פייתון בפרוד: הרמה הבאה

מקוד שעובד לקוד שניתן לתחזק, לבדוק ולהריץ בפרוד

יואב חיימוב

הקדמה

ברוכים הבאים וברוכות הבאות ל־פייתון בפרוד: הרמה הבאה.

אם אתם קוראים את השורות האלה, רוב הסיכויים שכבר כתבתם בפייתון קוד שעובד. וסבלתם גם מכזה שהפסיק לעבוד בשתיים לפנות בוקר. הספר הזה נועד למפתחות ולמפתחים שכבר שולטים היטב בפייתון (ב־OOP, ב־dataclasses, ב־generators, ב־decorators בסיסיים), ורוצים לעבור מקוד ש"עובד" לקוד שניתן לתחזק, לבדוק ולהריץ. זוהי הרמה הבאה: לא עוד סקריפטים חד־פעמיים, אלא פרויקטים אמיתיים עם טיפוסים שמגינים עלינו, שגיאות שניתן לאבחן, משאבים שמנוהלים כראוי, קוד מקבילי בטוח, בדיקות שנותנות ביטחון וארכיטקטורה שקל להחליף בה חלקים בלי לשבור את השלם.

המבנה הוא הדרגתי: כל פרק עומד בפני עצמו; ולכן אפשר לקרוא אותו גם מחוץ לסדר. יחד עם זאת, מומלץ לעבור על הפרקים לפי הסדר, משום שהם בנויים זה על גבי זה: הטיפוסים שבפרק הראשון חוזרים ב־Dependency Injection שבפרק השישי; ולוגינג מובנה צץ שוב בפרק על מוכנות בפרוד. בכל פרק תמצאו את אותו דגם חוזר: anti-pattern לעומת הדרך הטובה יותר, דוגמאות קוד מלאות וקצרות, ותרגיל קצר אחד לפחות שניתן לפתור בעצמכם לפני שממשיכים. הדגם החוזר הזה אינו תאונה. כמו בקוד, גם בספר חזרתיות טובה היא תכונה, לא באג.

הקוד בכל הפרקים נכתב לפי פייתון 3.10 ומעלה, עם דגש על תכונות של 3.11 ו־3.12 (כמו Exception Groups ו־Task Groups מ־3.11, ו־entry points בלי select מ־3.12) במקומות שהן רלוונטיות. חלק מהדוגמאות מניחות התקנה של חבילות צד־שלישי כגון structlog, pytest ו־hypothesis; הפרקים הרלוונטיים מציינים זאת במפורש. וזכרו: 3.10 היא הרצפה, לא התקרה: כשיהיה לכם פייתון חדש יותר, תקבלו עוד כלים. הרעיונות נשארים.

קריאה מהנה. ואני מאחל לכם דיפלואים משעממים ככל האפשר — זה הדבר הכי טוב שיכול לקרות לפרויקט.

תוכן העניינים

הקדמה I

פרק 1: טיפוסים מתקדמים ו־Protocols 1

פרק 2: טיפול תקין בשגיאות ולוגינג מובנה 11

פרק 3: ניהול משאבים ו־Context Managers 23

פרק 4: תכנות מקבילי נכון 31

פרק 5: חשיבה ביצועית 39

פרק 6: ארכיטקטורה נקייה ו־Dependency Injection 49

פרק 7: אריזה ומבנה פרויקט 57

פרק 8: אסטרטגיות בדיקות מתקדמות 65

פרק 9: מוכנות בפרוד ו־Observability 75

פרק 10: מודל האובייקטים של פייתון 83

פרק 11: כלי פונקציונליים: functools ו־itertools 101

פרק 12: קשירת שמות, סקופינג וסגירות 119

פרק 13: מדריך בחירת אוספים **133**

פרק 14: מכניקת ייבוא וטעינה עצלה **147**

פרק 15: סיכום ופרויקט מסכם **165**

נספח: כלים, קישורים ותבניות **173**

מילון מונחים **179**

Acronyms **179**

פרק 1: טיפוסים מתקדמים ו־Protocols

מערכת הטיפוסים של פייתון היא המקום שבו השפה מפתיעה מפתחי backend מנוסים. ברוב השפות היא עול בירוקרטי; בפייתון היא כלי אמיתי למניעת באגים, כשמשתמשים בה נכון. בפרק זה נלמד על Protocol (התאמה מבנית), TypedDict (מילונים מובנים), TypeVar (משתנה טיפוס) ו־ParamSpec (מפרט פרמטרים), ועל הדרך שבה Subtyping Structural (תת-טיפוס מבני) משנה את העיצוב של interfaces. כל זאת עם דוגמאות מ־repositories, services ו־plugins; המקומות שבהם הטיפוסים מצילים מבאגים מביכים בפרוד.

1.1 סוגים נומינליים לעומת מבניים

ב-Java או C++, שני טיפוסים תואמים רק אם יש ביניהם קשר יורש מפורש. זהו Nominal Subtyping (תת-טיפוס נומינלי): השם של הטיפוס הוא זה שקובע. בפייתון, לעומת זאת, מערכת ה־type hints תומכת בהתאמה מבנית (structural subtyping) טיפוס תואם ל־interface אם הוא כולל את האיברים שהחוזה (ה־contract) דורש: בלי קשר לשם שלו ובלי ירושה מפורשת.

הגדרה 1.1 (התאמה מבנית)**.** טיפוס X הוא תת־טיפוס מבני של Y אם לכל איבר שנדרש מ־Y קיים איבר תואם ב־X. המערכת בודקת את המבנה, לא את שורת הירושה.

מדוע זה חשוב? כי ירושה היא קשר חזק ויקר: היא קושרת יחידות קוד שעשויות להשתנות בקצבים שונים. לעומת זאת, התאמה מבנית מאפשרת לשתי מחלקות שאין ביניהן שום קשר להשתתף באותו חוזה, אם רק המבנה שלהן תואם. הכלים שממממשים זאת הם Protocols.

1.2 Protocol: החוזה בלי הירושה

Protocol מגדיר interface בלי לממש אותו. כל מחלקה שהאיברים שלה תואמים הופכת אוטומטית לsubtype שלו, בלי extends, בלי ABC, בלי רישום:

Protocol בסיסי

```python
from typing import Protocol

class Greeter(Protocol):
    def greet(self, name: str) -> str: ...

def welcome(g: Greeter) -> str:
    return g.greet("world")

class English:  # not inheriting from Greeter at all
    def greet(self, name: str) -> str:
        return f"hello {name}"

class Spanish:
    def greet(self, name: str) -> str:
        return f"hola {name}"

print(welcome(English()))   # hello world
print(welcome(Spanish()))   # hola world
```

הנקודה החשובה ביותר: גם English וגם Spanish אינם יורשים מ־Greeter, ובכל זאת שניהם נתפסים כ־Greeter על ידי כלי הבדיקה (כמו mypy). ההתחייבות היא מבנית: מה שחשוב הוא ש־greet קיים עם חתימה תואמת.

הערה 1.1. כברירת מחדל, Protocol הוא עניין של type checking בלבד: `isinstance(x, Greeter)` ייכשל בזמן ריצה. אם אתם באמת צריכים בדיקת isinstance בזמן ריצה (למשל בממשק עם קוד חיצוני), קשטו את ה־Protocol ב־`@runtime_checkable`. שימו לב לתמורה: הבדיקה בזמן ריצה בודקת **קיום** של איברים, לא את החתימה שלהם, אז היא חלשה יותר מה־type checker.

הערה 1.2. האליפסות (`...`) מסמנות "זה החוזה בלבד": לא פגמתי בהקלדה, זה הנוהל. זהו הגוף התקין של מתודה; גם `pass` יעבוד, אך `...` מקובל יותר בספרות.

> **כלל אצבע**
>
> כשצריך לחלוק interface בין כמה מחלקות (והמחלקות אינן חולקות התנהגות ממשית), העדיפו Protocol על פני ABC. ABC שווה את זה כשצריך ברירת מחדל או ירושה של התנהגות. בפרקטיקה: כל ABC שנכתב רק כדי "להיות מסודר" הופך ל־Protocol ברגע שמתחילים לבדוק אותו.

1.3 TypedDict: מילונים מובנים

פרויקטים אמיתיים עמוסים במילונים שמתנהגים כמו records: ה־`dict` הגדול שיודע הכול, חוץ ממה שהוא בעצם. TypedDict מאפשר לתאר את המפתחות והערכים שלהם בלי להפוך אותם למחלקות:

TypedDict עם מפתחות אופציונליים

```python
from typing import TypedDict, NotRequired

class User(TypedDict):
    id: int
    name: str
    email: NotRequired[str]

def describe(u: User) -> str:
    email = u.get("email", "no email")
    return f"{u['id']}: {u['name']} ({email})"

u1: User = {"id": 1, "name": "Dana"}
u2: User = {"id": 2, "name": "Omri", "email": "o@x.io"}
```

בהגדרות TypedDict מומלץ להוסיף `total=False` רק כשהכול אופציונלי. לכתיבה סלקטיבית עדיף NotRequired על מפתח ספציפי.

הערה 1.3. TypedDict הוא כלי לזמן type checking בלבד. בזמן ריצה הוא עדיין `dict` לכל דבר, ומומלץ לשלב אותו עם אימות בזמן ריצה (למשל pydantic) כשהנתונים מגיעים מבחוץ. `json.loads` בפרוד מחזיר כל מבנה שהוא; ה־type checker לא מגן עליך בזמן ריצה.

1.4 TypeVar ו־Generic: פולימורפיזם אמיתי

מערכות טיפוס חזקות בונות על משתני טיפוס. TypeVar אומר למערכת "הטיפוס הזה ישתמר בכל פעולה"; ו־Generic (מחלקה פרמטרית) מאפשר למחלקה להיות פרמטרית. זה אולי נראה כמו סיבוך מיותר — עד שנתקלים ב־`list[Any]` בפרוד ומבינים כמה מידע בדיוק ויתרנו עליו.

מחסנית פרמטרית

```python
from typing import Generic, TypeVar

T = TypeVar("T")

class Stack(Generic[T]):
    def __init__(self) -> None:
        self._items: list[T] = []

    def push(self, item: T) -> None:
        self._items.append(item)

    def pop(self) -> T:
        return self._items.pop()

numbers = Stack[int]()
numbers.push(10)
n: int = numbers.pop()   # mypy: OK
```

שימו לב למוסכמה: משתנה טיפוס חופשי (שניתן להחליפו בכל טיפוס) נכתב באות רישית אחת, כמו T. משתנה עם אילוצים (bound או constraints) או עם variance מקבל שם תיאורי ארוך יותר, כמו T_co. כברירת מחדל TypeVar הוא invariant. והכלל: TypeVar("T") צריך להתאים לשם המשתנה, כדי שכלי הבדיקה יזהו אותו נכון.

הגדרה 1.2 (ParamSpec). ParamSpec לוכד את החתימה המלאה של פונקציה, ומאפשר לשמר אותה בעיטוף (wrapping). חיוני ל־decorators שומרי טיפוסים.

decorator ששומר על החתימה

```python
from typing import Callable, ParamSpec, TypeVar

P = ParamSpec("P")
R = TypeVar("R")

def logged(fn: Callable[P, R]) -> Callable[P, R]:
    def inner(*args: P.args, **kw: P.kwargs) -> R:
        print(f"calling {fn.__name__}")
        return fn(*args, **kw)
    return inner

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

result = add(2, 3)   # int, not Any
```

ללא ParamSpec, כלי בדיקה רואים את add כפונקציה שמקבלת *args ו־**kwargs, ומאבדים כל ידע על החתימה. עם ParamSpec, ההעברה נשמרת בצורה מלאה.

1.5 Annotated, TypeAlias ו־cast

שלושת אלה הם ה־utility הקטנים והשימושיים של מערכת הטיפוסים: Annotated מוסיף תיעוד, TypeAlias נותן שם קצר לנוסחאות ארוכות, ו־cast מאפשר לומר לכלי הבדיקה "סמכו עליי הפעם"; משפט שמוביל לבאגים כשמשתמשים בו בקלות דעת.

מטא־נתונים על טיפוס

```python
from typing import Annotated, TypeAlias, cast

UserId = Annotated[int, "must be positive"]
JsonObject: TypeAlias = dict[str, object]

def load() -> object:
    return {"ok": True}

raw = load()
# mypy cannot prove this is a dict
data = cast(dict[str, object], raw)
```

- Annotated מוסיף לטיפוס מטא־נתונים (תיעוד, או מידע שכלים לאימות בזמן ריצה כמו pydantic קוראים), בלי לשנות את הטיפוס.
- TypeAlias נותן שם ברור לטיפוס מורכב, ומונע חזרה על הגדרות ארוכות.
- cast הוא מנגנון "אני יודע יותר טוב מכלי הבדיקה". הוא לגיטימי בממשקי גבול (כמו גיוס נתונים לא מובנים). אבל אם אתם מוצאים את עצמכם מרבים ב־cast, כנראה חסרה אפיון נכון של הנתונים.

1.6 בעולם האמיתי: Repositories, Services ו־Plugins

כאשר לוגיקת ה־domain תלויה רק ב־Protocol, אפשר להחליף את המימוש בלי לגעת בלוגיקה, ובלי frameworks כבדים. מעבר שלם ל־database אחר מצטמצם לשינוי של שורה אחת. הדוגמה הבאה היא תבנית Repository טיפוסית:

מאגר עם Protocol

```python
from typing import Protocol

class UserRepository(Protocol):
    def find(self, user_id: int) -> User | None: ...
    def save(self, user: User) -> None: ...

class PostgresUserRepository:
    def find(self, user_id: int) -> User | None:
        return query_postgres("SELECT ...", user_id)

    def save(self, user: User) -> None:
        execute_postgres("INSERT ...", user)

class FakeUserRepository:
    def __init__(self) -> None:
        self.users = {1: User(id=1, name="Dana")}

    def find(self, user_id: int) -> User | None:
        return self.users.get(user_id)

    def save(self, user: User) -> None:
        self.users[user["id"]] = user

class UserService:
    def __init__(self, repo: UserRepository) -> None:
        self.repo = repo

    def rename(self, user_id: int, name: str) -> User:
        user = self.repo.find(user_id)
        if user is None:
            raise ValueError(f"no user {user_id}")
        user["name"] = name
        self.repo.save(user)
        return user
```

FakeUserRepository לבדיקות, PostgresUserRepository בפרוד, ו־UserService לא יודע מי מביניהם עומד מאחורי ה־Protocol. (ה־User כאן הוא ה־TypedDict מהסעיף הקודם, וה־execute_postgres/query_postgres הם נציגים של דרייבר אמיתי.) זה הבסיס ל־Injection Dependency (הזרקת תלויות) שייראה בפרק 6 — מונח שנשמע מפחיד עד שמגלים שמדובר בסתם "העבירו לי את הרכיב מבחוץ".

Plugin עם נקודת הרחבה

```
from typing import Protocol

class OutputPlugin(Protocol):
    name: str
    def emit(self, line: str) -> None: ...

class Printer:
    name = "printer"
    def emit(self, line: str) -> None:
        print(line)

class FileSink:
    name = "file"
    def emit(self, line: str) -> None:
        with open("out.log", "a") as f:
            f.write(line + "\n")

def run(plugins: list[OutputPlugin]) -> None:
    for p in plugins:
        p.emit("hello")
```

1.7 Anti-pattern → הדרך הטובה יותר

כל מפתח מתחיל עם הקוד הזה: ה־Any, ה־dict חסר הצורה, הבליעה. ההבדל הוא לזהות את הדוגמאות האלה לפני שהן מגיעות לפרוד, לא אחרי. קחו את ההיכרות הזו:

לא טוב: Any ובליעה של טיפוסים

```
# what NOT to do
def process(raw: dict) -> dict:
    user_id = raw["id"]          # unknown type
    return {"result": user_id}   # dict loses shape
```

טוב: אפיון מדויק

```
from typing import TypedDict, NotRequired

class Payload(TypedDict):
    id: int
    name: str
    note: NotRequired[str]

def process(raw: Payload) -> dict[str, int | str]:
    return {"result": raw["id"], "name": raw["name"]}
```

הכלל המנחה: ככל שממעטים ב־Any ומשאירים את הסקת הטיפוסים (type inference) לכלי הבדיקה, כך הם עושים עבודה אמיתית. Any הוא בעצם "כבו את כלי הבדיקה כאן". כשמכבים את כל מערכות האזהרה, בהתחלה זה שקט; עד שמשהו נשרף. קוד מושלם לא קיים בפרוד.

1.8 תרגיל קצר

הגיע הזמן להפעיל את הידיים. אל תמהרו לדלג; חלק גדול מהלמידה כאן קורה דווקא כשנתקעים. הגדירו Protocol בשם ReportRenderer עם המתודה הבאה:

ReportRenderer

```python
from typing import Protocol

class ReportRenderer(Protocol):
    def render(
        self,
        header: str,
        rows: list[tuple[str, int]],
    ) -> str: ...
```

תרגיל 1: Protocol של Renderer

1. מימשו את CsvRenderer שמחזיר CSV, ואת PlainRenderer שמחזיר טקסט מיושר.
2. כתבו פונקציה `print_report(r: ReportRenderer)` שמדפיסה את הפלט, והשתמשו בה עם שני המימושים.
3. (רמה שניה) הוסיפו TypedDict בשם SalesRow עם `product: str` ו־`units: int`, והחליפו את הטיפוס של השורות בהתאם.

