זו הדרך לחשוב על עמודת נתונים ולא על מערך.

### DataFrame

DataFrame הוא אוסף של Series עם אותו אינדקס.
זהו מבנה דו-ממדי שמאפשר לעבוד עם נתונים כמו בטבלה, אבל עם ביצועים של מערך.

```
df = pd.DataFrame({
  "name": ["דנה", "רועי", "הילה"],
  "age": [29, 34, 41],
  "city": ["תל אביב", "חיפה", "ירושלים"]
})
```

ה-index נוצר אוטומטית, אך אפשר להגדיר אינדקס סמנטי:

```
df = df.set_index("name")
df.loc["דנה"]
```

כל עמודה היא Series עצמאית, וכל שורה מייצגת ישות.

המודל הזה חזק במיוחד כשמתייחסים לנתונים כ-features ו-labels.

### שימוש בעולם ה-AI

בכל שלב של הכנת dataset, ניתוח טקסט, ניקוי נתונים, או תיוג דוגמאות תעבוד על DataFrame אחד או כמה.

לדוגמה:

```
reviews = pd.DataFrame({
  "text": ["מוצר מצוין", "משלוח איטי", "שרות מדהים"],
  "sentiment": [1, 0, 1]
})

reviews[reviews["sentiment"] == 1]["text"]
```

כך נראה קוד אמיתי במערכת NLP.

אין צורך בלולאות או רשימות ביניים, הכול מבוסס פעולות וקטוריות.

## קריאה וכתיבה: CSV, JSON, Excel, Parquet

העבודה עם Pandas מתחילה כמעט תמיד בשאלה אחת:
איך לטעון נתונים, ואיך לשמור תוצאות.
הספרייה מספקת ממשק אחיד לכל הפורמטים הנפוצים.

### CSV

פורמט פשוט ונפוץ במיוחד.
מומלץ להגדיר תמיד קידוד UTF-8 כדי למנוע בעיות עם טקסטים בעברית.

```
import pandas as pd

df = pd.read_csv("data/users.csv", encoding="utf-8")
df.head()
```

ב-Pandas, כל עמודה מקבלת טיפוס נתון (dtype) אוטומטית.
כאשר עובדים עם datasets גדולים,
כדאי **להגדיר את ה-dtype ידנית** כדי לחסוך זיכרון ולשפר ביצועים.
לדוגמה:

```
df = pd.read_csv(
    "data/users.csv",
    encoding="utf-8",
    dtype={"id": "int32", "age": "float32", "is_active": "bool"}
)
```

הסיבה פשוטה: ברירת המחדל של Pandas משתמשת בטיפוסים רחבים (int64, float64),
מה שעלול להכפיל את צריכת הזיכרון על קבצים גדולים.
הגדרה מפורשת מאפשרת גם טעינה מדויקת יותר, בעיקר כשיש ערכים חסרים או עמודות בוליאניות.

## JSON

פורמט אידיאלי לעבודה עם **APIs**,
לוגים ונתונים חצי-מובְנים (Semi-Structured).
Pandas יודעת לטעון ישירות קובץ JSON שמכיל רשימה של אובייקטים:

```
import pandas as pd

df = pd.read_json("data/users.json")
```

כאשר מבנה הנתונים מקונן (nested), נדרש **שיטוח (Normalization)** כדי להפוך את הנתונים לטבלה שטוחה.
במקום לכתוב קוד רקורסיבי, משתמשים ב-pd.json_normalize:

```
import pandas as pd

data = [
  {"id": 1, "user": {"name": "דנה", "city": "תל אביב"}},
  {"id": 2, "user": {"name": "רועי", "city": "חיפה"}}
]
```

```
df = pd.json_normalize(data)
```

פלט:

```
  id user.name user.city
0  1     תל אביב   דנה
1  2     חיפה      רועי
```

הפונקציה יוצרת עמודות עם שמות היררכיים (user.name, user.city) ומאפשרת לעבד את המידע בדיוק כמו DataFrame רגיל.

במערכות AI זה שימושי במיוחד כשמייבאים תוצאות של APIs (כמו GPT או OpenAI Embeddings) שמחזירים מבנים מקוננים.

## Excel

שימושי במיוחד כשמקור הנתונים מגיע מצוות עסקי או ממערכת דיווח.

כל גיליון (Sheet) בקובץ ניתן לטעינה בנפרד:

```
import pandas as pd

df = pd.read_excel("data/sales.xlsx", sheet_name="2025_Q1")
```

אם הקובץ מכיל כמה גיליונות, ניתן לטעון את כולם כ-dict של DataFrames:

```
sheets = pd.read_excel("data/sales.xlsx", sheet_name=None
```

במקרים שבהם יש צורך לעבד את הנתונים ולשלוח חזרה דו"ח מעודכן
ניתן לשמור חזרה לקובץ Excel:

```
df.to_excel("data/clean_sales.xlsx", index=False)
```

חשוב להבין ש-Excel אינו פורמט יעיל לעיבוד כמו Parquet,
אך הוא שימושי לשכבת אינטגרציה עם משתמשים לא-טכניים.

במערכת AI, תראה אותו לרוב בשלב הייבוא הראשוני של נתונים גולמיים לפני ניקוי והמרה לפורמט יעיל יותר.

**Parquet**

פורמט עמודות (Columnar) מודרני שמיועד לנפחי נתונים גדולים.
בשונה מ-CSV, הוא שומר את **סוגי הנתונים (dtypes)** ואת מבנה הטבלה, דוחס כל עמודה בנפרד, ומאפשר טעינה סלקטיבית של עמודות בלבד.

```
import pandas as pd

df.to_parquet("data/users.parquet")
df = pd.read_parquet("data/users.parquet")
```

היתרון המרכזי, מהירות ויעילות.

טעינה מקובץ Parquet גדולה פי כמה מטעינה מקובץ CSV,

בזכות דחיסה חכמה (Snappy או ZSTD) וגישה ישירה לבלוקים.

במערכות AI, זהו הפורמט המועדף לאחסון datasets לאחר ניקוי:

- מאפשר טעינה ישירה למודלי למידה או ל-Data Pipeline של ה-Data Lake.
- משתלב טבעית עם Spark, Polars, DuckDB ו-BigQuery.
- שומר עקביות בטיפוסים בין שלבי עיבוד שונים.

במילים פשוטות:
CSV מתאים לשלבים הראשונים של איסוף נתונים.
Parquet מתאים לכל שלב אחרי הניקוי. לפני אימון, ניתוח או הפצה.

## שכבת IO אחידה

במערכת מבוססת AI, מומלץ לרכז את כל פעולות הקריאה והכתיבה בקובץ ייעודי, לדוגמה data_io.py.
כך ניתן להחליף פורמט או מקור נתונים מבלי לשנות את שאר הקוד.

## בחירה וסינון: loc, iloc, Boolean Indexing

לאחר טעינת הנתונים, מגיע השלב שבו צריך **לשלוף בדיוק את מה שרלוונטי**.

ב-Pandas קיימות שלוש דרכים עיקריות לגשת לנתונים:

iloc, loc, ו-Boolean Indexing.

כל אחת מהן פועלת באופן שונה, אך כולן בנויות סביב אותו רעיון.

**גישה וקטורית מהירה**.

### loc – לפי שם

loc עובדת לפי שמות האינדקס והעמודות.

זהו הממשק הברור ביותר כשיש עמודות בעלות משמעות.

```
import pandas as pd

df = pd.DataFrame({
  "name": ["דנה", "רועי", "הילה"],
  "age": [29, 34, 41],
  "city": ["תל אביב", "חיפה", "ירושלים"]
}).set_index("name")

df.loc["דנה", "city"]
```

ניתן גם לבחור תת-טבלה:

```
df.loc[["דנה", "הילה"], ["age", "city"]]
```

## iloc – לפי מיקום

iloc דומה אך מתבססת על מיקום מספרי (אינדקסים).
שימושית בעיקר כשאין אינדקס סמנטי.

```
df.iloc[0]      # השורה הראשונה
df.iloc[:, 1]   # העמודה השנייה
```

התחביר דומה ל-NumPy, אך מחזיר תמיד אובייקטים של Pandas (לא רשימות או מערכים).

## Boolean Indexing – לפי תנאי

זוהי השיטה הגמישה ביותר:
מסנן שמבוסס על ביטוי לוגי.

```
df[df["age"] > 30]
```

אפשר לשלב כמה תנאים:

```
df[(df["age"] > 30) & (df["city"] == "חיפה")]
```

שיטה זו היא הבסיס לכל סינון דינמי. החל ממיון משתמשים פעילים ועד חיתוך dataset לפני אימון.

## בחירה מתקדמת

כל השיטות ניתנות לשילוב.
לדוגמה, שליפה לפי תנאי ולאחר מכן בחירה בעמודות מסוימות בלבד:

```
df.loc[df["age"] > 30, ["city"]]
```

גישה כזו חוסכת לולאות, מונעת שגיאות, ונשארת קריאה גם כשעובדים על מיליוני שורות.

## טרנספורמציות: apply, map, groupby

לאחר שלב הקריאה והסינון, מגיע שלב העיבוד.

כאן מתבצעות כל ההמרות, החישובים והאגרגציות שמכינים את הנתונים לשלב הבא ב-Data Pipeline.

### map – טרנספורמציה לעמודה בודדת

Map מאפשרת לבצע שינוי ישיר על עמודה אחת.

ניתן להשתמש בפונקציה, ב-lambda, או במילון של החלפות.

```
import pandas as pd

df = pd.DataFrame({
  "name": ["דנה", "רועי", "הילה"],
  "city": ["tel aviv", "haifa", "jerusalem"]
})

df["city"] = df["city"].map(str.title)
```

פלט:

```
  name      city
0     דנה          Tel Aviv
1     רועי     Haifa
2     הילה          Jerusalem
```

שיטה זו יעילה כשנדרש שינוי פשוט בעמודה יחידה, כמו נירמול טקסטים או החלפת ערכים.