פרק 2: טיפול תקין בשגיאות ולוגינג מובנה

קוד פרוד נמדד לא רק במה שהוא עושה כשהכול עובד, אלא במה שהוא עושה כשמשהו נשבר. בפרוד תקלות הן בלתי נמנעות. בפרק זה נבנה היררכיית חריגות נכונה, נלמד על Excep-tion Groups ב־3.11+, ונמיר לוגינג טקסטואלי ללוגינג מובנה (Logging Structured (לוגינג מובנה)) שאפשר לשאול אליו שאלות בפרוד, כולל ID Correlation (מזהה קורלציה)s שעוקבים אחרי בקשה אחת בכל המערכת. היעד: כשמשהו יתפוצץ, תדעו איפה לחפש.

2.1 היררכיית חריגות נכונה

הטעות הנפוצה ביותר היא לזרוק חריגות מובנות (`ValueError`, `RuntimeError`) בלי להבדיל ביניהן. שימוש חוזר ב־`except Exception` ללא הבחנה מוביל להסתרת באגים. כשמשתמשי החבילה רוצים לתפוס "את השגיאות שלכם": אין להם במה לתפוס. הפתרון: בסיס ייחודי לכל domain:

הירכיית חריגות ייעודית

```python
class PaymentError(Exception):
    """Base error for the payment domain."""

class InsufficientFundsError(PaymentError):
    pass

class GatewayUnavailableError(PaymentError):
    pass

def charge(card, amount: int) -> None:
    if amount > card.balance:
        raise InsufficientFundsError(
            f"balance {card.balance} < amount {amount}"
        )
    if not gateway_online():
        raise GatewayUnavailableError("gateway is down")
    gateway.charge(card, amount)
```

עכשיו משתמשים מבחוץ יכולים לתפוס במדויק את מה שהם צריכים:

תפיסה לפי רמת החומרה

```python
try:
    charge(card, amount)
except InsufficientFundsError as exc:
    notify_user(card, str(exc))  # user-visible, retryable
except GatewayUnavailableError as exc:
    retry_later(exc)  # infrastructure, back off
except PaymentError as exc:
    report_to_sentry(exc)  # unexpected domain failure
```

הערה 2.1. בסיס החריגות יורש מ־`Exception` כמו כל חריגה רגילה. אל תתפסו אותו ב־`except Exception`, אלא במפורש עם `except PaymentError`: עם `except Exception`

2

תתפסו גם באגים בלתי צפויים ותסתירו אותם, והסתרת באג בשקט היא טעות חמורה.

2.2 Exception Groups וחריגות מרובות (3.11+)

עיבוד קבוצות (batching) נכשל בדרך כלל בכמה מקומות בבת אחת. כשזה נשבר, זה נשבר בגדול. במקום לזרוק רק את החריגה הראשונה (ולאבד מידע על השאר), Python 3.11 מציג ExceptionGroup (קבוצת חריגות):

איסוף כל הכישלונות

```
def charge_batch(charges: list[tuple["Card", int]]) -> None:
    errors: list[Exception] = []
    for card, amount in charges:
        try:
            charge(card, amount)
        except PaymentError as exc:
            errors.append(exc)
    if errors:
        raise ExceptionGroup("batch charge failed", errors)
```

לתפוס את החריגות שבקבוצה, משתמשים ב־except*:

טיפול מפוצל עם except*

```
try:
    charge_batch(charges)
except* InsufficientFundsError as eg:
    notify_users(eg.exceptions)   # every failed charge here
except* GatewayUnavailableError as eg:
    raise ServiceError(
        f"gateway failed for {len(eg.exceptions)} charges"
    ) from eg
```

מתי להשתמש ב־ExceptionGroup

אל תהפכו כל שגיאה לקבוצה. ExceptionGroup שווה את המחיר רק כשחשוב לדווח על **כל** הכישלונות בבת אחת, למשל ב־batch jobs, אימות מרובה של שדות, או compilation errors. לפעולה בודדת, חריגה אחת עדיפה.

2.3 לוגינג מובנה: JSON + logging ו־structlog

לוג שמדפיס מחרוזת הוא חצי עבודה: קשה לסנן, למיין ולשאול אותו; הוא קצת כמו עד ראייה, זוכר בערך ומערבב פרטים. לוג structured logging הוא רצף של אובייקטי JSON. ואז כל כלי (ELK, CloudWatch, Loki) יודע לשאול עליו שאלות.

ניתן להתחיל מהספרייה הסטנדרטית בלבד:

2

מעצב JSON עם logging

```python
import json
import logging

class JsonFormatter(logging.Formatter):
    def format(self, record):
        data = {
            "level": record.levelname,
            "logger": record.name,
            "message": record.getMessage(),
        }
        for key, value in record.__dict__.items():
            if key.startswith("ctx_"):
                data[key[4:]] = value
        return json.dumps(data, default=str)

logger = logging.getLogger("app")
handler = logging.StreamHandler()
handler.setFormatter(JsonFormatter())
logger.addHandler(handler)
logger.setLevel(logging.INFO)
logger.info("order created", extra={"ctx_order_id": "ORD-42"})
```

```
{"level": "INFO", "logger": "app", "message": "order created",
 "order_id": "ORD-42"}
```

בפרויקטים גדולים structlog מקצר את הדרך ומפשט חיבור הקשר:

אותה הודעה עם structlog

```
import structlog

log = structlog.get_logger()

log.info("order_created", order_id="ORD-42", user_id=7)
```

הערה 2.2. כל איבר ב־event dictionary של structlog הופך לשדה ב־JSON. הקפידו על מוסכמות שמות עקביות (snake_case לשדות), כדי שהשאילתות שלכם יהיו אחידות: אי־עקביות במוסכמות שמות גורמת לכישלון שאילתות ולבעיות בניתוח הלוגים.

2.4 שמירה על הקשר: Correlation IDs

במערכת של microservices או משימות עיבוד, בקשה אחת עוברת דרך רכיבים רבים. וכשהיא נכשלת, כולם מסתכלים אחד על השני. correlation IDs אחד מאפשר לקשר בין כל ההודעות של אותה בקשה, ולסיים את משחק ה"לא אני". contextvars הוא הכלי הנכון להעביר את ה־ID בלי להעביר אותו כארגומנט בכל פונקציה:

2

הזרקת ההקשר עם contextvars

```
import structlog
from contextvars import ContextVar

request_id: ContextVar[str] = ContextVar(
    "request_id", default="-",
)

def add_request_id(logger, method_name, event_dict):
    event_dict.setdefault("request_id", request_id.get())
    return event_dict

structlog.configure(
    processors=[
        add_request_id,
        structlog.processors.JSONRenderer(),
    ]
)

def handle(req) -> None:
    token = request_id.set(req.request_id)
    try:
        process(req)
    finally:
        request_id.reset(token)

def process(req) -> None:
    log = structlog.get_logger()
    log.info("processing", path=req.path)   # gets request_id
```

למה contextvars?

שלא כמו גלובלים, contextvars בטוחים במקביליות: לכל task (async) ולכל thread יש עותק משל עצמם; וכל אחד יכול להגדיר request id משלו בלי להפריע לאחרים. במילים פשוטות: כל בקשה נוסעת עם התיק שלה; והמטען של השכן נשאר של השכן.

2.5 מתי לזרוק, מתי לבלוע, מתי לעטוף

שלוש שאלות מנחות בכל except. אם תענו עליהן לפני הדיפלוי, תחסכו לעצמכם שיחת טלפון באמצע הלילה:

- **לבלוע**: רק אם השגיאה צפויה, ידועה, ואפשר להמשיך עם ערך חלופי. תמיד לתת סיבה debug.
- **לעטוף**: כשמחצינים שגיאה ברמת חוצץ (כמו `DBError` → `ServiceError`), עם `from exc` לשמירת הסיבה.
- **להעביר הלאה** (`raise`): כשהשגיאה כבר ברמת הייצוג המתאימה, או שאין דרך לטפל בה כאן.

2

השלושה בפעולה

```python
import logging

# swallow: known and expected, safe default
logger = logging.getLogger("metrics")  # ctx_ + JsonFormatter
try:
    metric = fetch_metric(name)
except MetricNotFoundError:
    logger.debug("metric missing, using default",
                 extra={"ctx_metric": name})
    metric = 0.0

# wrap: raise at the right boundary
try:
    records = db.query("SELECT ...")
except DBError as exc:
    raise ServiceError("database unavailable") from exc

# re-raise: this layer cannot handle it
try:
    validate(payload)
except ValidationError:
    raise   # caller decides what to do
```

2.6 Anti-pattern → הדרך הטובה יותר

את הגרסאות "הלא טובות" האלה נכתבות לעיתים קרובות בלחץ של deadline. העיקר הוא ללמוד לזהות אותן לפני שהן מגיעות לפרוד:

לא טוב: בליעה גורפת

```
# what NOT to do
try:
    do_work()
except Exception:            # catches everything
    pass                     # and hides it
```

לא טוב: מילוי לוג בלי מבנה

```
# what NOT to do
print(f"ERROR charging card {card} amount {amount}")
```

טוב: מדויק, מתועד ומבני

```
import logging

logger = logging.getLogger("payments")  # ctx_ + JsonFormatter

try:
    charge(card, amount)
except InsufficientFundsError:
    logger.info("charge declined",
                extra={"ctx_card_id": card.id,
                       "ctx_reason": "insufficient_funds"})
    raise
except GatewayUnavailableError as exc:
    logger.error("gateway failure",
                 extra={"ctx_error": str(exc),
                        "ctx_retryable": True})
    raise RetryableError("gateway unavailable") from exc
```

כללי האצבע

- לעולם לא `except Exception: pass` בלי לוג.
- כל `except` בודק את עצמו בשאלה: למה מותר לי לתפוס את זה כאן?
- לוג מובנה עם IDs correlation שווה יותר מתריסר שורות `print`.

שלושת הכללים האלה מצמצמים משמעותית קריאות לילה מיותרות.

2.7 תרגיל קצר

תרגיל נוסף לחיזוק ההבנה. המטרה: לסיים אותו עם ExceptionGroup אחד ועם פחות חשש מתקלות ליליות בשליחת מיילים.

תרגיל 2: שליחת מיילים עמידה

1. הגדירו EmailError ובניה: SendFailedError ו־RateLimitedError.
2. כתבו פונקציה `send_all(emails)` שאוספת את כל הכישלונות ומרימה ExceptionGroup אם היו כאלה.
3. השתמשו ב־`except*` כדי לטפל ב־RateLimitedError בניסיון חוזר, וב־SendFailedError בדיווח.
4. (רמה שניה) הוסיפו לוג מובנה עם IDs correlation לכל הבקשה, כך שכל הודעה תכלול את ה־ID של הקריאה ל־`send_all`.

פרק 3: ניהול משאבים ו־Context Managers

חיבורים, קבצים, נעילות ומשאבים זמניים הם מקור דליפות ושגיאות. בפרק זה נלמד לכתוב Manager Context (מנהל הקשר)s מתקדמים (סינכרוניים ואסינכרוניים), להשתמש ב־ExitStack (ערימת יציאה) לחיבור משאבים דינמי, ולהפנים את עקרון ה־RAII כפי שהוא מתבטא בפייתון. ניהול משאבים נכון מונע דליפות ושגיאות קשות לאיתור.

3.1 הבסיס: `__enter__` ו־`__exit__`

הגדרה 3.1 (Context Manager)**.** אובייקט שמבטיח סדר קפדני: פעולות ההכנה מתרחשות ב־`__enter__`, ופעולות הניקוי מתרחשות ב־`__exit__` **בכל מקרה**, כולל בזמן חריגה. זהו ה־try/finally המופשט והבטוח.

Context manager למחלקה

```
class Database:
    def __enter__(self) -> Connection:
        self._conn = connect(url)
        return self._conn

    def __exit__(self, exc_type, exc, tb) -> None:
        self._conn.close()

with Database() as conn:
    conn.query("SELECT 1")
# closed even if query raised
```

שלושת הארגומנטים של `__exit__` הם סוג החריגה, החריגה עצמה וה־traceback; ואפשר להשתמש בהם כדי להחליט אם לבלוע או להפיץ. החזרת `True` מהמתודה מדכאת חריגה בתוך הבלוק, כוח שכדאי להפעיל במשורה; דיכוי חריגות לצורך נוחות מוביל לבאגים שקטים.

3.2 contextlib: מהר יותר ונקי יותר

רוב ה־context managers קצרים מספיק כדי לכתוב אותם כgenerator עם contextmanager (כולל type hints), וכשזה עובד, זה מרגיש כמו קסם. היבט אחד חשוב להפנים: הכול שמתחת ל־`yield` הוא ה־`finally`.

3

contextmanager בפועל

```python
import os
from contextlib import contextmanager

@contextmanager
def temporary_file(suffix: str = ".tmp"):
    path = make_temp_path(suffix)
    try:
        yield path
    finally:
        os.unlink(path)

with temporary_file() as path:
    write_report(path)      # path always cleaned up
```

הערה 3.1. מלכודת קלאסית: בתוך contextmanager, `return value` בתוך ה־`try` (לפני ה־`yield`) יזרוק `RuntimeError` — אסור להחזיר ערך מפונקציה כזו. אם אתם צריכים להעביר ערך החוצה, החזירו אותו דרך פרמטר mutable, או כתבו context manager כיתתי במקום.

ולמשימות ניקוי best-effort שבהן היעדר משאב הוא מצב תקין, suppress:

suppress: ניקוי רגוע

```python
import os
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove("report.tmp")   # no crash if missing
```

הערה 3.2. suppress מתאים לניקוי שחסרונו תקין — ה־"בסדר, לא היה כאן כלום" של פייתון. אל תשתמשו בו כדי לבלוע שגיאות עסקיות; לזה משמשת היררכיית החריגות מפרק 2.

3.3 Async context managers

כשמנהלים משאבים אסינכרוניים (connections ל־databases, HTTP clients), הניקוי הוא `await`; לא תוכלו לסגור אותם כמו קובץ רגיל; והאזהרות על unclosed resources מחייבות טיפול נכון. צריך `__aenter__`/`__aexit__`, או asynccontextmanager:

context manager Async

```
from contextlib import asynccontextmanager

@asynccontextmanager
async def db_connection(url: str):
    conn = await aconnect(url)
    try:
        yield conn
    finally:
        await aclose(conn)

async def main() -> None:
    async with db_connection("postgres://...") as conn:
        rows = await conn.fetch("SELECT 1")
```

סימטריה

אל תשתמשו ב־`with` עם context manager סינכרוני או להפך. `async with` חוסר הסימטריה מוביל ל־unclosed resources שקטים: דליפה שהכי קשה לאתר, מסוג הדליפות שמתגלות רק לאחר חקירה ממושכת.

3.4 ExitStack: חיבור משאבים דינמי

לפעמים מספר המשאבים ידוע רק בזמן ריצה, למשל שרשור plugins או middlewares. כאן with בודד כבר לא מספיק; ובא לידי ביטוי ה־ExitStack; הוא אוסף משאבים ומוודא שכל אחד מהם נסגר (בסדר הפוך) בסוף הבלוק, גם במקרה של יציאה מוקדמת עקב חריגה.

ExitStack לחיבור משאבים

```python
from contextlib import ExitStack

def setup_pipeline(stack: ExitStack, config) -> Pipeline:
    source = stack.enter_context(open_source(config))
    sink = stack.enter_context(open_sink(config))
    lock = stack.enter_context(acquire_lock("pipeline"))
    return Pipeline(source, sink, lock)

def main() -> None:
    with ExitStack() as stack:
        pipeline = setup_pipeline(stack, config)
        pipeline.run()   # all three closed afterwards
```

אותו מנגנון מאפשר rollback: אם ההתקנה נכשלת בשלב שלישי, כל מה שנפתח קודם נסגר אוטומטית:

אל: ExitStack כגיבוי

```python
def install_migrations(stack: ExitStack, db) -> list[str]:
    done: list[str] = []
    for migration in list_migrations():
        stmt = db.prepare(migration)
        done.append(migration)
        stack.callback(print, f"rollback: {migration}")
    return done
```

הערה 3.3. `stack.callback(...)` רושם פעולה שתתבצע בסוף, עוד כלי נהדר לניקוי המותנה. כל מה שנרשם מתבצע בסדר LIFO, גם אם נזרקה חריגה בדרך. מנגנון זה מבטיח ניקוי מסודר ללא תלות בנתיב הביצוע.

3.5 RAII בפייתון

ב־C++ עקרון ה־RAII קובע שהמשאב נרכש בבנאי ומשוחרר בדסטרקטור, רעיון אלגנטי, כל עוד לא שואלים "מתי בדיוק". בפייתון, דסטרקטורים (`__del__`) הם בלתי צפויים: אין אחריות מתי הם רצים; ואין ערובה שזה יקרה בזמן שנוח לכם. לכן ה־idiom התקין של RAII כאן הוא context manager:

RAII בסגנון פייתון

```
# what NOT to do
f = open("data.csv")
rows = parse(f)
# never closed -> fd leak
```

העיקרון

רכישה בכניסה, **שחרור** ביציאה, בלי תלות במקום בו נזרקה החריגה. אם הקוד שלכם מתחיל ב־`try:` ומסתיים ב־`finally:` סביב משאב, הפכו אותו ל־context manager, כך המשאב "מנקה את עצמו"; הקוד נעשה ברור ובטוח יותר.

3.6 Anti-pattern → הדרך הטובה יותר

הדוגמאות הבאות מדגימות שגיאות נפוצות בניהול משאבים ואת הפתרון התקין באמצעות context managers.

לא טוב: try/finally ידני בכל מקום

```
# what NOT to do
conn = connect(url)
try:
    conn.query("SELECT 1")
finally:
    conn.close()

lock = Lock()
lock.acquire()
try:
    work()
finally:
    lock.release()
```

לא טוב: שכחת שחרור

```
# what NOT to do
f = open("data.csv")
rows = parse(f)
# never closed -> fd leak
```

טוב: הבלוק עושה את העבודה

```
with connect(url) as conn:
    conn.query("SELECT 1")

with lock:
    work()

with open("data.csv") as f:
    rows = parse(f)
```

3.7 תרגיל קצר

תרגיל זה מיישם את כלי הפרק. בסיומו תוכלו לתזמן כל פונקציה ולנהל משאבים באופן בטוח. בניית תשתית נכונה היא הבסיס לקוד יציב.

תרגיל 3: context manager של מדדי ביצוע

1. כתבו context manager בשם timer (בעזרת contextmanager) שמתזמן את הבלוק ומדפיס/מחזיר את משך הזמן באלפיות השנייה.
2. כתבו גרסה async בשם atimer שמתאימה לקוד async.
3. כתבו context manager שמנהל חיבור אמיתי (`close/connect`), ובדקו שהוא נסגר גם כשזורקים חריגה בתוך הבלוק.
4. (רמה שניה) השתמשו ב־ExitStack כדי לפתוח שני קבצים בבת אחת ולמזג אותם לשורה אחת, תוך הבטחת סגירת שניהם.

פרק 4: תכנות מקבילי נכון

תכנות מקבילי בפייתון מתחיל בשאלה הלא נכונה: "איזו שיטה הכי מהירה?". ומסתיים בשאלה הנכונה: "כמה זמן בילה הקוד בהמתנה לפעולות I/O?". הבעיה לעולם אינה מהירות, אלא עצירה לא נכונה. בפרק זה נבנה TaskGroup (קבוצת משימות)s, ננהל futures נכון, נמנע מ־race conditions, ונטפל בביטול וב־timeouts.

4.1 מתי asyncio, מתי threads, מתי processes

הבחירה נקבעת לפי סוג העבודה, לא לפי הרגל. קל ליפול על המודל שבו השתמשו בפרויקט הקודם. קבלו את הטבלה הזו כחוק:

מודל	מתאים ל־	שיקולים
asyncio (ספריית אסינכרוניות)	הרבה I/O במקביל, קוד async	קואופרטיבי, רץ ב־thread אחד
Threads	I/O חוסם (urllib, sqlite)	קל לשילוב, סיכון races
Processes	עבודה CPU-bound (CPU כבד)	GIL לא מגביל, עלות IPC

כלל האצבע

עבודה אינטנסיבית ב־CPU → processes. I/O חוסם שקשה להסב ל־async → threads. I/O שניתן להפוך ל־async → asyncio. וזכרו: לפעמים הדרך המהירה ביותר היא לעשות **פחות** (ראו אופטימיזציה בפרק 5). וגם: אל תבחרו כלי כי "אוהבים אותו"; תבחרו כי הוא מתאים. thread pools בפרוד שנולדו מרגש, לא מהנדסה, הם מתכון לבעיות.

4.2 asyncio מתקדם: TaskGroup ו־Structured Concurrency

ניהול tasks ידני מרגיש כמו לוליין בלי רשת; כל ניהול כזה הוא סיכון מחושב. TaskGroup (מ־Python 3.11) מחליף את ניהול ה־task הידני ומבטיח התנהגות מובנית: אם **כל** פעולה נכשלת, כל השאר מבוטלים, וה־ExceptionGroup מועבר הלאה:

TaskGroup: שליפה מקבילה

```
import asyncio

async def fetch(url: str) -> str:
    return await http_get(url)

async def main() -> None:
    urls = ["https://a.io/1", "https://b.io/2"]
    async with asyncio.TaskGroup() as tg:
        tasks = [tg.create_task(fetch(u)) for u in urls]
    results = [t.result() for t in tasks]
    print(results)

asyncio.run(main())
```

מה מקבלים חינם? ביטול ה־tasks כשאחד נכשל, איסוף של הכישלונות ל־ExceptionGroup, והבטחה שהבלוק ממתין לסיום כולם. זהו structured concurrency: ה־scope שולט במחזור החיים של ה־tasks.

הערה 4.1. גרסת ה־async context manager של TaskGroup. היא ה־scope של אם אין לכם TaskGroup (לפני 3.11), השתמשו ב־gather — אבל עם return_exceptions=True יסתיר כישלונות בודדים, ו־asyncio.wait_for יעצור משימה תלויה. ואם פייתון שלכם עדיין בלי TaskGroup, שדרוג משתלם יותר מכל workaround.

4.3 concurrent.futures בצורה נכונה

פעם, ב־3 לפנות בוקר, גיליתי thread ששכחו לסגור (לא אגיד כמה זמן הוא נותר חי, אבל הוא לא לבד). כשמשאירים את ה־ThreadPoolExecutor (מאגר threads) מחוץ ל־with, ה־threads נסגרים רק ב־exit. לכן תמיד להשתמש בו כ־context manager. וכשאספו תוצאות, תמיד עם timeout:

נכון ThreadPoolExecutor

```
from concurrent.futures import ThreadPoolExecutor
import urllib.request

def fetch(url: str) -> bytes:
    with urllib.request.urlopen(url) as r:
        return r.read()[:64]

def main() -> None:
    urls = [
        "https://www.python.org/",
        "https://www.python.org/about/",
    ]
    with ThreadPoolExecutor(max_workers=8) as pool:
        futures = [pool.submit(fetch, u) for u in urls]
        for fut in futures:
            data = fut.result(timeout=5)
            print(len(data))

main()
```

מה לגבי ProcessPoolExecutor?

אותו דפוס בדיוק, עם `from concurrent.futures import ProcessPoolExecutor`. תנאי מקדים: טענת הפונקציה וההחזר חייבות להיות picklable; ויש להעביר את העבודה מהשורה התחתונה (מודול רץ בלבד). אם העבודה היא CPU-heavy, לרוב עדיף ProcessPoolExecutor (מאגר תהליכים) עם chunks, לא threads שמתחככים ב־GIL. ואל תלמדו את זה בדרך הקשה: עבודה CPU-heavy על threads היא מתכון מוכח ללילה בלי שינה.

4.4 Race conditions, מצב משותף ו־immutability

כל מי שאומר שהקוד שלו נקי מ־race conditions משקר. או שהוא עדיין לא דיפלוי בפרוד. הדרך הבטוחה ביותר לחסל race condition היא לא לשתף מצב הניתן לשינוי. אם חייבים, נועלים:

נעילה נכונה

```python
from threading import Lock

class Counter:
    def __init__(self) -> None:
        self._n = 0
        self._lock = Lock()

    def bump(self) -> None:
        with self._lock:
            self._n += 1

    def value(self) -> int:
        with self._lock:
            return self._n
```

אבל העדיפות הראשונה היא מצב immutable: העבירו נתונים frozen, החזירו נתונים

חדשים, ואל תשנו אובייקטים משותפים במקום. כך ה־race נעלם לפני שהוא נוצר:

ערך immutable חולק

```python
from dataclasses import dataclass

@dataclass(frozen=True)
class Config:
    workers: int
    timeout: float

SHARED = Config(workers=8, timeout=5.0)
# threads only read; nothing can be mutated
```

4.5 ביטול (cancellation) ו־timeouts

שירות שמחזיר תשובה אחרי חצי שעה הוא לא שירות; הוא מכתב שאבד בדואר. קוד production חייב להחליף תשובה איטית בתשובה מהירה. `asyncio.timeout` (+3.11) נותן scope לביטול:

timeout עם גיבוי

```python
import asyncio

async def main() -> None:
    try:
        async with asyncio.timeout(2.0):
            data = await slow_call()
    except TimeoutError:
        data = fallback()
    return data
```

ולפעמים רוצים **לא** לבטל, למשל הודעה אחרונה שמבצעים לפני הפסקה. shield מגן על משימה מפני ביטול חיצוני:

shield: משימה מוגנת

```python
async def shutdown() -> None:
    try:
        await asyncio.shield(flush_and_close())
    except asyncio.CancelledError:
        print("shutdown cancelled")   # flush continues
```

הערה 4.2. cancellation הוא סוג שגיאה לגיטימי; ולכן except על `CancelledError` צריך להעביר אותו הלאה, אלא אם אתם באמת מסיימים את המשימה. בליעה של ביטול משאירה tasks תלויים, כמו לכסות נורת אזהרה בנייר דבק: השקט כוזב; והמנוע מתחמם.

4.6 Anti-pattern → הדרך הטובה יותר

הגרסאות ה"לא טובות" האלה נכתבות לעיתים קרובות בלחץ של deadline. קבלו אותן, ותלמדו מהטעויות:

לא טוב: threads ידניים ללא timeout

```python
# what NOT to do
threads = []
for url in urls:
    t = threading.Thread(target=fetch, args=(url,))
    t.start()
    threads.append(t)
for t in threads:
    t.join()          # may hang forever; errors hidden
```

לא טוב: מצב משותף מתפרץ

```
# what NOT to do
results: list[str] = []

def worker(url: str) -> None:
    results.append(fetch(url))   # shared list, no order

for url in urls:
    threading.Thread(target=worker, args=(url,)).start()
```

טוב: futures + timeout + אין shared state

```
from concurrent.futures import ThreadPoolExecutor

with ThreadPoolExecutor(max_workers=8) as pool:
    futures = [pool.submit(fetch, u) for u in urls]
    for fut in futures:
        data = fut.result(timeout=5)   # no shared list
        handle(data)
```

4.7 תרגיל קצר

הגיע הזמן לגלגל שרוולים. קחו קפה, ולכו:

תרגיל 4: עיבוד URLs במקביל

1. כתבו פונקציה `fetch_title(url)` → `str` שמחזירה את הכותרת של עמוד (HTTP GET פשוט).
2. גרסה ראשונה: ThreadPoolExecutor עם `timeout=5` וטיפול חסין בשגיאות.
3. גרסה שניה: async עם TaskGroup ו־`asyncio.timeout`.
4. (רמה שניה) הוסיפו: ExceptionGroup אם שתי שליפות נכשלו בו־זמנית, דווחו על שתיהן בחריגה אחת, מבלי לרסק את התוצאות שהצליחו.

פרק 5: חשיבה ביצועית

הדרך הקלה ביותר לפגוע בביצועים היא לייעל לפני שמדדו. בפרק זה נלמד למדוד נכון עם cProfile, py-spy ו־scalene, לאתר צריכת זיכרון, לבחור caching נכון, ולהשתמש ב־`__slots__`, ב-generators ובחשיבה מונחית נתונים; כולל מתי **לא** לייעל.

5.1 למדוד לפני שמשנים: cProfile

רוב הניחושים על "איפה הקוד איטי" שגויים. מול שירות איטי, קל לחשוב שהבקבוק ב־SQL query, ולגלות שה־query עצמו היה מהיר; אבל קראנו לו פעם אחת על כל שורה במקום פעם אחת (N+1). cProfile הוא הפרופיילר המובנה; אין צורך בספרייה חיצונית:

```
$ python -m cProfile -s cumtime app.py
      ncalls  tottime  cumtime  filename:lineno
         300    0.020    1.210  normalize.py:23
        9000    0.110    0.140  strings.py:7
        1500    0.090    0.100  parse.py:41
```

העמודה cumtime היא הזמן הכולל כולל קריאות משנה; לכן היא מספרת איפה לטפל. אל תסתכלו על הקוד; תסתכלו על ה־cumtime: מסמנים את השורה הגבוהה ביותר ושואלים "מה אם פשוט לא נקרא לה בכלל?"

מדידה מתוך הקוד

```
import cProfile

def run():
    records = load_all()
    return [normalize(r) for r in records]

cProfile.run("run()", sort="cumtime")
```

5.2 פרופיל של שירות חי: py-spy ו־scalene

כשהקוד כבר רץ בפרוד, אפשר לצרף פרופיילר בלי לגעת בו: בלי תקופת תחזוקה, בלי ticket, ובלי לבקש רשות:

```
$ py-spy top --pid 1234          # live sampling view
$ py-spy record -o out.svg --pid 1234
$ scalene app.py                 # CPU + memory in one tool
```

py-spy מצלם את המחסניות של תהליך רץ ללא שינוי קוד; ו־scalene מציע חיווי משולב ל־CPU ולזיכרון עם התמקדות בחשדים. שתי השיטות עדיפות על cProfile לרכיבי פרוד גדולים.

מתי איזה?

- cProfile: פונקציות CPU-bound במפתח.
- py-spy: שירות רץ שאי־אפשר לעצור.
- scalene: כשגם זיכרון וגם CPU מעניינים באותו ריצה.

ובחירה לפי שמועות אינה בחירה. יש כלים שנראים נהדר ב־demo; אבל בפרוד אין demo.

5.3 פרופיל זיכרון: tracemalloc

דליפות זיכרון וצריכה מופרזת לא מופיעות ב־cProfile. הן מתגלות בפרוד, או בלילה של דיפלוי. tracemalloc עוקב אחרי הקצאות:

tracemalloc בסיסי

```
import tracemalloc

tracemalloc.start()
records = run()
snapshot = tracemalloc.take_snapshot()
top = snapshot.statistics("lineno")[:5]
for stat in top:
    print(stat)
```

```
/tmp/x.py:12: size=4.2 MiB, count=51000, average=86 B
/tmp/x.py:20: size=1.1 MiB, count=9000, average=128 B
```

אם הראיה גדלה בלי לרדת, יש דליפה. tracemalloc מראה **איפה** נוצרו האובייקטים — חצי פתרון; החצי השני: להפסיק לקוות שהיא תעלם מעצמה.

5.4 אסטרטגיות caching

חזרה על חישוב יקר מאותו קלט היא בזבוז טהור. החישוב חוזר על עצמו בדיוק כשהוא יקר — וכמעט תמיד אפשר לחסוך אותו. שלושת הרבדים:

cache ו־lru_cache

```python
from functools import cache, lru_cache

@cache                          # unbounded memoization
def tokenize(text: str) -> tuple[str, ...]:
    return tuple(text.split())

@lru_cache(maxsize=128)         # bounded, LRU eviction
def fib(n: int) -> int:
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

הערה 5.1. cache אינו מוגבל בגודל, לכן השתמשו בו רק כשקבוצת הקלטים קטנה וידועה. ל־lru_cache אפשר לקרוא גם cache_clear() כשבודקים. וזכרו: קאש של פונקציה עם תופעות לוואי או עם ארגומנטים לא־hashable אינו תקין — קאש הוא מדיניות, לא מזומן שמתמלא מעצמו: מגדירים TTL, ביטול, ותשובה לשאלה "מה קורה כשהקאש נופל".

ומעבר לתהליך, cache חיצוני (למשל Redis) חולק תוצאות בין תהליכים ושרתים:

cache חיצוני עם TTL

```python
def get_user(user_id: int) -> User:
    key = f"user:{user_id}"
    cached = redis.get(key)
    if cached:
        return User.from_json(cached)
    user = load_from_db(user_id)
    redis.set(key, user.to_json(), ex=300)   # 5 min TTL
    return user
```

אזהרת cache

cache הוא **מדיניות**, לא תכונה: מגדירים TTL, ביטול (invalidation) ומה קורה כשהקאש נופל. קאש בלי מדיניות ביטול הוא מקור ל־stale data ולבאגים קשים לאבחון — נתונים מיושנים בפרוד נולדים בדיוק ככה.

5.5 ספריות cache מתקדמות: cachetools, diskcache, aiocache

הספרייה הסטנדרטית מספיקה למקרים פשוטים; בפרוד עם דרישות מורכבות יותר, כדאי להכיר את הכלים הבאים:

cache RR LFU, TTL, -- cachetools

```python
# pip install cachetools
from cachetools import cached, TTLCache, LFUCache, RRCache
from functools import lru_cache

# TTLCache: expires after TTL or when full
user_cache = TTLCache(maxsize=1000, ttl=300)  # 5 minutes

@cached(user_cache)
def get_user(user_id: int) -> User:
    return db.query(
        "SELECT * FROM users WHERE id = ?", user_id,
    )

# LFUCache: evicts the least used (Least Frequently Used)
freq_cache = LFUCache(maxsize=500)

# RRCache: random eviction -- fast, no access tracking
random_cache = RRCache(maxsize=500)
```

diskcache -- קאש עמיד לדיסק

```python
# pip install diskcache
from diskcache import Cache

# disk-backed cache -- survives a restart!
disk = Cache("./my_cache_dir")

@disk.memoize()
def expensive_computation(x: int) -> int:
    # runs once, the result is stored on disk
    return x ** 2

# clear expired
disk.expire()  # removes expired entries
disk.cull()    # shrinks to max size
```

aiocache -- קאש אסינכרוני ל-asyncio