## apply – פונקציה על שורה או עמודה

Apply מאפשרת הפעלת פונקציה על כל שורה או עמודה.

זו הדרך הנוחה ביותר לבצע חישובים מותאמים אישית.

```
df["name_length"] = df["name"].apply(len)
```

ניתן גם להפעיל פונקציה על כל שורה (axis=1):

```
df["desc"] = df.apply(lambda r: f"{r['name']} - {r['city']}", axis=1)
```

היתרון, גמישות.

החיסרון, איטי יחסית לפעולות וקטוריות.

לכן ב-datasets גדולים עדיף להשתמש ב-NumPy או ב-transform מובנות של Pandas.

```
sales = pd.DataFrame({
   "region": ["צפון", "דרום", "צפון", "מרכז"],
   "amount": [120, 80, 150, 200]
})

sales.groupby("region")["amount"].mean()
```

פלט:

```
region
 דרום  80.0
 צפון    135.0
 מרכז    200.0
Name: amount, dtype: float64
```

כך ניתן לחשב ממוצעים, סכומים, או סטטיסטיקות אחרות לכל קבוצה
למשל, ממוצע דירוגים לפי משתמש או קטגוריה.

## לסיכום

- Map – שינוי עמודה בודדת.
- Apply – טרנספורמציה מורכבת לפי פונקציה.
- Groupby – אגרגציה לפי מאפיין.

שלושת הכלים האלו מרכיבים את ליבת העיבוד של Pandas.
במערכות AI, הם משמשים בכל שלב של עיבוד features: ניקוי, העשרה, ויצירת משתנים חדשים לפני האימון.

## טיפול בנתונים חסרים: NaN, fillna, dropna

ב-datasets אמיתיים תמיד יהיו ערכים חסרים.

הם עשויים לנבוע משדות שלא נמדדו, טעויות הזנה, או מבנה נתונים חלקי.

ב-Pandas ערכים חסרים מיוצגים על-ידי **NaN** (Not a Number), והטיפול בהם הוא שלב חיוני לפני כל ניתוח או אימון מודל.

### זיהוי ערכים חסרים

השיטה הראשונה היא זיהוי:

```
import pandas as pd
import numpy as np

df = pd.DataFrame({
    "name": ["דנה", "רועי", "הילה"],
    "age": [29, np.nan, 41],
    "city": ["תל אביב", None, "ירושלים"]
})

df.isna()
```

isna() מחזירה טבלת True/False לפי מיקום הערכים החסרים.

כדי לבדוק כמה חסרים קיימים בכל עמודה:

```
df.isna().sum()
```

## הסרת ערכים חסרים

אם הנתונים החסרים מועטים, אפשר פשוט להסיר את הרשומות:

```
clean_df = df.dropna()
```

ברירת המחדל מסירה כל שורה שבה יש לפחות NaN אחד.
אם נרצה להסיר רק שורות שבהן כל הערכים חסרים:

```
df.dropna(how="all")
```

## מילוי ערכים חסרים

כאשר הנתונים חשובים מדי להסרה, ניתן למלא אותם בערך ברירת מחדל:

```
df["age"] = df["age"].fillna(df["age"].mean())
df["city"] = df["city"].fillna("לא ידוע")
```

fillna מאפשרת גם **שחזור ערכים סמוכים** ב-datasets סדרתיים (כמו סדרות זמן):

```
df["age"].fillna(method="ffill", inplace=True)  # העתקת הערך הקודם
```

## גישה הנדסית

במערכות AI, הדרך הנכונה לטפל ב-NaN תלויה בהקשר:

- **ב-features כמותיים**: החלפה בממוצע, חציון או ערך נורמלי אחר.
- **ב-features קטגוריים**: מילוי בערך ייחודי (למשל "missing").
- **ב-features חשובים לאימון**: שימוש במודל משני (imputer) לחיזוי ערכים חסרים.

המטרה, לשמור על עקביות הנתונים מבלי להחדיר הטיה.

## תובנה מעשית

טיפול בערכים חסרים הוא לא רק ניקוי טכני
זו החלטה סטטיסטית שמשפיעה על איכות המודל.
הדרך שבה אתה ממלא או מסיר NaN היא חלק מהאחריות ההנדסית שלך.

## מיזוג datasets: merge, concat, join

בפרויקטים אמיתיים המידע לעולם לא מגיע ממקור אחד.
יש טבלה עם משתמשים, טבלה עם רכישות, אולי גם לוגים או טקסטים.
ב-Pandas שלושת הכלים המרכזיים למיזוג נתונים הם merge, concat ו-join.

### merge – איחוד לפי מפתח משותף

merge היא המקבילה של פעולת JOIN ב-SQL.
היא מאפשרת לחבר שתי טבלאות לפי עמודה משותפת, למשל user_id.

```
import pandas as pd

users = pd.DataFrame({
    "user_id": [1, 2, 3],
    "name": ["דנה", "רועי", "הילה"]
})

orders = pd.DataFrame({
    "user_id": [1, 1, 2],
    "order_amount": [120, 80, 200]
})

merged = pd.merge(users, orders, on="user_id", how="left")
```

פלט:

```
  user_id  name  order_amount
0     1       דנה    120
1     1       דנה    80
2     2        רועי    200
3     3        הילה    NaN
```

הפרמטר how מגדיר את סוג המיזוג – "inner", "left", "right", או "outer".

השימוש הנפוץ ביותר הוא "left" כדי לשמור את הנתונים מטבלת הבסיס גם כשאין התאמה מלאה.

### concat – איחוד אנכי או אופקי

concat משמשת להדבקה של DataFrames זה מעל זה (או זה לצד זה).

מושלם כשמקבלים קבצים מאותו מבנה מכמה מקורות.

```
q1 = pd.DataFrame({"month": ["Jan", "Feb"], "sales": [100, 120]})
q2 = pd.DataFrame({"month": ["Mar", "Apr"], "sales": [130, 140]})

df = pd.concat([q1, q2], ignore_index=True)
```

פלט:

```
  month  sales
0   Jan    100
1   Feb    120
2   Mar    130
3   Apr    140
```

אם מעבירים axis=1, ההדבקה מתבצעת אופקית, עמודות לצד עמודות.

## join – קיצור נוח לאיחוד לפי אינדקס

join מאפשרת למזג DataFrames לפי אינדקס, שימושית במיוחד לאחר שהוגדר set_index().

```
users = users.set_index("user_id")
orders = orders.set_index("user_id")
users.join(orders, how="left")
```

אותו רעיון כמו merge, אבל תחביר נקי יותר כשעובדים עם אינדקסים.

## שימוש בעולם ה-AI

שלב המיזוג הוא קריטי בהכנת datasets לאימון:

- איחוד טבלאות features ממקורות שונים (מידע דמוגרפי, שימושי, התנהגותי).
- שילוב נתוני טקסטים עם תוויות (labels) ממערכות נפרדות.

- שמירה על עקביות מזהים בין שלבים שונים ב-Data Pipeline.

בחירה נכונה בין concat, merge ו-join קובעת אם תקבל dataset עקבי או בלגן שקשה לאתר בו שגיאות.

## דוגמה מרכזית: עיבוד dataset של טקסטים בעברית

עד עכשיו ראינו את כל הכלים הבסיסיים של Pandas.
בשלב הזה נחבר אותם יחד לתהליך שלם
מהקריאה של הנתונים ועד להכנה שלהם לשימוש במודל שפה.

### הנתונים

נניח שקיבלנו dataset של ביקורות משתמשים על מוצרים, בקובץ CSV:

```
id,text,rating
1,מוצר מצוין,5
2,משלוח איטי,2
3,שרות מעולה!,4
4,לא מרוצה,1
5,איכות טובה מאוד,5
```

טעינה וניקוי בסיסי

```
import pandas as pd

df = pd.read_csv("data/reviews.csv", encoding="utf-8")
df.dropna(subset=["text"], inplace=True)
```

נפטרנו מרשומות חסרות ונשארנו רק עם שורות שבהן יש טקסט.

## טרנספורמציה ונירמול טקסטים

השלב הבא הוא ניקוי הערות המשתמשים לקראת עיבוד שפה טבעית (NLP). נשתמש בפונקציה פשוטה לנירמול:

```
def normalize(text: str) -> str:
    text = text.strip().lower()
    return text.replace("!", "").replace(".", "")

df["clean_text"] = df["text"].apply(normalize)
```

פלט:

```
     text        clean_text
0   מוצר מצוין            מוצר מצוין
1   משלוח איטי            משלוח איטי
2   שרות מעולה!     שרות מעולה
3    לא מרוצה            לא מרוצה
4   איכות טובה מאוד       איכות טובה מאוד
```

## הוספת Feature חדש

נחשב את אורך כל ביקורת כמספר מילים

Feature בסיסי אך שימושי למודלים של סנטימנט:

```
df["word_count"] = df["clean_text"].apply(lambda t:
len(t.split()))
```

קיבוץ לפי דירוג

```
avg_len = df.groupby("rating")["word_count"].mean()
```

```
print(avg_len)
```

כך נוכל לגלות תובנות ראשוניות. למשל, האם ביקורות שליליות קצרות יותר מביקורות חיוביות.

## שמירה לפורמט יעיל

```
df.to_parquet("data/clean_reviews.parquet", index=False)
```

ה-DataFrame הנקי מוכן לשימוש במודל שפה, לאימון או לאחזור ב-RAG Pipeline.

Parquet מבטיח טעינה מהירה ויעילה לכלי AI מתקדמים.

## מבט מערכתי

התהליך משקף דפוס שחוזר כמעט בכל פרויקט AI:

1. קריאה ממקור נתונים (CSV, JSON).
2. ניקוי ונירמול.
3. יצירת מאפיינים חדשים.
4. סינון או קיבוץ לפי הקשר.
5. שמירה לפורמט יעיל.

זו השלד של כל תהליך עיבוד נתונים. פשוט, קריא, ומדויק.