```
# pip install aiocache[redis]
from aiocache import Cache, cached
from aiocache.serializers import JsonSerializer

cache = Cache(
    Cache.REDIS, endpoint="localhost", port=6379,
    serializer=JsonSerializer(),
)

@cached(ttl=60, cache=cache)
async def fetch_user_async(user_id: int) -> User:
    return await db.fetch_one(
        "SELECT * FROM users WHERE id = $1", user_id,
    )

# manual cleanup
await cache.delete("user:42")
await cache.clear()
```

בחירת כלי cache

Tool	Fits	Notes
`functools.lru_cache`	Pure functions, small/known input	In-memory, single process
`cachetools.TTLCache`	TTL + size limit, in-memory	Single process, thread-safe
`diskcache`	Persistence, share across runs	Disk-backed, slower
`aiocache`	asyncio, Redis/Memcached	Distributed, async

5.6 __slots__, generators וחשיבה מונחית נתונים

כשמיליוני אובייקטים קטנים, __slots__ חוסך זיכרון על ידי ביטול ה־__dict__. (וגם כאן: רק אחרי שמדדו — __slots__ הוא כלי, לא מדבקה על קוד.)

__slots__

```
class Point:
    __slots__ = ("x", "y")

    def __init__(self, x: int, y: int) -> None:
        self.x = x
        self.y = y
```

שלושת השיקולים: המחלקה היא leaf (בלי ירושה רחבה), שדות ידועים מראש, ואובייקטים רבים מאוד. אחרת, אל תגעו.

generators מעבירים נתונים בזרימה במקום לבנות רשימות שלמות:

generator במקום list

```
def active_ids(records):
    for r in records:
        if r.active:
            yield r.id

total = sum(active_ids(records))   # streams, no full list
```

הערה 5.2. חשיבה מונחית נתונים: שאלו מה באמת זורם. רשימה של מיליארד פריטים בזיכרון שונה מהותית מרצף שעיבדו תוך כדי קריאה. הפיכת צינור של list-בתוך-list לשרשרת גנרטורים היא אחד השיפורים המשמעותיים ביותר שאפשר לעשות, וגם אחד המשביעים: ה־diff קטן, הרווח גדול, והחיים חוזרים אליכם.

5.7 מתי לא לייעל

יש מקרים שבהם היד כבר רוצה לגעת בקוד. צריך לעמוד בפיתוי. קבלו את הרשימה:

- ◇ לפני שמדדו: רוב ה"ייעולים" מנחשים.
- ◇ בקנאה על קוד קריא: ברירות מחדל פשוטות עם caching זול עדיפות על תעלולי bit twiddling.

- בנקודות לא קריטיות: התמקדו ב־hot path שעלה ב־cProfile.
- כשזה מסבך את התחזוקה: הזמן של מתחזק שווה יותר מ־5% רווח.

ובמילה אחת: כנות. הקוד לא צריך להיראות חכם; הוא צריך לעבוד מהר, כשזה באמת משנה.

5.8 Anti-pattern → הדרך הטובה יותר

הדוגמה הבאה ממחישה אופטימיזציה ללא מדידה. הדבר המרשים בה: היא מעולם לא שאלה את הפרופיילר איפה הכאב האמיתי.

לא טוב: optimizing ללא מדידה

```
# what NOT to do
def process(rows):
    out = []
    for r in rows:
        # micro-optimized after guessing...
        out.append(r["a"] * 2)
    return out
# ...but the real cost was in load_all()
```

טוב: למדוד, למצוא, לטפל

```
import cProfile

def process(rows):
    return [r["a"] * 2 for r in rows]

rows = [{"a": i} for i in range(1000)]
cProfile.run("process(rows)")   # where is the actual cost?
```

הדרך הנכונה

מדדו → תמצאו את ה-cumtime הגדול → תטפלו שם → מדדו שוב. אופטימיזציה בלי **הלולאה** הזו היא ניחוש במקרה הטוב, ובמקרה הרע, היא שדרוג בפרוד ב־2 לפנות בוקר.

5.9 תרגיל קצר

בדקו את עצמכם, לא את מזלכם:

תרגיל 5: מצאו את הצוואר

1. כתבו פונקציה שמבצעת sort על רשימה גדולה, split על מחרוזות, וחישוב יקר פעמים רבות על אותו קלט.
2. הריצו cProfile ומצאו איפה ה-cumtime.
3. הוסיפו `lru_cache` במקום הנכון, והריצו שוב. מה השתנה?
4. (רמה שניה) בדקו את צריכת הזיכרון עם tracemalloc לפני ואחרי המעבר ל־generator.

פרק 6: ארכיטקטורה נקייה ו־Dependency Injection

ארכיטקטורה נקייה אינה מותרות "לעתיד". היא הכלי שמאפשר להחליף database, להוסיף provider חדש, או לשנות מדיניות עסקית בלי לכתוב מחדש חצי פרויקט. החלפת DB קטנה יכולה להפוך לסיפור של שבועיים בלי הפרדה נכונה. בפרק זה נבנה הפרדת שכבות (תחום, אפליקציה, תשתית), נחבר אותן ב־Injection Dependency ידני (בלי framework) ונבדוק כל רכיב בבידוד.

6.1 הפרדת שכבות: Domain, Application, Infrastructure

הבסיס של הפרק הזה: שלוש השכבות ותפקידיהן. ובניגוד לחיים, כאן הפרדה חדה בין שכבות היא דבר טוב:

שכבה	תפקיד
Domain	כללי העסק: models, חוקים, ערכים
Application	use cases: תיאום בין ה־אובייקטים
Infrastructure	טכנולוגיה: DB, HTTP, קבצים

הכלל הקריטי: **השכבה הפנימית אינה יודעת על החיצונית**. ה־Domain והלוגיקה של ה־Application מגדירים interfaces (כפי שנראה מיד); וה־Infrastructure היא שממשת

אותם. ככה ה־Postgres, ה־Stripe וה־filesystem הם פרטי מימוש שאפשר להחליף. להחליף Postgres בפרויקט בלי שכבות מבהיר למה אבותינו הקיפו ערים בחומה.

Domain: מודל טהור

```python
from dataclasses import dataclass

@dataclass(frozen=True)
class Order:
    id: str
    total: float

    def with_discount(self, pct: float) -> "Order":
        return Order(id=self.id, total=self.total * pct)
```


6.2 ה-Application והממשקים שלו

Application: use case עם Protocols

```python
from typing import Protocol

class OrderRepo(Protocol):
    def get(self, order_id: str) -> Order: ...

class PaymentGateway(Protocol):
    def charge(self, order: Order) -> None: ...

class Checkout:
    def __init__(
        self, orders: OrderRepo, payments: PaymentGateway
    ) -> None:
        self.orders = orders
        self.payments = payments

    def complete(self, order_id: str) -> None:
        order = self.orders.get(order_id)
        self.payments.charge(order)
```

שימו לב: Checkout יודע מה לבקש (`orders.get` ו־`payments.charge`), אבל לא מאיפה מגיעות התוצאות. זוהי בדיוק הבטחת Dependency Inversion: תלות בכיוון המופשט, לא במימוש. (הדבר הכי קרוב לזה בחיים: הבוס יודע מה לבקש, לא בהכרח איך.)

6.3 התשתית ממשת, composition root מחבר

Infrastructure: מימושים

```
class PostgresOrderRepo:
    def __init__(self, db) -> None:
        self.db = db

    def get(self, order_id: str) -> Order:
        row = self.db.fetch_one("SELECT ...", order_id)
        return Order(id=row["id"], total=row["total"])

class StripeGateway:
    def __init__(self, api_key: str) -> None:
        self.api_key = api_key

    def charge(self, order: Order) -> None:
        stripe_client.charge(order.id, order.total)
```

החיבור בין כל החלקים נעשה במקום אחד בלבד: נקודת הרכבה (composition root). תחשבו על זה כעל lineup של קבוצה: כולם באימון; אבל רק המאמן מחליט מי פותח. בכל מקום אחר אין צורך ליצור אובייקטים:

קומפוזיציה: פעם אחת

```
import os

def build_app() -> Checkout:
    db = create_pool(os.environ["DATABASE_URL"])
    orders = PostgresOrderRepo(db)
    payments = StripeGateway(os.environ["STRIPE_KEY"])
    return Checkout(orders, payments)

app = build_app()
app.complete("ord-42")
```

הערה 6.1. עד כאן בלי framework של DI. כשהפרויקט גדל אפשר להוסיף מנגנון container

(כמו dependency_injector או punq), אבל ההרגל לבנות composition root נקי משתלם עוד לפני שמכניסים כלים: framework לא יציל מהרגלים רעים; הוא רק ייתן להם שמות יפים.

6.4 בדיקה בבידוד

אחד היתרונות האהובים עליי: ה־Application תלוי ב־Protocols. להחליף את התשתית ב־fake בבדיקות, בלי mock מורכבים:

בדיקת Checkout עם fakes

```
class FakeOrderRepo:
    def __init__(self) -> None:
        self.orders = {
            "ord-1": Order(id="ord-1", total=100.0)
        }

    def get(self, order_id: str) -> Order:
        return self.orders[order_id]

class FakeGateway:
    def __init__(self) -> None:
        self.charged: list[str] = []

    def charge(self, order: Order) -> None:
        self.charged.append(order.id)

def test_complete_charges_order():
    checkout = Checkout(FakeOrderRepo(), FakeGateway())
    checkout.complete("ord-1")
    assert checkout.payments.charged == ["ord-1"]
```

הבדיקה רצה מהר, ללא רשת וללא database. היא גם מגדירה contract: אם Stripe או Postgres מפרים אותו, זה יתגלה בבדיקת אינטגרציה ייעודית (פרק 8), לא אצל לקוח. זו הדרך לישון בלילה: הבדיקה אומרת "לא" בזמן, לפני שמישהו מגלה את זה בפרוד.

6.5 Composition במקום inheritance

אחד הדפוסים השכיחים בפרויקטים. ובדרך כלל גם הסיבה שהם הסתבכו. תראו:

לא טוב: ירושה כדי למחזר התנהגות

```python
# what NOT to do
class EmailService:
    def send(self, msg: str) -> None:
        smtp.send(msg)

class AuditedEmailService(EmailService):
    def send(self, msg: str) -> None:
        super().send(msg)
        logger.info("sent", msg=msg)   # buried feature
```

טוב: הרכבה של שירותים

```python
class Notifier:
    def __init__(self, sender, logger) -> None:
        self.sender = sender
        self.logger = logger

    def notify(self, msg: str) -> None:
        self.sender.send(msg)
        self.logger.info("sent", msg=msg)
```

מתי ירושה לגיטימית

כשיש באמת יחס is-a והירושה משמשת ל־polymorphism על משפחה ידועה. כשהיא משמשת **למחזור קוד** או להרחבת התנהגות של מחלקה יחידה, הרכבה ו־DI נקיים יותר. (ואם אתם עושים ירושה כדי לחסוך שלוש שורות — תשאלו את עצמכם מי יטפל בקוד בעוד שנה.)

6.6 Anti-pattern → הדרך הטובה יותר

ראיתם את הדפוס הזה בפרויקטים. ואם לא, מזלכם שלמדתם לפני. הדרך הלא נכונה:

לא טוב: Service שמכיר את התשתית


```
# what NOT to do
class Checkout:
    def complete(self, order_id: str) -> None:
        conn = psycopg2.connect(DATABASE_URL)   # infra here
        row = conn.execute(...)
        stripe_api_key = os.environ["STRIPE_KEY"]   # and here
        stripe.charge(row["id"], row["total"])
```

טוב: תלות במופשט בלבד

```
class Checkout:
    def __init__(self, orders: OrderRepo,
                 payments: PaymentGateway) -> None:
        self.orders = orders
        self.payments = payments

    def complete(self, order_id: str) -> None:
        order = self.orders.get(order_id)
        self.payments.charge(order)
```

6.7 תרגיל קצר

משימה קטנה עם תמורה גדולה: תתפלאו כמה זה ברור כשמרכיבים אותו בעצמכם:

תרגיל 6: שירות הודעות מופרד

1. בנו Domain פשוט: Message (`to`, `text`).
2. Application: Sender ש־use case שלו הוא `send(msg)`: תלוי ב־Protocol Transport (`deliver`).
3. Infrastructure: שני מימושים: SmtpTransport ו־LogTransport.
4. composition root: פונקציית `build()` שמרכיבה את ה־Sender.
5. (רמה שניה) בדקו את ה־Sender עם fake, מבלי mock, בלי רשת.

פרק 7: אריזה ומבנה פרויקט

בפרויקטים ראשונים, כל תיקייה היא פרויקט בפני עצמו; וה־deploy הוא להעתיק קבצים לשרת ולקוות לטוב. לוקח זמן להפנים את מה שנראה מובן מאליו: פרויקט שמתקין כמו חבילה רגילה — ולא כמו קולאז׳ של תיקיות — קל יותר לשתף, לנהל את הגרסאות שלו, ולתלות בו כספרייה. בפרק זה נבנה מבנה src/ מודרני, נגדיר הכול בקובץ pyproject.toml יחיד, נכיר את hatch, poetry ו־flit, ונחבר entry points ל־scripts ול־plugins.

7.1 מבנה src/ מודרני

```
myproject/
├── pyproject.toml
├── README.md
├── src/
│   └── myproject/
│       ├── __init__.py
│       ├── cli.py
│       └── core.py
└── tests/
    ├── test_cli.py
    └── test_core.py
```


למה `src/`? החבילה אינה שוכבת ישירות בשורש, כך ש־import מתבצע רק דרך החבילה המותקנת. הבדיקות משתמשות במה שהותקן, לא בקוד שבשוגג. `import myproject` עובד בכל מקום, גם מחוץ לתיקיית הפרויקט. קוד שעבד "במקרה" בגלל מבנה התיקיות גרם ללילות לבנים רבים.

7.2 pyproject.toml: הכול במקום אחד

קובץ ההגדרה המודרני מחליף את `setup.py` ו־`setup.cfg`, ומרכז את ה־dependencies שהיו ב־`requirements.txt`:

pyproject.toml מינימלי

```toml
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "myproject"
version = "0.1.0"
description = "A tiny production-ready package"
requires-python = "⩾3.11"
dependencies = ["structlog⩾24,<26"]

[project.scripts]
mycli = "myproject.cli:main"

[tool.hatch.build.targets.wheel]
packages = ["src/myproject"]
```

הכול במקום אחד

ה־build backend מטפל באריזה, dependencies מגדיר את התלות של ה־runtime, ו־project.scripts יוצר פקודת CLI עם entry point — והכול במקום אחד, קריא ומגרס. מי שחווה את העידן של setup.py לצד requirements.txt שסותרים זה את זה, יודע להעריך רגעים כאלה.

7.3 ההתקנה בפועל: hatch, poetry, flit

שלושת הכלים המובילים, בקצרה:

- hatch: מהיר, מצוין ל־environments ו־scripts, עם ניהול גרסאות מובנה.
- poetry: lock file קפדני ו־dependency resolver מובנה.

- flit: קל ופשוט, מושלם לחבילות קטנות.