## ביצועים וסקיילינג: מתי Pandas לא מספיקה

Pandas היא ספרייה מצוינת לעיבוד נתונים.

עד גבול מסוים.
כאשר נפח הנתונים עובר כמה ג׳יגה-בייטים, או כשהפעולות נעשות כבדות מדי לזיכרון, מתחילים לראות האטות ואף קריסות.

בנקודה הזו חשוב להבין את המגבלות של Pandas ואת הכלים החלופיים שפותרים אותן.

## המגבלה המרכזית: עבודה בזיכרון (in-memory)

Pandas טוענת את כל הנתונים לזיכרון הראשי (RAM).
אם יש לך קובץ של GB5, תצטרך פי שניים או שלושה מזה בזיכרון כדי לעבוד עליו.
לכן במערכות AI שמתמודדות עם datasets עצומים, נדרשת גישה אחרת: עיבוד מבוזר או עיבוד בעמודות בלבד.

## Polars – הגרסה המהירה של Pandas

**Polars** היא ספרייה מודרנית שנכתבה ב-Rust, ומבוססת על **עיבוד עמודות**.
היא מהירה משמעותית מ-Pandas, תומכת בעיבוד מקבילי (multithreading), ומציגה ממשק API דומה מאוד, כך שקל לעבור אליה.

```
import polars as pl

df = pl.read_csv("data/large_dataset.csv")
summary = df.groupby("category").agg(pl.col("value").mean())
```

היתרונות:

- טעינה מהירה במיוחד גם על קבצי ענק.
- שימוש בזיכרון נמוך יותר.
- תחביר דומה ל-Pandas, אך פונקציונלי יותר (דמוי SQL).

במבחנים מעשיים, Polars מהירה פי 5–10 מ-Pandas על פעולות groupby או join גדולות.

### Pandas – Dask על פני כמה ליבות או מכונות

**Dask** מרחיבה את Pandas לעיבוד מקבילי מבוזר.
במקום לטעון את כל הנתונים בבת אחת, היא מחלקת את העבודה למקטעים קטנים (Chunks)
ומבצעת אותם במקביל בזיכרון או באשכול (Cluster).

```
import dask.dataframe as dd

df = dd.read_csv("data/large_dataset.csv")
df["value"].mean().compute()
```

שיטת העבודה כמעט זהה ל-Pandas
אך מאחורי הקלעים Dask מפעילה מערכת תורים שמבצעת את החישובים בחלקים.

שימושית במיוחד כשצריך לעבד **מאות ג׳יגה-בייטים** של נתונים על מחשב רגיל, או להריץ **Preprocessing** למודלי למידה בענן.

## מתי לעבור מ-Pandas?

| מצב | כלי מתאים |
|---|---|
| **עד 2 GB נתונים בזיכרון** | Pandas |
| **עיבוד על ליבות מרובות במכונה אחת** | Polars |
| **עיבוד מבוזר על כמה מכונות / ענן** | Dask |

המעבר אינו מחייב שינוי דרמטי, ברוב המקרים הקוד כמעט זהה, אבל התשתית שמאחוריו יעילה בהרבה.

## תובנה מעשית

Pandas נועדה לשלב ה-Exploration, כשבודקים, מנקים ומבינים את הנתונים.
כאשר הפרויקט עובר לשלב **אוטומציה** או **סקייל**,
יש לעבור לכלי שמבצע עיבוד במקביל או על אחסון מבוזר.

המפתח הטוב יודע לזהות את הנקודה הזו בזמן
לפני שהקוד נתקע או מאבד ביצועים.

## סיכום: Pandas כשכבת הנתונים במערכת AI

Pandas אינה רק ספרייה לעיבוד טבלאות.
היא שכבת הנתונים של מהנדס ה-AI, המקום שבו הנתון עובר את המעבר הקריטי ממידע גולמי ל-**ידע מוכן ללמידה**.

### התפקיד של Pandas ב-Data Pipeline

בכל מערכת AI קיימים שלושה שלבים מרכזיים:

1. **איסוף וטעינה (Ingestion)**
קריאת נתונים ממקורות שונים: CSV, APIs, קבצי לוג, JSON.
כאן Pandas היא תחנת הכניסה, מאפשרת טעינה מהירה ובקרה על סוגי הנתונים.

2. **עיבוד והעשרה (Processing & Enrichment)**
ניקוי, המרה, נירמול, יצירת features חדשים.
זהו הלב של העבודה ב-Pandas, באמצעות כלים כמו apply, map, ו-groupby.

3. **הכנה לשלב הבא (Output)**
שמירה לפורמט יעיל כמו Parquet, או העברה ישירה לשכבת למידה, אחסון, או שירות אחזור (RAG, Vector DB).

במילים פשוטות: Pandas סוגרת את הפער שבין הנתונים כפי שהם נאספים לבין הנתונים כפי שמודל ה-AI צריך אותם.

## שילוב עם ספריות מתקדמות

בפרויקטים מודרניים Pandas, היא רק תחילת השרשרת:

- **Pandas** – ניקוי והכנת הנתונים.
- **NumPy / Scikit-learn** – עיבוד מתמטי וטרנספורמציות מתקדמות.
- **TensorFlow / PyTorch** – אימון מודלים.
- **Polars / Dask** – סקיילינג ועיבוד מבוזר.

המעבר בין הכלים האלה חלק, משום שכולם דוברים את אותה "שפת נתונים" – DataFrame.

## Best Practices למהנדסי AI

- עבד תמיד עם `UTF-8` – כל dataset עתידי צריך לתמוך בעברית ובשפות נוספות.
- הגדר `dtype` כבר בקריאה כדי לחסוך זיכרון.
- אל תשתמש בלולאות – העדף פעולות וקטוריות (`apply`, `map`).
- אחסן תוצאות ב-Parquet, לא ב-CSV.
- אל תשאיר ערכי NaN לפני אימון – טפל בהם באופן עקבי.
- שמור שכבת IO אחת אחידה בקוד (`data_io.py`).

- כשתראה שהקובץ נהיה כבד מדי – עבור ל-Polars או Dask מוקדם, לפני שהמערכת תקרוס.

## המסקנה

Pandas היא הכלי שבו המהנדס שולט על ה-Data Pipeline שלו.
היא הופכת בלגן של נתונים לאובייקט מובנה, ניתן לבדיקה ולמדידה.
כל תהליך AI מצליח. ממודל פשוט ועד מערכת חכמה בקנה מידה ארגוני מתחיל ב-DataFrame נקי, עקבי, ומוכן ללמידה.

# פרק 15 – אסינכרוניות בסיסית וממשקי רשת

## למה async חשוב בפרויקטי AI

עולם ה-AI בנוי על **תקשורת רשת**.
כל קריאה למודל.
בין אם זה OpenAI, Hugging Face או שירות פנימי
היא קריאה **חיצונית**, ולכן איטית יחסית לפעולות CPU.

כאשר אתה שולח עשרות או מאות בקשות במקביל (למודלי שפה, שירותי Embedding, APIs חיצוניים),
הגישה הסינכרונית הקלאסית פשוט לא מספיקה.
הלולאה הראשית נתקעת, וכל משימה מחכה לסיום הקודמת.

כאן נכנסת **אסינכרוניות** – **(async/await)**
מנגנון שמאפשר לפייתון להריץ משימות במקביל לוגית (לא פיזית), כלומר להמשיך לעבד משימה אחת בזמן שאחרת ממתינה לתגובה מהשרת.

התוצאה: שיפור מהירויות פי עשרות, בלי צורך בתהליכים נפרדים או תורים חיצוניים.

במערכות AI אמיתיות, Async הוא כבר לא "אופטימיזציה", אלא **סטנדרט תשתיתי**.

## הבסיס :async/await

התחביר של אסינכרוניות בפייתון פשוט:
מגדירים פונקציה אסינכרונית בעזרת `async def`,
וממתינים לתוצאה של פעולה אסינכרונית בעזרת `await`.

```
import asyncio

async def fetch_data():
    print("🔄 מתחיל בקשת נתונים...")
    await asyncio.sleep(2)  # מדמה עיכוב רשת
    print("✅ נתונים התקבלו")
    return {"status": "ok"}

async def main():
    result = await fetch_data()
    print(result)

asyncio.run(main())
```

פלט:

```
🔄 מתחיל בקשת נתונים...
✅ נתונים התקבלו
{'status': 'ok'}
```

`await` משחרר את השליטה ללולאת האירועים בזמן שהפונקציה "מחכה" כך תהליכים אחרים יכולים לרוץ במקביל.

זהו ההבדל המהותי בין *ריבוי תהליכים (Threads)* ל-*אסינכרוניות*:
Async הוא **לא ריבוי מעבדים**, אלא ניהול חכם של זמני המתנה.

## לולאת האירועים: asyncio.run ו-gather

בכל תוכנית אסינכרונית קיימת **לולאת אירועים** (Event Loop) שאחראית על תזמון והרצת המשימות.
פייתון מספקת ממשק פשוט לניהול הלולאה הזו.

```
import asyncio

async def task(n):
    await asyncio.sleep(1)
    return f"Task {n} done"

async def main():
    results = await asyncio.gather(
        task(1),
        task(2),
        task(3)
    )
    print(results)

asyncio.run(main())
```

asyncio.gather() מריץ כמה פונקציות אסינכרוניות במקביל ומחזיר את התוצאות ברגע שכולן הסתיימו.

בדוגמה זו שלוש המשימות רצות יחד, לא זו אחר זו.

למה זה חשוב?
ב-AI pipelines אתה שולח עשרות בקשות ל-API של מודל.
בלי gather – כל קריאה תחכה לסיום הקודמת.
עם gather – כולן נשלחות ונאספות במקביל.

## הגבלת מקביליות בעזרת asyncio.Semaphore

כאשר משגרים עשרות בקשות במקביל יש צורך להגביל את מספר המשימות הפעילות כדי לא להציף שרתים או לעבור מגבלות קצב . asyncio.Semaphore קובע תקרה של משימות פעילות בו זמנית.

אם התקרה הושגה משימות נוספות ממתינות עד שאחת מסתיימת. כך שומרים על יציבות ובקרה תוך שמירה על רמת מקביליות גבוהה.

דוגמה מינימלית:

```
import asyncio

sem = asyncio.Semaphore(10)

async def guarded(coro):
    async with sem:
        return await coro
```

בדוגמה זו רק עשר משימות ירוצו במקביל. כאשר אחת מסתיימת משימה ממתינה נכנסת תחתיה.

### aiohttp: קריאות רשת אסינכרוניות

הספרייה **aiohttp** מספקת ממשק אסינכרוני לביצוע בקשות HTTP.
היא מחליפה את requests בסביבות שבהן נדרשת ריבוי קריאות במקביל.

```
import aiohttp
import asyncio

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        urls = ["https://example.com"] * 5
        tasks = [fetch(session, u) for u in urls]
        results = await asyncio.gather(*tasks)
        print(f"{len(results)} תגובות התקבלו")

asyncio.run(main())
```

כך ניתן לבצע **מאות קריאות רשת בו-זמנית**
תכונה קריטית בעבודה מול APIs של AI שבהם זמן תגובה ממוצע הוא שניות, לא מילישניות.

## Timeouts ו-Retry: הגנה על קריאות API

בעבודה עם APIs של AI, במיוחד כאלה שמבוזרים או חיצוניים
אין דבר בטוח יותר מהבלתי צפוי.
קריאה אחת עלולה לקחת שנייה, ואחרת, חמש.
שרת עלול להחזיר שגיאת 429 (Rate Limit) או פשוט להפסיק
להגיב.

אם לא תנהל את זה נכון, תוכנית אסינכרונית יכולה **להיתקע לנצח**.

לכן חובה להגדיר **Timeouts** ו-**Retry** חכמים לכל קריאה רשתית.

### Timeout – הגבלת זמן לכל בקשה

ב-aiohttp ניתן להגדיר Timeout ישירות על ה-session או על כל
בקשה בנפרד:

```
import aiohttp
import asyncio

async def fetch(url):
   timeout = aiohttp.ClientTimeout(total=3)
   async with aiohttp.ClientSession(timeout=timeout) as
session:
      async with session.get(url) as resp:
         return await resp.text()
```

```
async def main():
    try:
        result = await fetch("https://httpbin.org/delay/5")
        print(result)
    except asyncio.TimeoutError:
        print("⏱ Timeout – הבקשה נקטעה לאחר 3 שניות")

asyncio.run(main())
```

פייתון תזרוק TimeoutError אם השרת לא הגיב בזמן,
וכך הלולאה ממשיכה לרוץ במקום להיתקע.

## Retry – ניסיון חוזר אוטומטי

Timeout פותר תקיעות, אבל לא שגיאות חולפות.
לכן מוסיפים שכבת Retry, ניסיון נוסף לאחר כשל זמני.

```
import aiohttp
import asyncio


async def fetch_with_retry(url, retries=3):
    for attempt in range(1, retries + 1):
        try:
            async with aiohttp.ClientSession() as session:
                async with session.get(url) as resp:
                    return await resp.text()
        except Exception as e:
            print(f" ניסיון{attempt} נכשל: {e}")
```

```
        await asyncio.sleep(1)
    raise RuntimeError("הבקשה נכשלה לאחר כל הניסיונות")

asyncio.run(fetch_with_retry("https://example.com"))
```

גישה זו מאפשרת יציבות: גם אם קריאה אחת נכשלה,
המערכת לא קורסת אלא מנסה שוב – בדיוק כפי שמצופה
ממערכת AI ב-production.

### שילוב שני המנגנונים

בפרויקט אמיתי, מגדירים Timeout ו-Retry יחד כחלק משכבת
תקשורת אחידה.
לדוגמה, מודול בשם `api_client.py` שדרכו כל השירותים מבצעים
קריאות חיצוניות.
כך כל קריאה לרשת נהנית מהגנה אוטומטית.

### Cancellation: ביטול משימות אסינכרוניות

אחד היתרונות המשמעותיים של אסינכרוניות הוא היכולת לשלוט
במשימות בזמן אמת.
בעולם ה-AI, זה שימושי במיוחד: ייתכן שמודל מאט, שהמשתמש
לחץ "ביטול", או שהגיע מידע חדש שמייתר את הבקשה הקודמת.

כדי למנוע בזבוז משאבים.

צריך לדעת **איך לעצור משימה שרצה בלולאת האירועים**.

## משימה בודדת

ב-asyncio ניתן לבטל כל משימה (Task) באמצעות ()cancel.

```
import asyncio

async def slow_task():
  print("🚀 התחלת משימה...")
  try:
    await asyncio.sleep(5)
    print("✅ הסתיימה בהצלחה")
  except asyncio.CancelledError:
    print("🛑 המשימה בוטלה")

async def main():
  task = asyncio.create_task(slow_task())
  await asyncio.sleep(2)
  task.cancel()
  await task

asyncio.run(main())
```

פלט:

```
🚀 התחלת משימה...
🛑 המשימה בוטלה
```

הביטול לא "הורג" את המשימה מיידית, אלא מעלה חריגת CancelledError בתוך הפונקציה, מה שמאפשר **ניקוי מסודר (cleanup)** לפני סיום.

### ביטול קבוצת משימות

כאשר מריצים כמה משימות במקביל (למשל בקשות ל-API), ניתן לבטל את כולן בצורה מרוכזת.

```
import asyncio

async def fetch(n):
  try:
    print(f"🔄 התחלה{n}")
    await asyncio.sleep(3)
    print(f"✅ סיום{n}")
  except asyncio.CancelledError:
    print(f"🛑 {n} בוטלה")

async def main():
  tasks = [asyncio.create_task(fetch(i)) for i in range(3)]
  await asyncio.sleep(1)
  for t in tasks:
    t.cancel()
  await asyncio.gather(*tasks, return_exceptions=True)

asyncio.run(main())
```

כך ניתן לעצור את כל המשימות כשהמערכת מגלה מצב חריג. למשל כשנפלה תקשורת עם שרת חיצוני או התקבל

stop signal ממנוע התזמון.

## שימוש בעולם ה-AI

Cancellation חשוב במיוחד כשעובדים עם APIs יקרים או איטיים:

- המשתמש ביטל שאלה בממשק שיחה (Chat UI).
- אחת מבקשות ה-Embedding כבר לא נחוצה.
- קריאת RAG ארוכה מדי ומחליטים להחזיר תשובה חלקית.

במקום לחכות לסיום הקריאה, ניתן לעצור אותה מיידית ולשחרר משאבים.

## דוגמה מרכזית: שליחת בקשות רבות ל API במקביל

לאחר שהבנו כיצד להריץ משימות אסינכרוניות במקביל באמצעות asyncio.gather וכיצד להגביל את מספרן בעזרת Semaphore, נוכל לבנות דוגמה שלמה שמדגימה את כל עקרונות העבודה עם APIs בעולם ה-AI.

בפרויקטים אמיתיים לעיבוד שפה טבעית או לאימון מודלים, נדרש לעיתים לשלוח עשרות ואף מאות בקשות למודל חיצוני

למשל, ליצירת Embeddings, לסיווג טקסטים או לשאילתות RAG.
כדי לעשות זאת בצורה יציבה נשתמש בארבעה עקרונות בסיסיים:

1. **מקביליות מבוקרת** – בעזרת Semaphore כדי למנוע הצפה של השרת.
2. **Timeout** – הגבלת זמן תגובה לכל בקשה.
3. **Retry** – ניסיון חוזר במקרים של כשל זמני.
4. **ביטול משימות** – עצירה נקייה של כל הקריאות במקרה של ביטול כולל.

בדוגמה הבאה נבנה מימוש קצר המדגים את כל אלה יחד

כך נראה שלד טיפוסי שבו משתמשים בפרויקטי AI המתבססים על מודלים חיצוניים כמו OpenAI או Gemini.

```
import asyncio
import aiohttp

API_URL = "https://httpbin.org/delay/1"  # מדמה השהיה של שרת

class ApiError(Exception):
    pass

async def fetch_one(session: aiohttp.ClientSession, url: str, *,
timeout_s: float) -> str:
    """קריאה אחת עם Timeout וטיפול בשגיאה."""
```

```
    try:
        async with session.get(url, timeout=timeout_s) as resp:
            if resp.status >= 400:
                raise ApiError(f"סטטוס שגוי {resp.status}")
            return await resp.text()
    except asyncio.TimeoutError:
        raise ApiError("Timeout")
    except aiohttp.ClientError as e:
        raise ApiError(f"שגיאת רשת: {e}")

async def fetch_with_retry(session, url, *, timeout_s: float,
retries: int = 3, backoff_s: float = 0.5) -> str:
    """ניסיון חוזר במקרים של כשל זמני."""
    for attempt in range(1, retries + 1):
        try:
            return await fetch_one(session, url,
timeout_s=timeout_s)
        except ApiError as e:
            if attempt == retries:
                raise
            await asyncio.sleep(backoff_s * attempt)

async def fetch_many(urls: list[str], *, concurrency: int = 10,
timeout_s: float = 3.0) -> list[str]:
    """שליחה במקביל עם הגבלת קצב (Semaphore) ויכולת ביטול
חכמה."""
    sem = asyncio.Semaphore(concurrency)
```

```
  async with aiohttp.ClientSession() as session:
    async def guarded(url: str) -> str:
      async with sem:  # רק מספר מוגבל של משימות רצות בו-זמנית
        return await fetch_with_retry(session, url,
timeout_s=timeout_s)

    tasks = [asyncio.create_task(guarded(u)) for u in urls]

    try:
      results = await asyncio.gather(*tasks,
return_exceptions=True)
    except asyncio.CancelledError:
      for t in tasks:
        t.cancel()
      raise

  errors = [r for r in results if isinstance(r, Exception)]
  if errors:
    raise ApiError(f"כשל בחלק מהבקשות: {len(errors)} מתוך
{len(results)}")

  return results

async def main():
  urls = [API_URL for _ in range(50)]
  try:
```

```
        texts = await fetch_many(urls, concurrency=8, 
timeout_s=2.5)
        print(f" התקבלו{len(texts)} תגובות תקינות")
    except ApiError as e:
        print(f"שגיאה כוללת: {e}")

if __name__ == "__main__":
    asyncio.run(main())
```

כך נראית מערכת אסינכרונית מלאה ל-AI API:

הקריאות נשלחות במקביל אך בעומס מבוקר, כל אחת מוגנת ב-Timeout, וכשמתרחשת תקלה זמנית, מתבצע ניסיון חוזר אוטומטי.