```
$ pip install -e .          # editable: dev + install in one
# or with poetry
$ poetry add structlog
```


העיקרון המשותף: **לא** מתקינים dependencies ידנית. הכלי קורא מ־pyproject.toml. מי שרוצה שכל הצוות יריץ את אותן גרסאות מוסיף lock file (poetry.lock, uv.lock או pip-tools).

הערה 7.1. הרגע שבו מישהו ב־standup אומר "אצלי זה עובד" וכל השאר מרימים גבה. זה הרגע שבו מבינים למה בכלל יש lock file. אל תהיו כמוהו.

7.4 Entry points: scripts ו־plugins

ה־entry point `mycli` בפייתון:

CLI של החבילה

```python
import argparse

def main() -> None:
    parser = argparse.ArgumentParser(prog="mycli")
    parser.add_argument("--name", default="world")
    args = parser.parse_args()
    print(f"hello {args.name}")

if __name__ == "__main__":
    main()
```

ואפשר להשתמש ב־entry points למנגנון plugin: כל חבילה חיצונית מצהירה על נקודת הרחבה; וה־host מגלה ומטעין אותן:

טעינת plugins עם importlib.metadata

```python
from importlib import metadata

def load_output_plugins() -> list[object]:
    plugins: list[object] = []
    for ep in metadata.entry_points(group="myproject.output"):
        plugins.append(ep.load())
    return plugins
```

חבילת plugin מכריזה על עצמה

```toml
# in the plugin's pyproject.toml
[project.entry-points."myproject.output"]
json = "myplugin.jsonout:JsonOutput"
```

הערה 7.2. importlib.metadata היא ה־API המודרני ל־entry points. בפייתון 3.12+ אין צורך ב־select: `entry_points(group=...)` מסנן נכון כברירת מחדל. כך בונים plu-gin systems בלי גיוס דינמי ובלי exec. plugin system שנבנה עם exec עובד, עד שהוא מפסיק לעבוד.

7.5 גרסאות ותלות

- מקור אחד לאמת: `importlib.metadata.version("myproject")` או `__version__` במודול `__init__`.
- Semantic Versioning: major לשבירת API, minor לתכונות, patch לתיקונים.
- טווחי תלות צרים ומתועדים: `structlog` ≥ 24, <26; מרווח לתאימות, חסם לגרסה עתידית שוברת.
- lock file לקריאה מהירה (reproducibility): לא לנעול תלות לא רלוונטית.

7.6 Anti-pattern → הדרך הטובה יותר

קוד שנראה מושלם הוא לרוב קוד שעוד לא ארוז כמו שצריך. ההבדל בין חבילה אמיתית לערימת קבצים רואים בעין:

לא טוב: כל הקוד בשורש


```
# what NOT to do
myproject/
├── app.py          # imports work by accident
├── utils.py
└── requirements.txt   # hand-synced versions
```

טוב: חבילה אמיתית

```
myproject/
├── pyproject.toml     # single source of truth
├── src/myproject/
│   ├── app.py
│   └── utils.py
└── tests/
```

סימן לפרויקט בוגר

אם פותח חדש בצוות מריץ `pip install -e .` ואז `mycli` (וזה עובד), החבילה בנויה נכון. אם צריך לשנות PYTHONPATH או להעתיק קבצים, יש עבודה.

7.7 תרגיל קצר

תרגיל 7: חבילה קטנה

1. בנו פרויקט עם מבנה `src/`: מודול `greeter` שמחזיר הודעת ברכה.
2. כתבו `pyproject.toml` מלא עם `hatchling`, `dependencies` ותלות ב־`pytest` (מסגרת בדיקות) (בסעיף `[dependency-groups]`).
3. הגדירו `sayhi` entry point שרץ מה־CLI.
4. (רמה שניה) הוסיפו plugin חיצוני קטן שנטען דרך entry point `myproject.greeter`.

אם סיימתם תוך עשר דקות, מעולה. ואם אתם מרגישים צורך לגלול אחורה ולראות "על מה בעצם כתבנו פה", גם בסדר. קרה לכולנו.

פרק 8: אסטרטגיות בדיקות מתקדמות

התחושה מוכרת: הבדיקות רצות, ה־CI ירוק, ולפני deploy הכול נראה לפי הספר, עד שהפרוד מספר סיפור אחר. בדיקות טובות מעניקות ביטחון; בדיקות חכמות מעניקות ביטחון בלי להישבר על כל שינוי.

בפרק זה נלמד על fixtures ו־parametrization מתקדמים ב־pytest, על mocking נכון (ומתי להימנע), על property-based testing עם hypothesis, על בדיקת קוד async, ועל contract tests שמגנים על הממשקים שלנו.

8.1 Fixtures: scope, yield ו־conftest

fixture הוא מנגנון dependency injection של pytest: הבדיקה מקבלת מה שהיא צריכה, וכל מה שנוצר מתנקה בסוף:

fixture עם ניקוי

```
import pytest

@pytest.fixture
def db():
    conn = connect(TEST_URL)
    yield conn            # setup before, teardown after
    conn.close()

def test_insert(db):
    db.execute("INSERT INTO t VALUES (1)")
    assert db.count("t") == 1
```


- `scope="session"`: פעם אחת לכל הריצה (למשל URL ל־database).
- `scope="module"`: פעם אחת לכל קובץ.
- conftest.py: הגדרה משותפת לכל תיקיית הבדיקות, בלי import ידני.
- fixture יכול לצרוך fixture אחר, ובונים מחסנית עם סדר התנקה הפוך.

הערה 8.1. כלל מקובל לצוותים: module scope מספיק לרוב; session scope שמור למה שבאמת יקר ליצור. session fixture של database שמדליף מצב גורם ללילות ארוכים של דיבוג. אי־הבנה של scope עולה יותר מ־bug; היא עולה שעות.

8.2 Parametrize ו־markers

ריצת מקרים עם parametrize

```python
import pytest

@pytest.mark.parametrize("n,expected", [
    (1, "odd"), (2, "even"), (3, "odd"), (0, "even"),
])
def test_parity(n: int, expected: str) -> None:
    assert classify(n) == expected
```


markers מאפשרים לסנן ולסווג:

סימון בדיקות

```python
@pytest.mark.slow
def test_full_integration() -> None:
    ...

@pytest.mark.asyncio
async def test_fetch() -> None:
    ...
```

```
$ pytest -m "not slow"          # fast suite
$ pytest -m slow                # night build only
```

רשמו את ה־markers ב־pyproject.toml כדי להימנע מ־warnings וליצור מסמך אמת:

הגדרת markers

```
[tool.pytest.ini_options]
markers = ["slow: long-running tests",
           "asyncio: async tests"]
```

(העצה הזו נכתבה אחרי לילה שבו בדיקות slow רצו ב־CI של כל push, כי מישהו לא רשם marker.)

8.3 Mocking נכון: ומתי להימנע

mock מחליף חלק חיצוני כדי לבדוק התנהגות, אבל אם אתם mock את הקוד שלכם עצמכם, כנראה הארכיטקטורה (פרק 6) צריכה שינוי.

הערה 8.2. אבחנה מהירה ב־code review: ככל שהבדיקה mock יותר, היא מספרת פחות. אם אתם mock את מה שאתם בודקים, הבדיקה לא תיכשל גם כשהקוד שבור — ותופעה מוכרת היא בדיקה שנשארת ירוקה שבוע שלם על קוד שבור.

monkeypatch לגבול חיצוני

```python
def test_retry_on_network(monkeypatch):
    calls = {"n": 0}

    def flaky(url):
        calls["n"] += 1
        if calls["n"] < 3:
            raise ConnectionError("down")
        return b"ok"

    monkeypatch.setattr(requests, "get", flaky)
    assert fetch_with_retry("http://x") == b"ok"
    assert calls["n"] == 3
```

מתי mock לגיטימי

- גבולות חיצוניים: network, שעון, filesystem, env.
- בדיקת התנהגות ספציפית: "נקרא לגייטוויי פעם אחת".
- מתי **לא**: כשמחליפים את הלוגיקה שבדקים, או כשמקובל fake פשוט יותר (בדיקות פרק 6).

8.4 Property-based testing עם hypothesis

במקום שלוש דוגמאות מומצאות, hypothesis מייצר מאות מקרים ומכווץ כל כשל ל-counterexample מינימלי:

hypothesis על sorted

```python
from hypothesis import given, settings
from hypothesis import strategies as st

@given(st.lists(st.integers()))
@settings(max_examples=200)
def test_sort_is_ordered(items: list[int]) -> None:
    result = sorted(items)
    for a, b in zip(result, result[1:]):
        assert a <= b

@given(st.lists(st.integers()))
def test_sort_is_idempotent(items: list[int]) -> None:
    once = sorted(items)
    assert sorted(once) == once
```

הערה 8.3. היזהרו: property צריך להיות נכון עבור **כל** קלט; זה הכוח וגם המלכודת. אם ההנחה לא תקפה לכל הקלטים, נוקטים assume או filter. וה־counterexample שמחזיר hypothesis הוא זהב לאבחון באגים. הפעם הראשונה ש־hypothesis מוצא bug תוך שניות (בעוד חיפוש ידני לוקח שבוע) משכנעת לכל החיים.

8.5 בדיקת קוד async

בדיקה async

```python
import asyncio
import pytest

@pytest.mark.asyncio
async def test_fetch_returns_bytes() -> None:
    data = await fetch("https://a.io/1")
    assert data.startswith(b"<")

@pytest.mark.asyncio
async def test_timeout_fallback() -> None:
    with pytest.raises(TimeoutError):
        async with asyncio.timeout(1):
            await slow_endpoint()
```

- pytest-asyncio: דגל asyncio על הבדיקה, או anyio שמאפשר לבדוק גם asyncio וגם trio.
- אל תנסו mock את Loop Event (לולאת אירועים); בדקו התנהגות, לא מנגנון.
- ל־async משולב עם fixtures: yield fixture עובד גם בבדיקות async.

הערה 8.4. הכלל "אל תנסו mock את ה־event loop" כתוב בדם: mockתי פעם את ה־loop כדי "להריץ מהר"; והבדיקה עברה בלי להריץ כלום. ירוקה, אבל הקוד לא עבד. מאז: loop אמיתי, תמיד.

8.6 Contract tests: ממשקים מוגנים

כשיש Protocol עם כמה מימושים (פרק 6), בדיקת contract אחת רצה על כולם:

Contract test למאגר

```python
def test_repo_contract(repo) -> None:
    repo.save(User(id=1, name="Dana"))
    assert repo.find(1).name == "Dana"
    assert repo.find(999) is None

def test_postgres_repo(db):
    test_repo_contract(PostgresRepo(db))

def test_memory_repo():
    test_repo_contract(InMemoryRepo())
```


עיקרון

מגדירים את החוזה פעם אחת, ורצים אותו על כל מימוש. כל מימוש חדש עובר את אותן דרישות; וכל שינוי בחוזה נמדד מיד. השילוב של contract testing עם Protocols (פרק 6) נותן רשת ביטחון זולה וערכית במיוחד — כשכל המימושים נשברים יחד, סימן שהחוזה הוא שהציל את הפרויקט מגרסה שבורה.

8.7 Anti-pattern → הדרך הטובה יותר

דפוס הקוד הבא נפוץ מדי: mock על הכול, ובדיקה שבודקת את ה־mock של עצמה:

לא טוב: mock על הכל

```
# what NOT to do
def test_checkout():
    mock_repo = Mock()
    mock_repo.get.return_value = Order("o1", 10.0)
    mock_gateway = Mock()
    checkout = Checkout(mock_repo, mock_gateway)
    checkout.complete("o1")
    mock_gateway.charge.assert_called_once()
# tests the mocks, not the behavior
```

טוב: fake + בדיקה של התנהגות

```
def test_checkout_real_behavior():
    repo = InMemoryRepo(Order("o1", 10.0))
    gateway = RecordingGateway()
    Checkout(repo, gateway).complete("o1")
    assert gateway.charged == ["o1"]
```


8.8 תרגיל קצר

תרגיל 8: חבילת בדיקות

1. קחו את Notifier מפרק 6 (`send(msg)` דרך Transport).
2. כתבו בדיקות עם parametrize: כמה סוגי הודעות, כולל מחרוזת ריקה.
3. בדקו retry על Transport שנופל, עם monkeypatch, בלי רשת.
4. (רמה שניה) כתבו property-based test על פונקציית normalize_text שלכם, עם hypothesis, ומצאו את המקרה ששובר אותה.

ואם hypothesis לא מוצא כלום, אל תחגגו מיד. חפשו invariant טוב יותר. ה־bug תמיד שם.

9

פרק 9: מוכנות בפרוד ו־Observability

יש רגע שבו שירות "עובד" ועדיין מרגיש שביר. אם הוא קורס ב־3 בלילה, תגלו על זה רק מהטלפון של ה־on-call. קוד שמוכן בפרוד שותה מים מהתצורה, מדווח על עצמו, מגיב לאותות ומאפשר להדליק ולכבות תכונות בלי deploy. בפרק זה נשלים את מעגל ה־observability (לוגים, מדדים ו־traces) עם OpenTelemetry (טלמטריה פתוחה) בסיסי, ננהל הגדרות לפי 12-factor, נבנה health checks, graceful shutdown ו־feature flags.

9.1 שלושת עמודי ה־Observability

- Logs: אירועים מפורטים (לוגים מובנים מפרק 2).
- Metrics: מספרים מצטברים (counters, gauges).
- Traces: מסלול בקשה אחת דרך כל הרכיבים (spans).

כל אחד מהם עונה על שאלה אחרת. logs: **מה** קרה? metrics: **כמה**? traces: **איפה** זה לקח זמן?

הערה 9.1. האנלוגיה האהובה: logs זה היומן של האחיות; metrics זה מד־הלחץ; ו־traces זה המסלול של הבקשה דרך כל המחלקות. שירות בלי שלושתם הוא חולה בלי תיק רפואי; ב־Kubernetes אף אחד לא יודע מי הוא.

9.2 Metrics ו־traces עם OpenTelemetry

Metric: מונה פשוט

```python
from opentelemetry import metrics

meter = metrics.get_meter("shop")
created = meter.create_counter("orders.created")

def create_order(user_id: str) -> None:
    ...
    created.add(1, {"status": "ok"})
```

Trace: איבר במסלול

```python
from opentelemetry import trace

tracer = trace.get_tracer("shop")

def handle(req) -> None:
    with tracer.start_as_current_span("handle") as span:
        span.set_attribute("order_id", req.order_id)
        process(req)   # creates child spans
```

החוט המשותף

המפתח המחבר בין שלושת העמודים הוא ה־trace ID, אותו correlation IDs מפרק 2. עם OpenTelemetry ה־SDK (למשל OTLP אל Jaeger או Tempo) מקבלים distributed tracing על פני כל השירותים בלי לכתוב אותו ידנית. מרגע שחיברתם trace ID אחד ללוגים, שאלת "איפה בדיוק הבקשה איבדה זמן?" מקבלת תשובה תוך דקה. התקופה שלפני כן לא מתגעגעים אליה.

9.3 ניהול תצורה לפי 12-factor

החוק: כל מה שמשתנה בין סביבות מגיע מ־env variables. לא מקובצי קוד, לא מ־constants, לא מ־git. (לשמור config ב־git? קורה לכולנו. לא ננקוב שמות.)

תצורה מה־env

```python
import os

def env_int(name: str, default: int) -> int:
    return int(os.environ.get(name, str(default)))

class Settings:
    database_url = os.environ["DATABASE_URL"]
    port = env_int("PORT", 8080)
    debug = os.environ.get("DEBUG", "0") == "1"
```

הערה 9.2. לפרויקטים גדולים, pydantic-settings או dynaconf עושים את אותו הדבר, עם אימות (וב־dynaconf גם עדכון חי). שימו לב: בדוגמה לעיל ה־env נקרא בזמן ייבוא המודול — מצוין לשירות שעולה פעם אחת, ומסורבל בבדיקות; לפרויקטים עם צרכים כאלה, בנו מופע Settings באתחול ולא כמאפייני מחלקה. העקרון נשאר: מפתחות env חד־משמעיים, ברירת מחדל מפורשת, וללא סודות בקוד או ב־git.

9.4 Health checks: liveness ו־readiness

שני endpoints עם תפקידים שונים:

שני מסלולי בריאות

```python
# liveness: is the process alive?
def liveness() -> dict:
    return {"status": "ok"}

# readiness: can it serve traffic?
def readiness(db, cache) -> dict:
    if not db.is_ready():
        return {"status": "not_ready", "reason": "db"}
    if not cache.is_ready():
        return {"status": "not_ready", "reason": "cache"}
    return {"status": "ready"}
```


- liveness: שה־orchestrator (למשל Kubernetes) ידע אם להפעיל מחדש.
- readiness: להפסיק לשלוח traffic עד שהשירות מוכן.
- אל תבדקו ב־readiness dependencies שמחוץ לאחריות; רק את מה שמשפיע על היכולת לשרת.

הערה 9.3. הקריסה הקלאסית: readiness שבודק DB של שירות אחר; ה־cluster "מנותק" כי אותו DB נפל. readiness צריך לענות על שאלה אחת: אפשר לשרת traffic עכשיו? לא יותר.

9.5 Graceful shutdown: כיבוי נקי

כשמגיע SIGTERM, רוצים: להפסיק לקבל בקשות חדשות, לסיים את הקיימות, ורק אז לסגור משאבים:

כיבוי על ידי signal

```
import signal
import threading

stop = threading.Event()

def on_signal(signum, frame):
    logger.info("shutdown requested")
    stop.set()

signal.signal(signal.SIGTERM, on_signal)
signal.signal(signal.SIGINT, on_signal)

while not stop.is_set():
    serve_one_request()   # accept a request

logger.info("draining in-flight requests")
drain()
pool.close()
```


הערה 9.4. איך יודעים שה־graceful shutdown באמת עובד? מריצים load test, שולחים SIGTERM באמצע, ומסתכלים על ה־in-flight: אם הם לא נקטעו, סיימתם. אם נקטעו, לפחות עכשיו אתם יודעים מה לתקן — לפני הלילה שבו זה יקרה בפרוד.

כלל

כיבוי פתאומי הורג in-flight בקשות ומשאיר מצבים חצי־כתובים. graceful shutdown (עם signal handler, Event ו־drain) הוא תנאי הכרחי ל"אפס-השבתה" בעדכוני פרוד.

9.6 Feature flags ותצורה דינמית

feature flag מאפשר להדליק תכונה בלי deploy, ועוד יותר חשוב, לכבות אותה מהר כשמשהו נשבר: דגל שדורש deploy כדי לכבות הוא לא דגל; הוא תקווה. תקווה בפרוד היא מתכון לאסון.

דגל פשוט מהתצורה

```
import json

class FlagStore:
    def __init__(self) -> None:
        self._data: dict[str, bool] = {}

    def load(self, path: str) -> None:
        with open(path) as f:
            self._data = json.load(f)

    def flag(self, name: str) -> bool:
        return self._data.get(name, False)

flags = FlagStore()
flags.load("/etc/shop/flags.json")

if flags.flag("new_checkout"):
    checkout_new(req)
else:
    checkout_old(req)
```

הערה 9.5. עיקרון קריטי: דגל הוא deploy-time או runtime; ואת ה־runtime קוראים שוב בכל בקשה (או בכל X שניות). feature flags אמיתיים (כמו LaunchDarkly או Flagr) נותנים audit trail, גלגול הדרגתי וביטול מיידי. השיעור ששמרתי בלב: תכונה שנדלקה בפרוד וסיימה ב־rollback תוך חצי שעה, רק בגלל שהיה דגל. תודה, feature flag.

9.7 Anti-pattern → הדרך הטובה יותר

יש קוד ש"עובד בפרוד" רק כי יש לו מזל. הנה הוכחה מהחיים: הדרך הלא נכונה והדרך הנכונה:

לא טוב: config + print בקוד + exit פתאומי

```
# what NOT to do
import sys
DATABASE_URL = "postgres://prod"   # hardcoded config
print("started on", DATABASE_URL)  # log = print

PORT = 8080
DB = "postgres://prod"   # hardcoded config

sys.exit()               # no graceful shutdown
```

טוב: מדוות, מוגדר ומתנקה

```
import signal, threading
from opentelemetry import metrics

meter = metrics.get_meter("shop")
started = meter.create_counter("shop.started")

def main() -> None:
    cfg = Settings()                    # from env
    started.add(1)
    install_shutdown_handlers()         # signal + drain
    serve(cfg)
```

9.8 תרגיל קצר

תרגיל 9: שירות מוכן בפרוד

1. בנו Settings (PORT ,DATABASE_URL) עם ברירות מחדל, כולל feature flag FF_BETA.
2. כתבו liveness ו־readiness טהורים.
3. הוסיפו signal handler ל־SIGTERM עם graceful drain.
4. (רמה שניה) חברו metric (מונה בקשות) עם OpenTelemetry, גם אם ה־SDK רק מדפיס ללוג.

ולפני שסוגרים את הלפטופ, שלחו SIGTERM ותראו שהכול מתנקה יפה. קרה לי ששכחתי. ה־CI לא שכח.

פרק 10: מודל האובייקטים של פייתון

10

מודל האובייקטים של פייתון הוא הלב הפועם של השפה. רוב המפתחים משתמשים ב־`__init__` ו־`self` בלי להבין מה קורה מאחורי הקלעים. בפרק זה נצלול למנגנון היצירה (`__new__`), לתיאורים (descriptors) שמפעילים את `property`, `classmethod` ו־`staticmethod`, ל־`__slots__` שחוסך זיכרון, ולשאר שיטות ה־data model שהופכות אובייקט רגיל לאזרח מלא בפייתון. הבנה זו היא המפתח לכתיבת קוד שמתנהג כמו built-in; ולתיקון באגים מוזרים שצצים כשהדברים לא עובדים כמצופה.

10.1 `__new__` מול `__init__`: יצירה מול אתחול

כשכותבים `MyClass()`, פייתון קורא לשתי שיטות לפי סדר: קודם `__new__` (שמחזירה את המופע), ואז `__init__` (שמאתחלת אותו). רוב הזמן `__init__` מספיק; אבל כשצריך לשלוט ביצירת המופע עצמו (למשל singleton, flyweight, או החזרת מופע קיים ממטמון); `__new__` הוא הכלי.

__new__ מחזיר מופע קיים

```python
class Singleton:
    _instance: "Singleton | None" = None

    def __new__(cls) -> "Singleton":
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self) -> None:
        # runs on every call, but self is the same object
        if not hasattr(self, "initialized"):
            self.initialized = True
            self.value = 42

a = Singleton()
b = Singleton()
print(a is b)  # True
print(a.value) # 42
```


הערה 10.1. `__new__` היא static method סמויה: הפרמטר הראשון הוא `cls`, והיא **חייבת** להחזיר מופע של `cls` (או תת־מחלקה). אם לא מחזירים כלום, מקבלים `None` וכל הקוד שאחרי נכשל. `__init__` לעומת זאת מחזירה `None` תמיד.

כלל אצבע: מתי `__new__`?

- כמעט אף פעם. `__init__` מכסה 99% מהמקרים.
- `__new__` נחוץ כש: (א) רוצים להחזיר מופע קיים (סינגלטון, מאגר מופעים); (ב) יוצרים תת־מחלקה של טיפוס immutable כמו `tuple`, `str`, `frozenset`; (ג) צריך לשלוט בזיכרון ברמה נמוכה.
- ב־dataclasses ו־attrs: לעולם לא נוגעים ב־`__new__`.

10.2 Descriptors: property, classmethod ו־staticmethod

Descriptor (תיאורן) מגדיר את `__get__`, `__set__` ו/או `__delete__`: כשמאפיין של מחלקה הוא descriptor, הגישה אליו (`obj.attr`, `obj.attr = val`) מפעילה את השיטות במקום גישה למילון. זה הבסיס לכל ה־magic של פייתון.

הגדרה 10.1 (פרוטוקול Descriptor)**.** אובייקט d הוא descriptor אם יש לו לפחות אחת מהשיטות:

- `__get__(self, obj, objtype=None)` $\to$ Any; קריאה
- `__set__(self, obj, value)` $\to$ None; כתיבה
- `__delete__(self, obj)` $\to$ None; מחיקה

אם יש רק `__get__`; זה non-data descriptor (כמו פונקציות, `classmethod`). אם יש גם `__set__` או `__delete__`; זה data descriptor (כמו `property`).

descriptor פשוט: validation בזמן כתיבה

```
class Positive:
    def __init__(self, name: str) -> None:
        self.name = name

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        return obj.__dict__[self.name]

    def __set__(self, obj, value):
        if value <= 0:
            raise ValueError(f"{self.name} must be positive")
        obj.__dict__[self.name] = value

class Product:
    price = Positive("price")
    weight = Positive("weight")

    def __init__(self, price: float, weight: float) -> None:
        self.price = price
        self.weight = weight

p = Product(10.0, 2.5)
print(p.price)  # 10.0
p.price = -5    # ValueError: price must be positive
```


הערה 10.2. שימו לב ל־`obj is None` ב־`__get__`: כשניגשים דרך המחלקה (`Product.price`) ולא דרך מופע, `obj` הוא `None`; זה מאפשר להחזיר את ה-descriptor עצמו (לתיעוד ולכלים).

10.3 איך `property`, `classmethod`, `staticmethod` עובדים

הכול descriptors בספרייה הסטנדרטית:

מימוש מפושט של property

```
class Property:
    def __init__(
        self, fget=None, fset=None, fdel=None, doc=None,
    ):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        self.__doc__ = doc or (fget.__doc__ if fget else None)

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(obj)

    def getter(self, fget):
        return type(self)(
            fget, self.fset, self.fdel, self.__doc__,
        )

    def setter(self, fset):
        return type(self)(
            self.fget, fset, self.fdel, self.__doc__,
        )

    def deleter(self, fdel):
        return type(self)(
            self.fget, self.fset, fdel, self.__doc__,
        )
```

descriptor as classmethod

```
import functools

class ClassMethod:
    def __init__(self, func):
        self.func = func

    def __get__(self, obj, objtype=None):
        if objtype is None:
            objtype = type(obj)
        return functools.partial(self.func, objtype)

# usage:
class MyClass:
    @ClassMethod
    def factory(cls, value):
        return cls(value)
```

סדר עדיפות: descriptor מול `__dict__`

כשניגשים ל־`obj.attr`:

1. אם `attr` הוא data descriptor במחלקה (יש `__set__`); **מנצח תמיד**
2. אחרת, אם `attr` ב־`obj.__dict__`; מחזירים אותו
3. אחרת, אם `attr` הוא non-data descriptor במחלקה (רק `__get__`); מפעילים אותו
4. אחרת, מחפשים במחלקות האם (MRO (סדר פתרון שיטות))
5. אחרת: `AttributeError`

לכן `property` (שיש לו `__set__`) דורס ערך ב־`__dict__`, אבל פונקציה רגילה (רק `__get__`) לא.

10.4 `__slots__`: חיסכון בזיכרון ומניעת מאפיינים דינמיים

ברירת המחדל בפייתון: לכל מופע יש `__dict__`: מילון שמאפשר להוסיף מאפיינים בזמן ריצה. זה גמיש אבל יקר בזיכרון. `__slots__` מחליף את המילון במערך קבוע של מצביעים.

__slots__ בסיסי

```python
import sys

class Regular:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Slotted:
    __slots__ = ("x", "y")
    def __init__(self, x, y):
        self.x = x
        self.y = y

r = Regular(1, 2)
s = Slotted(1, 2)

print(sys.getsizeof(r), sys.getsizeof(r.__dict__))  # 48 296
print(sys.getsizeof(s))  # 48 bytes
```

השוואת זיכרון

```python
import sys

class Regular:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Slotted:
    __slots__ = ("x", "y")
    def __init__(self, x, y):
        self.x = x
        self.y = y

r = Regular(1, 2)
s = Slotted(1, 2)

print(sys.getsizeof(r), sys.getsizeof(r.__dict__))  # 48 296
print(sys.getsizeof(s))  # 48 bytes
```

```
48 296
48
```

שימו לב למה שהמספרים אומרים: `sys.getsizeof` של המופע עצמו זהה בשני המקרים (48 בייט). ההבדל הוא בתוספת: למופע הרגיל יש `__dict__` משלה שעולה עוד 296 בייט (ולאלה מתווספים האובייקטים שהמילון מחזיק). כשיש מיליוני מופעים, ההפרש הזה מצטבר לכמה מאות מגה.

הערה 10.3. `__slots__` לא עובד עם ירושה אלא אם **כל** המחלקות בשרשרת מגדירות `__slots__` (או לא מגדירות בכלל). אם מחלקת אב מגדירה `__slots__` והבן לא, לבן יהיה `__dict__` והחיסכון מתבטל.

מתי להשתמש ב־__slots__

- ◇ מיליוני מופעים (נקודות, חלקיקים, צמתים בעץ); חיסכון של 40-50% בזיכרון.
- ◇ כשצריך למנוע הוספת מאפיינים דינמיים (אבטחה, חוזה נוקשה).
- ◇ בונוס: גישה למאפיינים של slotted גם מהירה יותר, כי אין חיפוש במילון.
- ◇ **לא** כש: המופעים מעטים; צריך **kwargs דינמי; או משתמשים ב־multiple inheritance מורכב בלי תכנון.

10.5 שיטות Data Model חיוניות

פייתון קוראת לשיטות מיוחדות (עם קווים תחתיים כפולים) כשמבצעים פעולות מובנות. הנה החשובות ביותר:

השוואה, ייצוג, המרה

```python
from typing import Self

class Vector:
    __slots__ = ("x", "y")

    def __init__(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    # formal representation (for developers)
    def __repr__(self) -> str:
        return f"Vector({self.x}, {self.y})"

    # friendly representation (for users)
    def __str__(self) -> str:
        return f"({self.x}, {self.y})"

    # equality
    def __eq__(self, other: object) -> bool:
        if not isinstance(other, Vector):
            return NotImplemented
        return self.x == other.x and self.y == other.y

    # boolean conversion
    def __bool__(self) -> bool:
        return self.x != 0 or self.y != 0

    # length (for sequences)
    def __len__(self) -> int:
        return 2

    # index access
    def __getitem__(self, index: int) -> float:
        if index == 0:
            return self.x     94
        elif index == 1:
            return self.y
        raise IndexError("Vector index out of range")
```

הערה 10.4. החזרת `NotImplemented` (לא `False`) ב־`__eq__` אומרת לפייתון "נסה את הפעולה ההפוכה" (`other.__eq__(self)`). זה מאפשר השוואה סימטרית עם טיפוסים אחרים.

10.6 `__getattr__`, `__getattribute__`, `__setattr__`: יירוט גישה

- `__getattr__(self, name)`: נקרא **רק** כשהמאפיין לא נמצא בדרך הרגילה. שימושי ל־dynamic attributes או delegation.
- `__getattribute__(self, name)`: נקרא **תמיד** בכל גישה. מסוכן: קריאה ל־`self.x` בתוכו גורמת רקורסיה אינסופית; חייבים `super().__getattribute__`.
- `__setattr__(self, name, value)`: נקרא בכל השמה. שימושי ל-validation גלובלית או לוגינג.

`__getattr__` לדלגציה דינמית

```
class ServiceProxy:
    def __init__(self, service) -> None:
        self._service = service

    def __getattr__(self, name: str):
        # called only if name is missing on ServiceProxy
        return getattr(self._service, name)

    def __repr__(self) -> str:
        return f"ServiceProxy({self._service!r})"
```

__setattr__ גלובלית ל-validation

```
class Validated:
    def __setattr__(self, name: str, value) -> None:
        if (
            name.startswith("_")
            or name in {"id", "created_at"}
        ):
            super().__setattr__(name, value)
        elif value is None:
            raise ValueError(f"{name} cannot be None")
        else:
            super().__setattr__(name, value)
```


הבדל: __getattr__ מול property

- `property`: למאפיין **ידוע מראש** עם לוגיקה מותאמת (validation, חישוב, קאש).
- `__getattr__`: למאפיינים **דינמיים/לא ידועים מראש** (פרוקסי, ORM שטוען עמודות מהמסד).
- `__getattribute__`: נדיר; רק כשצריך ליירט **כל** גישה (פרופיילינג, אבטחה, debug).

10.7 Anti-pattern → הדרך הטובה יותר

לא טוב: שימוש מופרז ב־__getattr__

```
# what NOT to do
class MagicBag:
    def __getattr__(self, name):
        return f"you asked for {name}"

bag = MagicBag()
print(bag.hello)      # "you asked for hello"
print(bag.foo.bar)  # foo is a str, not an object
```

טוב: descriptor מפורש ל-validation

```python
# good: reusable descriptor
class ValidatedField:
    def __init__(self, validator, name: str) -> None:
        self.validator = validator
        self.name = name

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        return obj.__dict__[self.name]

    def __set__(self, obj, value):
        if not self.validator(value):
            raise ValueError(f"{self.name} failed validation")
        obj.__dict__[self.name] = value

class User:
    name = ValidatedField(
        lambda v: isinstance(v, str) and len(v) > 0, "name",
    )
    age = ValidatedField(
        lambda v: isinstance(v, int) and v >= 0, "age",
    )

    def __init__(self, name: str, age: int) -> None:
        self.name = name
        self.age = age
```


10.8 תרגיל קצר

תרגיל: LazyProperty descriptor

1. מימשו מחלקה LazyProperty שמתנהגת כמו property אבל מחשבת את הערך רק בפעם הראשונה ושומרת אותו ב־__dict__ של המופע.
2. השתמשו בו כדי לטעון הגדרות מקובץ רק כשניגשים אליהן לראשונה.
3. (רמה שניה) הוסיפו תמיכה ב־__delete__ כדי לאפשר איפוס המטמון.

רמז: מבנה בסיסי

```
class LazyProperty:
    def __init__(self, func):
        self.func = func
        self.name = func.__name__

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        # TODO: compute and cache if not present
        ...

    def __set__(self, obj, value):
        obj.__dict__[self.name] = value

    def __delete__(self, obj):
        # TODO: clear the cache
        ...
```

פרק 11: כלי פונקציונליים: functools ו־itertools

11

הספרייה הסטנדרטית של פייתון מכילה שני מודולים שהם אבן יסוד לכתיבת קוד נקי, יעיל ופונקציונלי: functools ו־itertools. הראשון מספק כלים לעבודה עם פונקציות: קישוט, קאש, הרכבה ויישום חלקי; השני מספק איטרטורים אינסופיים, קומבינטוריים, וכלים לעיבוד זרמים בלי לממש אותם בזיכרון. בפרק זה נכיר את הכלים השימושיים ביותר, נבין מתי כל אחד עדיף על לולאה רגילה, ונראה איך הם משתלבים עם generators ו־lazy evaluation שכבר פגשנו בפרק 5.

11.1 functools: כלי עבודה לפונקציות

functools מכילה דקורטורים וכלים שמתמרנים פונקציות כערכים; כל אחד מהם פותר בעיה ספציפית שבדרך כלל היינו פותרים עם לולאה או למבדה מסורבלת.

11.2 lru_cache ו־cache: ממוחזרים חישובים יקרים

הדקורטור `@lru_cache` שומר תוצאות של קריאות קודמות (Least Recently Used) ומחזיר אותן מיידית כשהארגומנטים חוזרים. מגרסה 3.9 יש גם `@cache`; גרסה פשוטה בלי הגבלת גודל.

lru_cache בסיסי

```python
from functools import lru_cache

@lru_cache(maxsize=128)
def fibonacci(n: int) -> int:
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(30))  # 832040 -- instant thanks to the cache
print(fibonacci.cache_info())  # CacheInfo(hits=28, misses=31)
```


הערה 11.1. `lru_cache` דורש שהארגומנטים יהיו hashable. רשימות, מילונים וקבוצות לא עובדים; צריך להמיר ל־`tuple` או `frozenset`. הפרמטר `maxsize=None` (או `@cache`) אומר "אין הגבלה"; שימושי כשמספר הקריאות הייחודיות קטן וידוע.

cache עם ארגומנטים מורכבים

```python
from functools import cache

class DB:
    def query(self, sql: str, user_id: int) -> dict:
        return {"user_id": user_id}

db = DB()

@cache
def fetch_user_data(
    user_id: int, include_posts: bool = False,
) -> dict:
    # expensive DB / API call
    return db.query(
        "SELECT * FROM users WHERE id = ?", user_id,
    )

# first call - runs the query
data1 = fetch_user_data(42)
# second call - served from cache instantly
data2 = fetch_user_data(42)
assert data1 is data2  # exactly the same object
```


11.3 partial ו־partialmethod: יישום חלקי

`functools.partial` יוצר פונקציה חדשה עם חלק מהארגומנטים כבר קבועים. זה תחליף נקי לפונקציות למבדה כשצריך לשמור על חתימה קריאה וכלי טיפוס.

partial במקום למבדה