במקרה של ביטול כולל (למשל, המשתמש עצר את הבקשה בממשק), כל המשימות נעצרות באופן נקי, מבלי להשאיר קריאות "יתומות" פתוחות.

הגישה הזו מבטיחה מערכת מהירה, יציבה וניתנת לניטור.

בדיוק מה שנדרש בעבודה עם שירותי AI חיצוניים בקנה מידה אמיתי.

## Best Practices: אסינכרוניות ל-I/O בלבד וטיפול חכם בשגיאות

אסינכרוניות נועדה לפעולות של **קלט-פלט**
רשת, קבצים, בסיסי נתונים.
היא לא מיועדת לחישוב כבד. אם אתה צריך לעבד נתונים או להריץ אלגוריתם ארוך,
הרץ אותו בתהליך נפרד בעזרת `ProcessPoolExecutor` או ספרייה ייעודית.
כך לולאת האירועים תישאר פנויה לניהול רשת ולא תיתקע על חישוב.

כשעובדים מול שירותי AI, כדאי לרכז את כל הקריאות החיצוניות בקובץ אחד, לדוגמה `api_client.py`.
זו "שכבת תקשורת" אחת שמנהלת Timeout, Retry ולוגים אחידים.
ברגע שכל הקריאות עוברות דרך אותו שער. הרבה יותר קל לנטר, לבדוק ולשפר.

עוד כלל חשוב: אל תפתח `ClientSession` חדש בכל בקשה.
במקום זה, פתח Session אחד בתחילת העבודה ומחזר אותו לכל הבקשות.
פתיחה וסגירה חוזרת מייצרת overhead מיותר ולעיתים גם דליפות משאבים.

```
import aiohttp

timeout = aiohttp.ClientTimeout(total=5)
connector = aiohttp.TCPConnector(limit=20)  # הגבלת מספר
חיבורים פתוחים
session = aiohttp.ClientSession(timeout=timeout,
connector=connector)
# ... עבודה עם session ...
# await session.close()
```

כאשר מריצים משימות רבות במקביל,
מומלץ להשתמש ב- asyncio.gather(..., return_exceptions=True).
כך גם אם חלק מהמשימות נכשלות, תקבל את כל התוצאות
התקינות
ותוכל להחליט מה לעשות הלאה.

**לדוגמה**, לנסות שוב רק את אלו שנכשלו.

גם ביטול משימות צריך להיות נקי.
ביטול מעלה את החריגה CancelledError, ולכן כדאי לעטוף את
הקוד הקריטי ב-try ולוודא שסוגרים חיבורים או קבצים לפני
שהפונקציה האסינכרונית מסתיימת.

מדיניות Timeout ו-Retry צריכה להיות **שמרנית**.
קבע זמן תגובה קצר יחסית, מספר ניסיונות מוגבל, והמתנה מעט

ארוכה יותר בכל ניסיון.
כבד את כותרות ה-Rate Limit של ה-API והשהה בהתאם,
כדי למנוע חסימה או ענישה מהשרת.

בנוסף, חשוב לתעד כל בקשה בלוגים: מזהה, זמן תגובה, סטטוס, סיבת כשל.
במערכות AI מרובות קריאות, הלוגים הם כלי אבחון קריטי למציאת צווארי בקבוק.

לבסוף, הקפד לסגור הכול.
סגור את ה-Session, בטל משימות תלויות, נקה חיבורים.
סגירה מסודרת היא לא המלצה.

זו הדרך היחידה לשמור על יציבות לאורך זמן.

## סיכום: למה async ו-aiohttp הם חובה בפרויקטי AI

בעולם של מערכות AI, כמעט כל שלב כולל **תקשורת רשת**
בקשות למודל שפה, שאילתות למנוע Embeddings, גישה ל-API של חיפוש או שירות אחזור.
כל בקשה כזו אורכת שניות, לא מילישניות,
וכשיש עשרות מהן בכל שלב, ביצוע סינכרוני פשוט לא עומד בקצב.

כאן נכנס async.
במקום לחכות לכל בקשה שתסתיים,

המערכת שולחת את כולן במקביל וממשיכה לעבוד בזמן שהתגובות חוזרות.
זהו ההבדל בין קוד איטי שחוסם את עצמו לבין מערכת יעילה שמנצלת כל רגע המתנה.

הספרייה aiohttp הופכת את העיקרון הזה לפרקטיקה:
היא מאפשרת לנהל אלפי חיבורים פתוחים בצורה קלה,
לטפל ב-Timeout, לנסות שוב בקריסה זמנית,
ולשמור על שליטה מלאה עם Cancellation ו-Semaphore.

השילוב של **Semaphore**, **gather**, **async**, ו-**Retry**
הוא לא טריק של מתכנתים מתקדמים. זהו הסטנדרט.
כל מערכת AI אמיתית.

בין אם היא מנהלת קריאות ל-OpenAI, ל-Gemini או למנוע אחזור פנימי.

חייבת להיות אסינכרונית כדי להישאר יציבה, מהירה ויעילה.

במילים פשוטות:
בלי async, כל מערכת AI תהפוך לצוואר בקבוק.
עם async, היא הופכת לרשת חכמה של משימות שמדברות זו עם זו במקביל,
חוסכות זמן, ומפיקות יותר תובנות בפחות משאבים.

# פרק 16 – ממשק שורת פקודה (CLI)

## למה CLI חשוב בפרויקטי AI

בפרויקטי **AI**, גם הקוד הכי חכם חסר ערך אם אי אפשר **להפעיל אותו בקלות**.
ממשק שורת הפקודה (**CLI**) הוא הדרך להפוך קוד גולמי **לכלי אמיתי**
כזה שאפשר **להריץ, לבדוק ולשלב** בתהליכים אחרים בלי לפתוח את העורך.

CLI הוא לא שריד מעולם ישן, אלא **שכבת השליטה הטבעית** של פרויקטים חכמים.
הוא מעניק דרך **יציבה, מהירה ואחידה** להפעיל תהליכים. בין אם מדובר בהרצת מודלים, ניקוי טקסטים או ניתוח נתונים.

נניח שבניתם כלי שמנקה טקסטים לפני שליחה למודל.
בלי CLI, צריך לפתוח את הקובץ ולהריץ פונקציות מתוך הקוד.
עם CLI, זה נראה כך:

```
mintx clean data/articles.csv --lang he
```

או

```
mintx stats output/cleaned.csv
```

פקודה אחת, והתהליך רץ **מההתחלה ועד הסוף**.

## למה זה כל כך חשוב בפרויקטי AI

- **אוטומציה קלה:**
כל פקודת CLI ניתנת לשילוב ישיר בתוך scripts, pipelines או cron jobs. כך בונים מערכות **שעובדות לבד**.
- **עקביות בין סביבות:**
אותה פקודה עובדת **בלפטופ, בענן או בתוך Docker**, בלי לשנות שורה אחת של קוד.
- **נוחות לשיתוף:**
אפשר למסור כלי לאחרים: חוקרים, אנשי דאטה, אוDevOps, והם יוכלו להשתמש בו **בלי לדעת איך הוא כתוב**.
- **מודולריות וניקיון:**
CLI מפריד בין **הלוגיקה העסקית** לבין **דרך ההפעלה**, ומאפשר לבנות קוד גמיש שקל לתחזק.

בסופו של דבר, CLI הוא **המפתח שהופך קוד לריצה אמיתית**.
הוא מעניק לפרויקט שלכם נוכחות בעולם. מאפשר **להריץ, למדוד, לשתף ולשלב** וכל זאת דרך פקודה אחת ברורה ונקייה.

## argparse – הכלי המובנה

כשאנחנו מריצים סקריפט פייתון, לדוגמה:

```
python clean_text.py
```

הקובץ רץ, אבל אין לו מושג **מאיפה לקרוא קובץ**, **לאן לשמור תוצאה**, או **באיזו שפה להשתמש**.

אם נרצה להפעיל את אותו סקריפט עם קלטים שונים בכל פעם, נצטרך דרך **להעביר לו פרמטרים מבחוץ**, בלי לשנות את הקוד בכל הרצה.

כאן נכנסת לתמונה **argparse**.

זו מערכת קטנה ומובנית בפייתון שתפקידה אחד:

### לאפשר לקוד שלך להבין פקודות מהמשתמש

בדיוק כמו שעושים כלים מוכרים כמו git, pip או docker.

במילים פשוטות, argparse היא המוח שמתרגם את מה שכתבת בשורת הפקודה לערכים שהקוד שלך מבין.

### למה צריך אותה בכלל

בלי argparse, הדרך היחידה לדעת מה המשתמש כתב היא לבדוק את רשימת המילים אחרי שם הקובץ, שנמצאת במשתנה sys.argv:

```
import sys

input_path = sys.argv[1]
```

```
output_path = sys.argv[2]
```

זה עובד, אבל זה גם **שביר ומסורבל**.

אם המשתמש שכח להזין פרמטר, הקוד קורס.

אם הוא רוצה לדעת אילו אפשרויות קיימות, אין לו מושג.

ואם נוסיף פרמטר חדש, צריך לשנות שוב את הקוד.

`argparse` פותרת את כל זה באופן אלגנטי:

- **מפרשת** את כל מה שנכתב אחרי שם הקובץ (`input.txt`, `--verbose`, `--lang` וכו׳).
- **בודקת תקינות** האם חסר פרמטר? האם סוג הערך נכון?
- **מייצרת עזרה אוטומטית** (`--help`) שמסבירה למשתמש איך להשתמש בכלי.
- **מטפלת בשגיאות** בצורה ברורה, בלי Traceback מבולגן.

### דוגמה פשוטה

```
import argparse

parser = argparse.ArgumentParser(description=" ניקוי טקסטים לפני עיבוד AI")
parser.add_argument("input", help="נתיב לקובץ המקור")
parser.add_argument("output", help="נתיב לשמירת התוצאה")
parser.add_argument("--lang", default="he", help=" שפת הטקסט (he/en)")
args = parser.parse_args()
```

```
print(f"קורא מ-{args.input} ושומר ל-{args.output} (שפה: {args.lang})")
```

הרצה:

```
python clean_text.py data/raw.csv data/clean.csv --lang en
```

פלט:

```
קורא מ- data/raw.csv ושומר ל- data/clean.csv (שפה: en)
```

הרצה שגויה (למשל בלי אחד הפרמטרים):

```
python clean_text.py data/raw.csv
```

פלט:

```
usage: clean_text.py [-h] [--lang LANG] input output
clean_text.py: error: the following arguments are required: output
```

## למה זה משנה

ממשק שורת הפקודה (**CLI**) הוא **השפה שבה המשתמש מדבר עם הקוד שלך**.

argparse היא **המתורגמנית**

היא הופכת את מילת הפקודה (--lang en)

למשתנה בקוד (args.lang = "en").
היא מאפשרת לך **לבנות כלי שניתן להשתמש בו שוב ושוב**, עם פרמטרים שונים, בלי לגעת בקוד הפנימי.

## בקצרה

אם תכתבו סקריפט בלי argparse – יש לכם **קוד**.
אם תכתבו אותו עם argparse – יש לכם כלי **אמיתי**.

argparse היא **הגשר בין המשתמש לקוד**, בין מה להריץ לאיך להבין את זה.
בזכותה, כל פרויקט פייתון יכול להפוך ליישום קטן, **יציב, גמיש ונוח להרצה מכל מקום**.

## Typer – הכלי המודרני

אם argparse היא הוותיקה והאמינה, אז **Typer** היא הדור החדש.
היא נבנתה על ידי Sebastián Ramírez (יוצר **FastAPI**) במטרה אחת:
לאפשר למתכנתים לבנות ממשקי CLI **קריאים, חכמים ומוקפדים**, תוך שימוש ב-**type hints** של פייתון.

בעוד ש-argparse מחייבת להגדיר כל פרמטר ידנית, Typer מזהה את סוג המשתנים שלך, יוצרת תיעוד אוטומטי, ומפיקה CLI נקי כמעט בלי תצורה.

## למה בכלל נוצר Typer

עם השנים, מתכנתים התחילו לדרוש מ-CLI יותר נוחות ואלגנטיות:

- כתיבה מהירה בלי הגדרות כפולות.
- טיפוסי נתונים ברורים (int, str, Path).
- תיעוד אוטומטי וקריא.
- תמיכה בפקודות משנה (כמו git add, git commit).

Typer נבנתה בדיוק לשם כך. היא **שכבת CLI מודרנית מעל Click**, ספרייה ותיקה שמאפשרת ניהול פקודות מתקדמות.
אבל Typer עושה משהו מעבר: היא **מתאימה את עצמה למבנה הפונקציות שלך**.

## לדוגמה

```
import typer
from pathlib import Path

app = typer.Typer(help="ניקוי טקסטים לפני עיבוד AI")

@app.command()
def clean(input: Path, output: Path, lang: str = "he"):
    """
    מנקה טקסטים מקובץ קלט ושומר לקובץ פלט.
    """
```

מאת: תומר קדם

```
    typer.echo(f"קורא מ-{input}, שומר ל-{output} (שפה: {lang})")

if __name__ == "__main__":
    app()
```

הרצה:

```
python clean_text.py clean data/raw.csv data/clean.csv --lang en
```

פלט:

```
קורא מ-data/raw.csv, שומר ל-data/clean.csv (שפה: en)
```

## מה קרה כאן

- **@app.command()** הופך כל פונקציה לפקודה עצמאית ב-CLI.
- Typer קוראת את **רמזי הטיפוס (type hints)** ומייצרת מהם ממשק חכם:
  - אם המשתנה הוא `Path`: היא תוודא שהקובץ קיים.
  - אם הוא `int`: תתריע על טקסטים שאינם מספרים.
  - פרמטרים עם ערכי ברירת מחדל (`lang="he"`) מזוהים אוטומטית כפרמטרים אופציונליים.
  - הכול מגיע עם **תיעוד מיידי וקריא**.

## עזרה מובנית

בדיוק כמו ב-Typer ,argparse תומכת בפקודת help--:

```
python clean_text.py clean --help
```

פלט:

```
Usage: clean_text.py clean [OPTIONS] INPUT OUTPUT

  מנקה טקסטים מקובץ קלט ושומר לקובץ פלט.

Arguments:
  INPUT   [required]
  OUTPUT  [required]

Options:
  --lang TEXT  שפת הטקסט (ברירת מחדל: he)
  --help       הצג עזרה ויציאה
```

## למה מתכנתים אוהבים את Typer

**כתיבה קצרה וקריאה:** במקום להגדיר parser וארגומנטים, פשוט כותבים פונקציה רגילה.

- **טיפוסי נתונים מובנים:**

  סוגי המשתנים כבר מגדירים את אופי הפרמטרים.

- **תיעוד אוטומטי:**

כל פונקציה מתועדת לבד בעזרת ה-docstring שלה.

- **תמיכה בפקודות משנה (subcommands):**
מושלם לכלים מורכבים כמו mintx stats, mintx clean ועוד.
- **אינטגרציה טבעית עם FastAPI:**
מי שמכיר את FastAPI ירגיש בבית  אותה פילוסופיה, אותה נוחות.

## מתי לבחור Typer

- כשאתם בונים **כלי CLI עם יותר מפקודה אחת**.
- כשאתם רוצים **תחזוקה פשוטה וקריאות גבוהה**.
- כשאתם עובדים בצוותים ומעדיפים קוד שנראה כמו API ולא כמו קונפיגורציה.

Typer הפכה בתוך זמן קצר לסטנדרט החדש של פרויקטיCLI מודרניים.

היא לא רק מקלה על הכתיבה, היא **מקרבת את עולם הפקודות לעולם הקוד**,
ומאפשרת לבנות ממשקים חכמים, מתועדים וברורים. כמעט בלי מאמץ.

## פקודות משנה – Subcommands

כלי CLI אמיתי כולל לרוב יותר מפעולה אחת.

במקום קובץ נפרד לכל משימה, נוח לרכז הכול תחת ממשק אחד עם **פקודות משנה** – בדיוק כמו ב-`git commit`, `git add` או `pip install`.

ב-Typer זה פשוט במיוחד:

```
import typer
app = typer.Typer(help="כלי מיני-טקסט (mintx) לעיבוד טקסטים")

@app.command()
def clean(input: str, output: str):
    typer.echo(f"ניקוי טקסטים: {input} → {output}")

@app.command()
def stats(file: str):
    typer.echo(f" סטטיסטיקות על{file}")

if __name__ == "__main__":
    app()
```

כעת ניתן להריץ:

```
python mintx.py clean data/raw.csv out.csv
python mintx.py stats out.csv
```

כל פקודה פועלת בנפרד, עם פרמטרים שונים, אך חולקת אותו בסיס קוד ותיעוד.

## למה זה חשוב

- מאפשר לאחד מספר כלים קטנים לכלי אחד ברור.
- קל לתחזוקה – אין שכפול קוד.
- מונע בלבול בשמות קבצים או סקריפטים.

כך CLI הופך ממספר סקריפטים לכלי שלם, מסודר וברור.

שלב חשוב בכל פרויקט AI אמיתי.

## קודי יציאה (Exit Codes) – 0 מול שאר

מאחורי הקלעים, כל תוכנית CLI מסיימת את פעולתה עם **קוד יציאה** מספר שמסמן למערכת האם הפעולה הצליחה או נכשלה. זה אולי נראה פרט טכני, אבל בפרויקטי AI (ובעיקר באוטומציה ו-pipelines) הוא **הקו שמפריד בין תהליך תקין לשגוי**.

הכלל פשוט:

- **0 – הצלחה.**
- **כל מספר אחר** – שגיאה כלשהי.

ב-Typer (וגם ב-argparse) ניתן לקבוע זאת בקלות:

```
import typer

def main(file: str):
```

```
    if not file.endswith(".csv"):
        typer.echo("שגיאה: רק קבצי CSV נתמכים")
        raise typer.Exit(code=1)
    typer.echo("עיבוד הסתיים בהצלחה")
    raise typer.Exit(code=0)

if __name__ == "__main__":
    typer.run(main)
```

כעת, מי שיריץ את הכלי מתוך סקריפט אחר יוכל לדעת אם הכול עבר בשלום:

```
mintx clean data.txt || echo "הרצה נכשלה!"
```

אם הקובץ לא חוקי. הפקודה תסתיים עם קוד 1, והמערכת תזהה זאת מיד.

## למה זה חשוב

מאפשר ל-scripts ול-CI להבין אם השלב הצליח.
משפר דיווחי שגיאות ב-pipelines.
עוזר לתחזק כלים יציבים שניתן לסמוך עליהם בתהליכים אוטומטיים.

CLI טוב לא רק מדפיס הודעה, הוא גם מסמן אותה לקוד שמריץ אותו.
זה ההבדל בין תוכנה אינטראקטיבית לבין רכיב אמין בשרשרת אוטומציה.