```python
from functools import partial
import requests

# instead of: lambda url: requests.get(
#     url, timeout=5, headers={"User-Agent": "my-bot"},
# )
fetch = partial(
    requests.get, timeout=5, headers={"User-Agent": "my-bot"},
)

# fetch is now a regular function with a clear signature
response = fetch("https://api.github.com/users")
```


partialmethod למתודות מחלקה

```python
from functools import partialmethod

class Button:
    def __init__(self, label: str) -> None:
        self.label = label

    def click(self, source: str) -> None:
        print(f"{self.label} clicked from {source}")

    # bind source in advance
    click_from_ui = partialmethod(click, source="UI")
    click_from_api = partialmethod(click, source="API")

btn = Button("Submit")
btn.click_from_ui()    # Submit clicked from UI
btn.click_from_api()   # Submit clicked from API
```

partial מול partialmethod

- `partial`: מחזיר callable רגיל; לא מעביר `self` אוטומטית.
- `partialmethod`: מיועד למתודות; כשנקרא על מופע, `self` מועבר אוטומטית לפני הארגומנטים הקבועים.

11.4 wraps: שומרים על metadata בדקורטורים

בלי `@wraps`, דקורטור מוחק את `__name__`, `__doc__`, `__annotations__` של הפונקציה המקורית; שובר introspection, `help()` וכלי טיפוס.

למה צריך wraps

```python
from functools import wraps

def logged(fn):
    @wraps(fn)  # without it: inner.__name__ == "inner"
    def inner(*args, **kwargs):
        print(f"Calling {fn.__name__}")
        return fn(*args, **kwargs)
    return inner

@logged
def add(a: int, b: int) -> int:
    """Add two numbers."""
    return a + b

print(add.__name__)       # "add" (not "inner")
print(add.__doc__)        # "Add two numbers."
print(add.__annotations__)  # {'a': <class 'int'>, ...}
```

11.5 singledispatch, singledispatchmethod: פולימורפיזם לפי טיפוס

פונקציה שמתנהגת אחרת לפי טיפוס הארגומנט הראשון; בלי `if isinstance` שרשרת.

singledispatch בסיסי

```python
from functools import singledispatch

@singledispatch
def serialize(obj) -> str:
    raise NotImplementedError(f"Cannot serialize {type(obj)}")

@serialize.register
def _(obj: int) -> str:
    return f"int:{obj}"

@serialize.register
def _(obj: str) -> str:
    return f"str:{obj}"

@serialize.register
def _(obj: list) -> str:
    return "list:[" + ",".join(
        serialize(item) for item in obj
    ) + "]"

print(serialize(42))           # int:42
print(serialize("hello"))      # str:hello
print(serialize([1, "two", 3])) # list:[int:1,str:two,int:3]
```

singledispatchmethod למתודות

```python
from functools import singledispatchmethod

class EventHandler:
    @singledispatchmethod
    def handle(self, event):
        raise NotImplementedError(
            f"Unknown event: {type(event)}"
        )

    @handle.register
    def _(self, event: str):
        print(f"Text event: {event}")

    @handle.register
    def _(self, event: dict):
        print(f"Dict event: {event.get('type')}")

h = EventHandler()
h.handle("hello")
h.handle({"type": "click", "x": 10})
```


11.6 cmp_to_key: המרת פונקציית השוואה למפתח מיון

פייתון 3 הסירה cmp מ־sort; cmp_to_key מגשרת.

cmp_to_key למיון מותאם

```
from functools import cmp_to_key

# sort by length, then alphabetically
def compare(a: str, b: str) -> int:
    if len(a) != len(b):
        return len(a) - len(b)
    return (a > b) - (a < b)

words = ["apple", "pie", "banana", "a", "zebra"]
words.sort(key=cmp_to_key(compare))
print(words)  # ['a', 'pie', 'apple', 'zebra', 'banana']
```


11.7 כלים נוספים ששווה להכיר

- `reduce(func, iterable[, initial])` – קיפול (fold) שמאלי; שימושי לסכום, מכפלה, מיזוג מילונים.
- `total_ordering` – מחלקה שמגדירה `__eq__` ואחת מ־`__lt__`, `__le__`, `__gt__`, `__ge__` מקבלת את השאר אוטומטית.
- `cached_property` (3.8+) – כמו `property` אבל מחשב פעם אחת ושומר במופע.
- `singledispatch` תומך גם ב־type unions ב־3.11+ עם `register(int | float)`.

11.8 itertools: איטרטורים קומבינטוריים ואינסופיים

itertools מספקת איטרטורים lazy; הם לא ממשים את כל הרצף בזיכרון, אלא מייצרים איבר אחד בכל פעם. זה מאפשר עבודה עם רצפים אינסופיים או גדולים מאוד.

11.9 איטרטורים אינסופיים

repeat ,cycle ,count

```python
from itertools import count, cycle, repeat, islice

# count: infinite counter
for i in islice(count(start=10, step=2), 5):
    print(i)  # 10, 12, 14, 16, 18

# cycle: repeat a sequence infinitely
for c in islice(cycle("AB"), 6):
    print(c)  # A, B, A, B, A, B

# repeat: repeat one value (infinite or bounded)
print(list(repeat("hi", 3)))  # ['hi', 'hi', 'hi']
```

11

11.10 איטרטורים סופיים: קומבינציות, תמורות, מכפלה קרטזית

קומבינציות ותמורות

```python
from itertools import (
    combinations, permutations, product,
    combinations_with_replacement,
)

items = ["A", "B", "C"]

print(list(combinations(items, 2)))
# [('A', 'B'), ('A', 'C'), ('B', 'C')] -- order does not matter

print(list(permutations(items, 2)))
# [('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ...]

print(list(product(items, repeat=2)))
# cartesian product: [('A', 'A'), ('A', 'B'), ('A', 'C'), ...]

print(list(combinations_with_replacement(items, 2)))
# [('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ...]
```


11.11 איטרטורי זרמים: islice, takewhile, dropwhile, groupby

islice: פרוסת איטרטור

```python
from itertools import islice, count

# first 10 numbers divisible by 7
divisible_by_7 = (x for x in count(1) if x % 7 == 0)
print(list(islice(divisible_by_7, 10)))
# [7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
```

groupby: קיבוץ רציף

```
from itertools import groupby

data = [
    ("fruit", "apple"), ("fruit", "banana"),
    ("veg", "carrot"), ("fruit", "date"),
]
for key, group in groupby(data, key=lambda x: x[0]):
    print(key, list(group))
# fruit [('fruit', 'apple'), ('fruit', 'banana')]
# veg [('veg', 'carrot')]
# fruit [('fruit', 'date')]  -- note: only adjacent runs!
```


הערה 11.2. `groupby` מקבץ רק ערכים **סמוכים**. אם צריך לקבץ את כל הערכים לפי מפתח; צריך למיין קודם: `sorted(data, key=...)` ואז `groupby`.

11.12 starmap, accumulate, chain, zip_longest

כלים שימושיים נוספים

```python
from itertools import starmap, accumulate, chain, zip_longest
import operator

# starmap: call a function with tuple unpacking
pairs = [(1, 2), (3, 4), (5, 6)]
print(list(starmap(operator.add, pairs)))  # [3, 7, 11]

# accumulate: running totals (or any binary operator)
print(list(accumulate([1, 2, 3, 4])))               # [1, 3, 6, 10]
print(list(accumulate([1, 2, 3, 4], operator.mul))) # [1, 2, 6, 24]

# chain: concatenate iterators without materializing
print(list(chain([1, 2], "ab", (3, 4))))  # [1, 2, 'a', 'b', 3, 4]

# zip_longest: zip until the longest is exhausted
# default fillvalue is None
print(list(zip_longest([1, 2], "abc")))
# [(1, 'a'), (2, 'b'), (None, 'c')]

# with an explicit fillvalue
print(list(zip_longest([1, 2], "abc", fillvalue="?")))
# [(1, 'a'), (2, 'b'), ('?', 'c')]
```


הערה 11.3. שני תוספות מודרניות ששווה להכיר: `itertools.pairwise` (3.10+) מחזיר זוגות עוקבים – `[(1, 2), (2, 3), (3, 4)]` – תחליף נקי ללולאת `for i in range(len(seq) - 1)`; ו־`itertools.batched` (3.12+) מחלק איטרטור לגושים של גודל קבוע, במקום לרשום ידנית slices. שניהם lazy ומחסלים לולאות אצבעות נפוצות.

11.13 ביטויי גנרטור (Generator Expressions) והערכה עצלה

ביטוי גנרטור `(x for x in items)` דומה ל־`[x for x in items]` אבל **לא**

ממשם את הרשימה; מחזיר איטרטור שמייצר ערכים אחד אחד.

השוואת זיכרון: רשימה מול גנרטור

```python
import sys

# list - the whole million in memory
million_list = [x * 2 for x in range(1_000_000)]
print(sys.getsizeof(million_list))  # ~8 MB

# generator - just the iterator object
million_gen = (x * 2 for x in range(1_000_000))
print(sys.getsizeof(million_gen))   # ~200 bytes
```



```
8448728
200
```

מתי גנרטור, מתי רשימה?

- **גנרטור**: ברירת מחדל לרצפים גדולים/אינסופיים, עיבוד חד־פעמי, שרשור פעולות (`map`, `filter`, `itertools`).
- **רשימה**: כשצריך גישה באינדקס, אורך (`len`), איטרציה חוזרת, או העברה לפונקציה שדורשת sequence.
- `list(gen)`: ממיר גנרטור לרשימה כשצריך.

11.14 Anti-pattern → הדרך הטובה יותר

לא טוב: מימוש מוקדם של רצף גדול

```
# what NOT to do
def process_all(items):
    results = [
        expensive_transform(x) for x in items
    ]  # materializes everything in memory
    return [
        r for r in results if r.is_valid()
    ]  # another whole list

# if items is a million entries -> two giant lists in memory
```


טוב: צינור עצל (lazy pipeline)

```
from itertools import filterfalse

def process_all(items):
    # each step yields an iterator; nothing is
    # materialized until final consumption
    transformed = (expensive_transform(x) for x in items)
    valid = filterfalse(
        lambda r: not r.is_valid(), transformed,
    )
    return list(valid)  # only materialized here
```

לא טוב: לולאות מקוננות במקום product

```python
# what NOT to do
list_a = [1, 2, 3]
list_b = [4, 5]
list_c = [6]

def combine(a, b, c):
    return a * b * c

results = []
for a in list_a:
    for b in list_b:
        for c in list_c:
            results.append(combine(a, b, c))
```

טוב: product קריא ויעיל

```
from itertools import product

list_a = [1, 2, 3]
list_b = [4, 5]
list_c = [6]

def combine(a, b, c):
    return a * b * c

results = [
    combine(a, b, c)
    for a, b, c in product(list_a, list_b, list_c)
]
# or lazy:
results = (
    combine(a, b, c)
    for a, b, c in product(list_a, list_b, list_c)
)
```


11.15 תרגיל קצר

תרגיל: צינור עיבוד נתונים עצל

1. כתבו פונקציה `parse_log_lines(lines)` שמקבלת איטרטור של שורות לוג, ומחזירה איטרטור של מילונים מפורשים (השתמשו ב־`re` או `str.split`).
2. הוסיפו פילטר ששומר רק שורות ברמת `ERROR` או `CRITICAL`.
3. הוסיפו `groupby` לפי `logger_name` והדפיסו ספירה לכל לוגר.
4. (רמה שניה) השתמשו ב־`functools.partial` כדי ליצור גרסה קבועה של הפונקציה עם פילטר מוגדר מראש.

מבנה התחלתי

```
from itertools import groupby, filterfalse
import re

LOG_PATTERN = re.compile(
    r"(?P<level>\w+)\s+(?P<logger>\S+)\s+(?P<msg>.+)"
)

def parse_log_lines(lines):
    for line in lines:
        match = LOG_PATTERN.match(line)
        if match:
            yield match.groupdict()

# TODO: build the pipeline
```

פרק 12: קשירת שמות, סקופינג וסגירות

קשירת שמות (name binding) היא המנגנון שבו פייתון מקשרת בין מזהה (`x`, `my_func`) לבין אובייקט בזיכרון. הבנת LEGB (Local, Enclosing, Global, Built-in), סגירות (closures), וההבדל בין קשירה מוקדמת למאוחרת (late binding); חיונית כדי לכתוב דקורטורים, פונקציות פנימיות, ולהבין למה קוד שלכאורה תקין מתנהג בצורה מפתיעה. בפרק זה נפרק את המנגנון, נדגים את המלכודות הנפוצות, ונראה איך `functools.partial` ופרמטרים עם ערכי ברירת מחדל פותרים אותן.

12.1 חוק LEGB: סדר חיפוש שמות

כשפייתון נתקלת בשם, כמו `x`, היא מחפשת אותו בארבעה מרחבי שמות (namespaces) בסדר:

הגדרה 12.1 (כלל LEGB)**.**

1. **Local L:** מרחב השמות של הפונקציה הנוכחית (פרמטרים ומשתנים מקומיים).
2. **Enclosing E:** מרחבי השמות של פונקציות עוטפות (nested functions), מהפנימית לחיצונית.
3. **Global G:** מרחב השמות של המודול (`globals()`).

4. **Built-in B:** מרחב השמות המובנה (`builtins`, כמו `len`, `str`, `Exception`).
השם הראשון שנמצא; מנצח. אם לא נמצא באף אחד, נזרקת `NameError`.

דוגמת LEGB

```python
# Built-in
len = "shadowed"  # never do this!

# Global
counter = 0

def outer():
    # Enclosing
    counter = 10

    def inner():
        # Local
        counter = 100
        print(f"Local: {counter}")      # 100

    inner()
    print(f"Enclosing: {counter}")       # 10

outer()
print(f"Global: {counter}")              # 0
```

```
Local: 100
Enclosing: 10
Global: 0
```

הערה 12.1. הצללת (shadowing) שמות מובנים (`list`, `dict`, `type`, `len`) היא מקור נפוץ לבאגים עדינים. כלי בדיקה כמו ruff ו־pylint מתריעים על זה.

12.2 global ו־nonlocal: שינוי קשירות במרחבים חיצוניים

ברירת המחדל: השמה (`x = ...`) בתוך פונקציה יוצרת משתנה **מקומי** חדש. כדי לשנות משתנה במרחב חיצוני, צריך להצהיר על כך במפורש.

global: שינוי משתנה גלובלי

```
counter = 0

def increment():
    global counter  # tell Python: use the global counter
    counter += 1

increment()
print(counter)  # 1
```

nonlocal: שינוי משתנה ב־Enclosing

```
def make_counter():
    count = 0  # Enclosing scope

    def increment():
        nonlocal count  # modifies count of make_counter
        count += 1
        return count

    return increment

c1 = make_counter()
print(c1())  # 1
print(c1())  # 2

c2 = make_counter()  # new, separate counter
print(c2())  # 1
```

הבדל: global מול nonlocal

- `global x` במודול (`globals()`). עובד בכל מקום.
- `nonlocal x` במרחב Enclosing הקרוב ביותר (פונקציה עוטפת). **לא** עובד במודול גלובלי.
- שניהם משפיעים על **קשירה** (binding), לא על שינוי אובייקט קיים (`list.append` לא צריך אותם).

12.3 סגירות (Closures): פונקציות שזוכרות את הסביבה

Closure (סגירה) היא פונקציה פנימית שמתייחסת למשתנים מהמרחב Enclosing שלה; ואותם משתנים **נשמרים** כל עוד הסגירה קיימת, גם אחרי שהפונקציה החיצונית הסתיימה.

סגירה בסיסית

```python
def make_multiplier(factor: int):
    def multiply(x: int) -> int:
        return x * factor  # factor from the enclosing scope
    return multiply

times3 = make_multiplier(3)
times5 = make_multiplier(5)

print(times3(10))  # 30
print(times5(10))  # 50

# each closure keeps its own factor
print(times3.__closure__[0].cell_contents)  # 3
print(times5.__closure__[0].cell_contents)  # 5
```

הערה 12.2. המשתנים הנתפסים מאוחסנים ב־`__closure__`; טאפל של אובייקטי `cell`.

כל `cell` מכיל הפניה לאובייקט. זה למה שינוי אובייקט mutable שנתפס משפיע על כל הסגירות שתופסות אותו.

12.4 המלכודת: Late Binding בלולאות

הבאגים הנפוצים והמתסכלים בפייתון: פונקציות שנוצרו בלולאה לוכדות את **המשתנה עצמו**, לא את ערכו בעת היצירה.

לא טוב: קשירה מאוחרת בלולאה

```
# what NOT to do
functions = []
for i in range(3):
    def get_i():
        return i
    functions.append(get_i)

print([f() for f in functions])  # [2, 2, 2] -- not [0, 1, 2]!
```

```
[2, 2, 2]
```

הערה 12.3. כל הפונקציות מצביעות על **אותו** משתנה `i` מהמרחב הגלובלי/העוטף. כשהלולאה מסתיימת, `i` שווה 2. כל הקריאות קוראות את הערך **הנוכחי** של `i`; לא את הערך שהיה בזמן הגדרת הפונקציה.

12.5 פתרון 1: ערכי ברירת מחדל (Default Arguments)

פרמטרים עם ערכי ברירת מחדל מחושבים **פעם אחת** בזמן הגדרת הפונקציה; לא בזמן הקריאה. זה יוצר קשירה מוקדמת (early binding).

טוב: לכידת ערך עם ברירת מחדל