## תיעוד אוטומטי (--help)

כלי CLI טוב לא דורש מדריך.
הוא מסביר את עצמו ברגע שמקלידים:

```
mintx --help
```

גם argparse וגם Typer מייצרים תיעוד אוטומטי שמציג את כל הפקודות, הארגומנטים והאפשרויות הקיימות, יחד עם הסבר קצר על כל אחד מהם.
זו לא תוספת קוסמטית, זו שכבת **שקיפות והנגשה** שהופכת כלי CLI לשימושי באמת.

ב-Typer, למשל, זה קורה בלי שום מאמץ:

```
python mintx.py --help
```

פלט:

```
Usage: mintx clean [OPTIONS] INPUT OUTPUT

ניקוי טקסטים מקובץ קלט ושמירה לקובץ פלט.

Arguments:
  INPUT   [required]
  OUTPUT  [required]

Options:
  --lang TEXT  שפת הטקסט (he/en)
  --help       הצג עזרה ויציאה
```

## למה זה חשוב

- **חוסך תיעוד חיצוני** הכלי מתעד את עצמו.
- **מונע טעויות משתמשים** אין צורך לזכור פרמטרים.
- **מקרין מקצועיות** כלי שמסביר את עצמו נראה מושלם גם בעיני מי שלא כתב אותו.

התוצאה היא CLI שמכבד את המשתמש, כלי שמסביר בדיוק מה הוא יודע לעשות, עוד לפני שמישהו פותח את הקוד.

## דוגמה מרכזית – CLI מלא למיני-טקסט (mintx)

כעת נחבר הכול לכלי אחד שלם:

ממשק שורת פקודה שמאפשר להריץ פעולות שונות על טקסטים, ניקוי, חישוב סטטיסטיקות, ועוד.

```
# mintx.py
import typer
from pathlib import Path

app = typer.Typer(help="mintx – כלי CLI  קטן לעיבוד טקסטים בפרויקטי AI")

@app.command()
def clean(input: Path, output: Path, lang: str = "he"):
    """
```

```
    מנקה טקסטים מקובץ קלט ושומר את התוצאה בקובץ פלט.
    """
    typer.echo(f"ניקוי טקסטים ({lang}) מ-{input} → {output}")
    # כאן תתבצע קריאה לפונקציה האמיתית: clean_text(input,
output, lang)
    typer.echo("הניקוי הושלם בהצלחה ✅")

@app.command()
def stats(file: Path):
    """
    מחשב סטטיסטיקות בסיסיות על טקסטים.
    """
    typer.echo(f" קורא טקסטים מתוך{file}")
    # כאן תתבצע קריאה לפונקציה: compute_stats(file)
    typer.echo("הממוצעים חושבו ונשמרו 📊")

if __name__ == "__main__":
    app()
```

הרצות לדוגמה:

```
python mintx.py clean data/raw.csv data/clean.csv --lang en
python mintx.py stats data/clean.csv
```

או בקיצור (לאחר התקנה מקומית):

```
python mintx.py clean data/raw.csv data/clean.csv --lang en
python mintx.py stats data/clean.cs
```

## מה מקבלים כאן

- **פקודות משנה (Subcommands):** `stats`, `clean`.
- **תיעוד אוטומטי:** `mintx --help` מציג עזרה מלאה.
- **קודי יציאה ברורים:** אפשר להחזיר raise typer.Exit(1) במקרה של שגיאה.
- **מודולריות מלאה:** כל פקודה מופרדת לפונקציה, כך שקל להרחיב בהמשך.

הכלי הזה קטן, אבל הוא כבר **התשתית של מערכת אמיתית**: אפשר לשלב אותו ב-pipeline, להריץ אותו מתסריט אוטומטי, או למסור אותו לחוקרים וצוותי דאטה בלי הסברים מיותרים.

## Best Practices – שמות ברורים ו-Defaults הגיוניים

ממשק CLI טוב הוא לא רק פונקציונלי, הוא גם נעים לשימוש.
כשמפתחים כלי שאחרים יריצו, חשוב לזכור: המשתמש לא רואה את הקוד, הוא רואה **פקודות**.
כל מילה חשובה.

### שמות פקודות

בחרו שמות **ברורים וקצרים**.
עדיף פועל ברור אחד, שמייצג פעולה:

✓ `clean`, `stats`, `train`, `serve`

✗ `text_cleaning`, `run_statistics_now`

אם יש פקודות דומות, שמרו על אחידות:

```
mintx clean ...
mintx stats ...
mintx export ...
```

### שמות פרמטרים

פרמטרים טובים הם אינטואיטיביים:

- `input`, `--output`, `--lang`, `--model--`
- לעולם אל תשתמשו בקיצורים מבלבלים כמו `-in–` או `op--`
העדיפו שמות מלאים גם אם הם ארוכים במעט. הם נקראים פעם אחת, אבל מבטיחים שימוש נכון.

## ערכי ברירת מחדל (Defaults)

ברירת מחדל טובה חוסכת הקלדה מיותרת ומונעת תקלות:

```
def clean(input: Path, output: Path = Path("output.csv"), lang:
str = "he"):
```

כך המשתמש יכול להריץ רק:

```
mintx clean data.csv
```

והכלי כבר יידע לשמור ל-output.csv בעברית.

## הודעות פלט

CLI נוח גם **מדבר יפה.**

לא רק "הסתיים", אלא גם "מה קרה":

```
typer.echo(f"נוקו 324 שורות (שפה: he)")
```

כשמדובר בכלים הנדסיים, חוויית שימוש שקטה וברורה עושה הבדל גדול.

## קונסיסטנטיות

שמרו על אחידות בין פקודות, פרמטרים והודעות.

משתמש שמכיר פקודה אחת, צריך להבין את כולן מיד.

כלל הזהב:

**כלי CLI טוב הוא כזה שהמשתמש מצליח להבין בלי לקרוא תיעוד.**

אם השמות, ברירות המחדל וההודעות שלכם עומדים בכך.

הצלחתם לבנות כלי אמיתי, לא רק סקריפט שעובד.

## סיכום – איך להפוך קוד לכלי שימושי

CLI הוא לא קישוט, אלא שכבת שליטה שמעניקה לקוד שלכם **חיים אמיתיים מחוץ לעורך**.

בעולם של AI, שבו סקריפטים מתמזגים עם תהליכים אוטומטיים, זה ההבדל בין קוד שעובד רק אצלכם לבין כלי שיכול לעבוד בכל מקום.

במהלך הפרק ראינו:

- איך **argparse** מספקת בסיס יציב ונטול תלות לבניית CLI פשוט.
- איך **Typer** מאפשרת ליצור ממשקים אלגנטיים בעזרת פונקציות רגילות ו-type hints.
- איך **פקודות משנה (Subcommands)** מאחדות כמה פעולות לכלי אחד מסודר.
- איך **קודי יציאה** מאפשרים לסקריפטים לזהות הצלחה או כישלון בצורה אוטומטית.
- ואיך **תיעוד אוטומטי ו-defaults חכמים** הופכים כל כלי לקל לשימוש גם עבור אחרים.

הכוח האמיתי של CLI הוא בפשטות:
פקודה אחת, פרמטר אחד, והרבה בהירות.

כשכלי ה-AI שלכם מגיע לשלב שבו אחרים צריכים להריץ אותו.

בין אם זה אנליסט, חוקר או שרת אוטומטי

CLI הוא הדרך **להפוך את הקוד למוצר קטן, יציב ונגיש**.

מכאן והלאה, כל מודול שתכתבו יכול להפוך לפקודה, וכל פרויקט יכול להפוך לכלי שלם – אחד שמדבר בשפה אנושית וברורה.

# פרק 17 – תרגול מאוחד: בניית mini_text_analyzer

הגענו לרגע שבו כל הידע מתלכד לפרויקט אחד שלם.
בפרקים הקודמים למדתם את עקרונות השפה, מבני הנתונים, ניהול קבצים, לוגים, מודולריות, טיפוסים, דקורטורים, בדיקות ו-CLI.
עכשיו נבנה מהם יחד את **mini_text_analyzer**, פרויקט קונספטואלי קטן, אך עם מבנה וסטנדרטים של מערכת אמיתית.

המטרה כאן אינה רק לכתוב קוד שעובד, אלא **לראות איך כל החלקים מתחברים לתמונה אחת, תהליך מלא של עיבוד טקסטים במבנה פרויקט הנדסי שלם.**

## תרגיל 1 – מבנה פרויקט ראשוני

צרו תיקייה בשם `mini_text_analyzer` עם מבנה תקני:

```
mini_text_analyzer/
├──── src/
│     ├──── mini_text_analyzer/
│     │     ├──── __init__.py
│     │     └──── text_pipeline.py
├──── tests/
│     └──── test_pipeline.py
├──── data/
│     └──── sample.txt
├──── requirements.txt
```

```
└─── README.md
```

אתחלו סביבה וירטואלית:

```
python -m venv .venv
source .venv/bin/activate  # (או .venv\Scripts\activate ב-Windows)
```

אתחלו גם Git:

```
git init
echo ".venv/" > .gitignore
```

## תרגיל 2 – טוקניזציה וניקוי בסיסי (פרקים 2–4)

צרו בקובץ `text_pipeline.py` פונקציות `normalize()` ו-`tokenize()` על פי שלמדתם:

- ניקוי רווחים, הורדת אותיות גדולות.
- פיצול למילים בעזרת ביטוי רגולרי.

ודאו שהן מחזירות רשימת מילים תקנית.

## תרגיל 3 – מודולים וארגון (פרק 5)

ארגנו את הפונקציות בתוך מודול נפרד `text_utils.py` וייבאו אותן למחלקה הראשית.

וודאו שכל מודול אחראי על פעולה אחת בלבד – מבנה נקי וברור.

## תרגיל 4 – קונפיגורציה חיצונית (פרק 7)

צרו קובץ `config.json` עם הגדרות ברירת מחדל (שפה, מיקום קובצי קלט ופלט).

בנו פונקציה שטוענת את ההגדרות בצורה גנרית:

```
import json, pathlib

def load_config(path: str = "config.json") -> dict:
    return
json.loads(pathlib.Path(path).read_text(encoding="utf-8"))
```

## תרגיל 5 – חריגות ולוגים מובנים (פרק 8)

הוסיפו טיפול בשגיאות עם `try/except` ולוגים באמצעות מודול `logging`:

- לוג מידע (INFO) בכל שלב.
- לוג שגיאות (ERROR) במקרה של חריגה.

הקפידו שכל הודעה תכלול זמן, שלב ופרטים רלוונטיים.

## תרגיל 6 – OOP: מחלקת TextPipeline (פרק 9)

צרו מחלקה:

```
class TextPipeline:
  def __init__(self, config: dict):
    self.config = config

  def clean(self, text: str) -> str:
    ...

  def stats(self, text: str) -> dict:
    ...
```

המחלקה מאחדת את כל השלבים: קריאה, ניקוי, ניתוח, וכתיבה.

## תרגיל 7 Type hints – מלאים (פרק 10)

הוסיפו **type hints** לכל פונקציה ומחלקה:

```
def stats(self, text: str) -> dict[str, float]:
```

כך תקבלו אוטוקומפלישן מדויק ועזר למבקרים סטטיים.

## תרגיל 8 – דקורטורים למדידה (פרק 11)

הוסיפו דקורטור בשם measure_time שמודד כמה זמן נמשכה כל פעולה.

שלבו אותו על פונקציות הניקוי והסטטיסטיקה.

## תרגיל 9 – בדיקות אוטומטיות (פרק 12)

צרו בדיקות יחידה בסיסיות בקובץ tests/test_pipeline.py עם pytest:

- בדקו שטוקניזציה מחזירה רשימה.
- בדקו שפונקציית stats מחזירה מפתחות צפויים.

## תרגיל 10 – אופטימיזציה עם Pandas (פרקים 13–14)

הוסיפו שלב שבו ()stats אוספת נתונים ל-DataFrame ומחשבת ממוצעים.
ודאו שאתם משתמשים ב-pandas בצורה יעילה.

## תרגיל 11 – CLI עם Typer (פרקים 15–16)

בנו קובץ mintx.py עם פקודות:

```
mintx clean input.txt output.txt
mintx stats output.txt
```

הקפידו על תיעוד (--help) וקודי יציאה תקינים.

## תרגיל 12 – הרכבה סופית + CI/CD

השלימו את הפרויקט:

- ודאו שכל הקוד ממודר וקריא.
- הפעילו pytest לוודא שכל הבדיקות עוברות.
- הגדירו GitHub Actions להרצת הבדיקות אוטומטית עם כל commit.

## פתרון מלא

הפתרון המלא (זמין בריפוזיטוריון GitHub המצורף) תמצאו:

- קוד שלם לכל השלבים.
- מבנה פרויקט הנדסי אמיתי.
- דוגמה להרצת pipeline אמיתי מקובץ טקסט גולמי ועדCLI פועל.

זהו הסיום המעשי של המסע:
מ-שורה בודדת של קוד ועד מערכת שלמה, מתועדת, נבדקת, נמדדת וניתנת להרצה בכל סביבה.
**mini_text_analyzer** הוא אולי פרויקט קטן, אבל הוא הדגם הקלאסי של איך בונים כלי הנדסי חכם בעולם ה-AI.

# פרק 18 – תבנית פרויקט Production ל-AI Engineer

בפרקים הקודמים בנינו את **mini_text_analyzer** כמודול עובד.
כעת נלמד איך להפוך אותו, וכל פרויקט אחר, לשלד הנדסי מלא שמוכן לפריסה אמיתית.

**הפרק הזה הוא מדריך תשתיתי:**
הוא מלמד איך לארגן קוד, קונפיגורציה, תיעוד ובדיקות כך שהכול ירגיש כמו מוצר של צוות פיתוח אמיתי.

## למה להתחיל עם שלד מסודר

פרויקט AI הוא לא רק מודל או קובץ פייתון, הוא מערכת.
שלד נכון מראש חוסך אינספור באגים, כפילויות והפתעות מאוחרות.

המטרה היא אחת: **לאפשר לך לפתח, לבדוק ולפרוס באותו מבנה קבוע**.

גם פרויקט קטן שמתחיל בתיקייה אחת, עדיף שייבנה כשלד production מהיום הראשון.

כך כל מי שיצטרף אחריך יוכל להבין את הקוד תוך דקות, ולא תוך שבוע.

## מבנה תיקיות מומלץ

המבנה הבסיסי לפרויקט AI הנדסי:

```
project_root/
├──── config/       ← קבצי הגדרות JSON/YAML
├──── data/         ← קלטים, פלטים ודוגמאות
├──── src/          ← קוד המקור
│     └──── project_name/
│        ├──── __init__.py
│        └──── core_modules.py
├──── tests/        ← בדיקות אוטומטיות
├──── scripts/      ← קבצי הרצה וכלים פנימיים
├──── requirements.txt ← ספריות נדרשות
├──── README.md     ← הסבר ותיעוד ראשוני
└──── .gitignore
```

זהו שלד קלאסי שמקל על ניהול גרסאות, בדיקות ופריסה לכל סביבה.

## קונפיגורציה: JSON / YAML + משתני סביבה

אל תשמרו נתיבים, מפתחות או פרמטרים ישירות בקוד.
השתמשו בקבצי **config.json** או **config.yaml** וב-**os.environ** לטעינת משתנים רגישים (כמו API keys).
דוגמה:

```
import json, os

with open("config/config.json", encoding="utf-8") as f:
    cfg = json.load(f)

api_key = os.getenv("OPENAI_API_KEY")
```

כך הפרויקט נשאר נייד – כל סביבה יכולה לספק קובץ הגדרות משלה.

## scripts: אוטומציה של הרצות

במקום לזכור פקודות ארוכות, צרו תיקייה `scripts/` עם קבצים קטנים:

```
scripts/
 ├──── run_clean.sh
 ├──── run_stats.sh
 └──── train_model.py
```

כל קובץ מפעיל פעולה אחת ברורה.
זו הדרך הנכונה להפוך פרויקט AI מתהליך ידני למערכת ניתנת להרצה מתוזמנת.

## בדיקות: תיקיית tests + pytest

כל פרויקט production צריך בדיקות יחידה ובדיקות אינטגרציה.
pytest מאפשר לבדוק גם פונקציות בודדות וגם זרימות מלאות.
שמרו על מבנה זהה:

```
tests/
├── test_clean.py
├── test_stats.py
└── conftest.py
```

הרצה:

```
pytest -v
```

בדיקות הן לא רק הגנה, הן **הוכחה שהפרויקט שלכם בשליטה**.

## תיעוד: README + Docstrings + MkDocs

תיעוד הוא חלק מהקוד.
כל מודול צריך **docstring ברור**, וכל פרויקט צריך **README.md קצר וקריא**:

- מה הכלי עושה
- איך מתקינים
- איך מריצים

לפרויקטים גדולים יותר השתמשו ב-**MkDocs** כדי לבנות אתר תיעוד אוטומטי מ-GitHub.

## Git ו-gitignore.: מה לא לשמור

אל תשמרו קבצים שנוצרים אוטומטית:

```
__pycache__/
*.pyc
.venv/
data/
.env
config/*.local.json
```

כך הקוד נשאר נקי ומשקל הריפו קטן.
זכרו, Git אמור להכיל רק את מה שצריך כדי לבנות את הפרויקט מחדש מאפס.

## Dockerfile מינימלי

כדי שהפרויקט יעבוד זהה בכל מחשב, צרו **Dockerfile פשוט**:

```
FROM python:3.12-slim
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["python", "src/project_name/main.py"]
```

הרצה:

```
docker build -t mini_text_analyzer .
docker run mini_text_analyzer
```

כעת הפרויקט ניתן לפריסה בכל מקום. מקומי, שרת או ענן.

## CI/CD בסיסי

ב-GitHub Actions או Azure DevOps הגדירו pipeline שמריץ בדיקות בכל commit:

```yaml
name: mini_text_analyzer CI
on: [push]
jobs:
 test:
  runs-on: ubuntu-latest
  steps:
   - uses: actions/checkout@v4
   - uses: actions/setup-python@v5
     with:
      python-version: "3.12"
   - run: pip install -r requirements.txt
   - run: pytest
```

זהו שלב קטן שהופך פרויקט למקצועי באמת, לא משנה מי לוחץ "commit", הכול נבדק אוטומטית.

## דוגמה מרכזית: mini_text_analyzer כתבנית לפרויקט חדש

`mini_text_analyzer` כבר כולל את כל הרכיבים האלו

מבנה תיקיות, קונפיגורציה, CLI ובדיקות.

פשוט העתיקו את השלד שלו כנקודת פתיחה לפרויקט הבא שלכם.
החליפו את השם בתיקיות, התאימו את הקונפיגורציה, והמערכת מוכנה לעבודה.

## Best Practices

- ✓ שמרו על **הפרדה חדה** בין קוד, נתונים, קונפיגורציה ולוגים.
- ✓ השתמשו באוטומציה לכל תהליך שחוזר על עצמו.
- ✓ הקפידו על תיעוד קצר אך עקבי.
- ✓ הריצו בדיקות לפני כל commit.
- ✓ עדכנו את התלויות (requirements.txt) באופן קבוע.

## סיכום הספרון – מה למדנו ואיך להמשיך

בספרון זה ראיתם איך להפוך ידע בפייתון למקצוע אמיתי בעולם ה-AI:
מהבנת מבני נתונים ועד בניית כלי CLI שלם.
הבנתם איך לארגן קוד, לבדוק אותו, לתעד אותו, ולפרוס אותו כמו מהנדס תוכנה אמיתי.

מכאן, הצעד הבא הוא לקחת את הידע הזה לפרויקטים אמיתיים: להשתמש בתבנית production שבניתם, וליצור ממנה כלים שמשרתים משתמשים אמיתיים.
זוהי תחילת הדרך כמהנדס AI לא סקריפטיסט, אלא **בונה מערכות מלאות שחושבות, רצות ומתוחזקות לאורך זמן.**