```
functions = []
for i in range(3):
    def get_i(captured=i):  # i evaluated now, kept as default
        return captured
    functions.append(get_i)

print([f() for f in functions])  # [0, 1, 2]
```

```
[0, 1, 2]
```


הערה 12.4. זה עובד כי `captured=i` מחושב בזמן `def` (כל איטרציה יוצרת פונקציה חדשה עם `captured` שונה). החיסרון: החתימה משתנה (`get_i(captured=0)`), וזה עלול לבלבל משתמשים.

12.6 פתרון 2: functools.partial; קשירה מפורשת

`partial` יוצר פונקציה חדשה עם ארגומנטים קבועים מראש; קריא יותר ומשמר חתימה נקייה.

partial לפתרון binding late

```
from functools import partial

functions = []
for i in range(3):
    def get_value(captured):
        return captured
    functions.append(partial(get_value, i))  # i is bound now

print([f() for f in functions])  # [0, 1, 2]
```

partial עם פרמטר שנוצר בלולאה

```
from functools import partial

def make_handlers():
    handlers = []
    for i in range(3):
        def handler(captured, value):
            return f"handler {captured} -> {value}"
        handlers.append(partial(handler, i))  # captured bound now
    return handlers

for h in make_handlers():
    print(h(5))
# handler 0 -> 5
# handler 1 -> 5
# handler 2 -> 5
```


הערה 12.5. תראו לפעמים partial משמש גם כdecorator כדי ליצור פונקציה עם ארגומנטים קבועים מראש – זה עובד, אבל הוא מפתיע לקריאה. קשירה בתוך לולאה (כמו בשתי הדוגמאות למעלה) או פונקציית factory (פתרון 3) ברורות יותר. אם כבר לוקחים partial, חשוב לדעת: הוא שומר על החתימה של הפונקציה המקורית – בניגוד לפתרון ברירת המחדל, שמשנה אותה.

12.7 פתרון 3: פונקציה חיצונית (Factory Function)

הפתרון הקלאסי והקריא ביותר: פונקציה שמחזירה פונקציה, עם הפרמטר כארגומנט רגיל.

Factory function: קריא וברור

```
def make_getter(captured):
    def getter():
        return captured
    return getter

functions = [make_getter(i) for i in range(3)]
print([f() for f in functions])  # [0, 1, 2]
```

השוואת פתרונות ל-Late Binding

שיטה	קריאות	חתימה נשמרת	ביצועים
ברירת מחדל (captured=i)	בינונית	לא	מצוין
`partial`	טובה	כן	טוב
function Factory	מצוינת	כן	מצוין

המלצה: **function Factory** ברוב המקרים; `partial` כשצריך לשמור חתימה ספציפית לכלים אוטומטיים.

12.8 סגירות ומשתנים Mutable: מלכודת משותפת

כשסגירה לוכדת אובייקט mutable (רשימה, מילון), כל הסגירות חולקות את אותו אובייקט.

מלכודת mutable משותף

```
def make_adders():
    adders = []
    shared_list = []  # one shared object!
    for i in range(3):
        def adder(x):
            shared_list.append(x)  # all append to one list
            return sum(shared_list)
        adders.append(adder)
    return adders

a, b, c = make_adders()
print(a(10))  # 10
print(b(20))  # 30 (10 + 20) -- not 20!
print(c(30))  # 60 (10 + 20 + 30)
```


פתרון: ליצור אובייקט חדש בכל איטרציה

```
def make_adders_fixed():
    def make_adder():
        local_list = []  # new each iteration
        def adder(x):
            local_list.append(x)
            return sum(local_list)
        return adder
    return [make_adder() for _ in range(3)]

a, b, c = make_adders_fixed()
print(a(10))  # 10
print(b(20))  # 20
print(c(30))  # 30
```

12.9 else בלולאות וב־try: הפנינה הנסתרת

לפייתון יש `else` בלולאות `while`/`for` וב־`try`; הוא רץ **רק אם לא היה `break` (בלולאה) או חריגה (ב-`try`)**.

else ב-for -- חיפוש בלי דגל

```
def find_first_even(numbers):
    for n in numbers:
        if n % 2 == 0:
            return n
    else:
        # runs only if the loop ended without break/return
        raise ValueError("No even number found")

print(find_first_even([1, 3, 5, 8, 9]))  # 8
# find_first_even([1, 3, 5])  # ValueError
```

else ב-try -- ניקוי רק בהצלחה

```
def read_config(path):
    try:
        with open(path) as f:
            data = f.read()
    except OSError as e:
        raise ConfigError(f"Cannot read {path}") from e
    else:
        # runs only if no exception occurred
        return parse_config(data)
    # finally would run in any case
```

מתי להשתמש ב-else בלולאה?

כשמחפשים משהו ורוצים להבדיל בין "נמצא" ל-"לא נמצא" בלי משתנה דגל (`found = False`). הקוד נעשה קריא יותר והכוונה מפורשת.

12.10 Anti-pattern → הדרך הטובה יותר

לא טוב: דגל ידני לחיפוש

```
# what NOT to do
def find_user(users, target_id):
    found = None
    for u in users:
        if u.id == target_id:
            found = u
            break
    if found is None:
        raise NotFoundError(target_id)
    return found
```


טוב: for/else קריא ומפורש

```
# good: for/else
def find_user(users, target_id):
    for u in users:
        if u.id == target_id:
            return u
    else:
        raise NotFoundError(target_id)
```

לא טוב: למבדה בלולאה בלי לכידה

```
# what NOT to do
buttons = []
for i in range(3):
    btn = Button(f"Btn {i}")
    btn.on_click = lambda: print(f"Clicked {i}")  # prints 2!
    buttons.append(btn)
```

טוב: function factory או partial

```
class Counter:
    def __init__(self):
        self._count = 0

    def increment(self):
        self._count += 1
        return self._count

    def reset(self):
        self._count = 0

# or with closure + nonlocal:
def make_counter():
    count = 0
    def increment():
        nonlocal count
        count += 1
        return count
    def reset():
        nonlocal count
        count = 0
    # how to return both? (hint: object, dict, or namedtuple)
    ...
```


12.11 תרגיל קצר

תרגיל: מונה עם מצב ושחזור

1. מימשו `make_counter()` שמחזיר פונקציה `increment()` הסופרת כלפי מעלה.
2. הוסיפו שיטה `reset()` שמאפסת את המונה (רמז: החזירו אובייקט עם מתודות, או השתמשו ב-`nonlocal` בפונקציה פנימית נוספת).
3. (רמה שניה) מימשו `make_lazy_sequence(generator_func)` שמקבלת פונקציה גנרטור ומחזירה אובייקט שמאפשר גישה באינדקס (`seq[5]`); מחשב רק את מה שצריך ושומר במטמון.

רמז לחלק 2

```
class Counter:
    def __init__(self):
        self._count = 0

    def increment(self):
        self._count += 1
        return self._count

    def reset(self):
        self._count = 0

# or with closure + nonlocal:
def make_counter():
    count = 0
    def increment():
        nonlocal count
        count += 1
        return count
    def reset():
        nonlocal count
        count = 0
    # how to return both? (hint: object, dict, or namedtuple)
    ...
```

פרק 13: מדריך בחירת אוספים

פייתון מציעה מגוון רחב של מבני נתונים מובנים ובספרייה הסטנדרטית; מרשימות ומילונים בסיסיים ועד `deque`, `defaultdict`, `Counter`, `namedtuple`, `dataclass`, `array`, `memoryview` וספריות חיצוניות כמו `numpy`, `pandas`, Cython. הבחירה הנכונה משפיעה על ביצועים, זיכרון, קריאות ותחזוקה.
בפרק זה נבנה מדריך החלטה מעשי: מתי כל מבנה מתאים, מה המחיר של כל בחירה, ואיך לעבור ביניהם כשהדרישות משתנות.

13.1 רשימה (list): ברירת המחדל לרצפים ששומרים על סדר

- **סיבוכיות**: גישה באינדקס `O(1)`, הוספה בסוף `O(1)` ממוצע, הוספה/מחיקה באמצע `O(n)`, חיפוש `O(n)`.
- **זיכרון**: מערך מצביעים לאובייקטים; תקורה של 8 בייט למצביע + האובייקטים עצמם.
- **מתי**: רצף ששומר על סדר ההכנסה וצריך גישה אקראית, איטרציה חוזרת, מיון או פריסה (`slice`).

רשימה -- שימושים נפוצים

```
# construction
items = [1, 2, 3]
items = list(range(100))

# editing
items.append(4)           # O(1) amortized
items.extend([5, 6])      # O(k)
items.insert(0, "first")  # O(n) -- shifts everything right
items.pop()               # O(1) from the end
items.pop(0)              # O(n) -- shifts everything left

# lookup and sorting
3 in items                # O(n)
items.index(3)  # O(n) -- raises ValueError if not found
items.sort()              # O(n log n) in place
sorted(items)             # O(n log n) -- returns a new list
```


13.2 collections.deque; תור דו־כיווני יעיל

deque (Double-Ended Queue) מותאם להוספה והסרה משני הקצוות ב־O(1) מובטח. מומש כרשימה מקושרת של בלוקים.

deque -- תור וסטאק יעילים

```python
from collections import deque

# queue (FIFO)
q = deque([1, 2, 3])
q.append(4)        # right side
q.popleft()        # 1 -- left side, O(1)

# stack (LIFO)
s = deque()
s.append(1)
s.append(2)
s.pop()            # 2 -- right side, O(1)

# size limit (drops items from the other end automatically)
recent = deque(maxlen=3)
recent.append(1)
recent.append(2)
recent.append(3)
recent.append(4)   # deque([2, 3, 4], maxlen=3)
```


list מול deque

פעולה	list	deque
pop/append (ימין)	O(1)	O(1)
pop(0)/insert(0) (שמאל)	O(n)	**O(1)**
גישה באינדקס [i]	**O(1)**	O(1)*
איטרציה	מהיר	מעט איטי יותר
זיכרון לפריט	נמוך יותר	גבוה יותר (מצביעים לבלוקים)

* deque תומך באינדקס אבל זה O(1) רק בקצוות, O(n) באמצע.

13.3 collections.defaultdict; מילון עם ערכי ברירת מחדל

מונע KeyError ומפשט קוד שצובר ערכים.

defaultdict -- קיבוץ וספירה

```python
from collections import defaultdict

# group values by key
data = [
    ("fruit", "apple"), ("fruit", "banana"),
    ("veg", "carrot"),
]
grouped = defaultdict(list)
for category, item in data:
    grouped[category].append(item)

print(dict(grouped))
# {'fruit': ['apple', 'banana'], 'veg': ['carrot']}

# counting with defaultdict(int)
counter = defaultdict(int)
for word in ["apple", "banana", "apple", "cherry"]:
    counter[word] += 1
```

הערה 13.1. `defaultdict` יוצר ערך ברירת מחדל **רק בגישה** (`d[key]` או `d.get(key)`). `key in d` לא יוצר ערך. הפונקציה שמספקים נקראת בלי ארגומנטים (`list`, `int`, `set`, `lambda: []`).

13.4 collections.Counter; Multiset

תת־מחלקה של `dict` לספירת אובייקטים hashable. מספקת אריתמטיקה נוחה.

Counter -- ספירה, חיתוך, איחוד

```python
from collections import Counter

c1 = Counter("abracadabra")
c2 = Counter("alacazam")

print(c1)  # Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
print(c1.most_common(3))     # [('a', 5), ('b', 2), ('r', 2)]

print(c1 + c2)   # union (sum of counts)
print(c1 - c2)   # subtraction (removes counts ≤ 0)
print(c1 & c2)   # intersection (minimum counts)
print(c1 | c2)   # union (maximum counts)

# in-place update
c1.update("hello")
c1.subtract("world")
```


13.5 dataclass / namedtuple / tuple; רשומות Immutable

dataclass namedtuple, tuple, :השוואה

```python
from collections import namedtuple
from dataclasses import dataclass

# plain tuple -- position only
PointT = tuple[float, float]
p1: PointT = (1.0, 2.0)
print(p1[0])  # 1.0 -- not readable

# namedtuple -- field names, immutable, lightweight
PointNT = namedtuple("PointNT", ["x", "y"])
p2 = PointNT(1.0, 2.0)
print(p2.x, p2.y)  # 1.0 2.0
print(p2[0])       # 1.0 -- still works

# dataclass -- full-featured, mutable by default
@dataclass(slots=True)
class PointDC:
    x: float
    y: float
    label: str = "origin"

p3 = PointDC(1.0, 2.0)
print(p3.x, p3.label)  # 1.0 origin
p3.x = 5.0             # mutable
```

מתי כל אחד?

- ◇ `tuple`: זוגות/שלישיות קצרות, החזרה מרובה מפונקציה, מפתחות מילון.
- ◇ `namedtuple`: רשומה immutable קלה, תאימות לאחור עם קוד שמצפה לטאפל, ביצועים קריטיים (אין `__dict__`).
- ◇ `dataclass(slots=True)`: ברירת מחדל לרשומות מודרניות: טיפוסים, ברירות מחדל, מתודות מותאמות, slots לחיסכון בזיכרון.
- ◇ `dataclass(frozen=True)`: כשצריך immutable עם תחביר `dataclass`.

13

13.6 array.array ו־memoryview; מערכים מספריים דחוסים

`array` שומר ערכים פרימיטיביים (`int`, `float`) ברצף זיכרון רציף; בלי תקורת אובייקטים. `memoryview` מאפשר גישה לבאפר בלי העתקה.

array -- חיסכון בזיכרון למספרים

```
import array
import sys

# list of a million ints
py_list = [i for i in range(1_000_000)]
# array of signed long (usually 8 bytes)
arr = array.array("q", range(1_000_000))

print(sys.getsizeof(py_list))      # ~8 MB (pointers only)
print(sys.getsizeof(arr))  # ~8 MB (actual data)
# but: py_list holds separate int objects (~28 MB extra)
# arr stores the numbers directly -- only ~8 MB total
```

```
8448728
8183816
```

memoryview -- גישת באפר אפס-העתק

```python
import array

arr = array.array("d", [1.0, 2.0, 3.0, 4.0, 5.0])
mv = memoryview(arr)

# slicing without copying
sub = mv[1:4]  # memoryview of the same memory
print(list(sub))  # [2.0, 3.0, 4.0]

# convert to bytes for I/O
data = mv.tobytes()
print(len(data))  # 40 bytes (5 * 8)
```


הערה 13.2. קודי טיפוס array: `'B'/'b'` (signed/unsigned char), `'H'/'h'` (short), `'I'/'i'` (int), `'L'/'l'` (long), `'Q'/'q'` (long long), `'f'` (float), `'d'` (double). גודל תלוי בפלטפורמה; השתמשו ב־`array.array('q')` לint64 מובטח.

13.7 numpy, pandas, Cython, PyPy; מתי לצאת מהסטנדרט

מדריך החלטה: Python Pure מול כלים כבדים

תרחיש	המלצה	סיבה
עיבוד טבלאי קטן (<100K שורות)	`dataclass` / `list[dict]`	אין תלות, קריא
עיבוד טבלאי בינוני (100K-10M)	pandas	SQL-like, אופטימיזציות C
חישובים מספריים כבדים	numpy	וקטוריזציה, SIMD BLAS
לולאות צפופות ב-Python Pure	numba / Cython	קומפילציה ל-C, טיפוסים סטטיים
יישום ארוך-חי עם הרבה אובייקטים	PyPy	JIT, מהיר פי 2-10 ב-CPython
הטמעה במערכת C/C++	`cffi` / `ctypes` / Cython	תואם ABI C

numpy -- פעולות וקטוריות

```python
import numpy as np

# Pure Python: slow loop
py_list = [i * 2 for i in range(1_000_000)]

# NumPy: one vectorized operation (in C)
np_arr = np.arange(1_000_000, dtype=np.int64)
result = np_arr * 2  # runs in C, checks bounds, SIMD

# performance comparison (orders of magnitude)
# %timeit [i*2 for i in range(1_000_000)]  # ~40 ms
# %timeit np.arange(1_000_000) * 2          # ~2 ms
```

Cython -- טיפוסים סטטיים למהירות

```
# file: compute.pyx
# cython: language_level=3
def sum_squares(int n):
    cdef int i
    cdef long long total = 0
    for i in range(n):
        total += i * i
    return total

# compile: cythonize -i compute.pyx
# usage: from compute import sum_squares
# 50-100x faster than Pure Python
```


13.8 טבלת החלטה מהירה

איזה מבנה לבחור?

צריך...	השתמש ב...
רצף שומר־סדר, גישה אקראית, מיון	`list`
תור/סטאק יעיל משני הצדדים	`deque`
מילון עם ערכי ברירת מחדל	`defaultdict`
ספירת פריטים / Multiset	`Counter`
רשומה immutable קלה	`namedtuple`
רשומה מודרנית עם טיפוסים	`dataclass(slots=True)`
מערך מספרי דחוס (Pure Python)	`array.array`
גישת באפר אפס-העתק	`memoryview`
עיבוד טבלאי / SQL-like	pandas
חישובים מספריים וקטוריים	numpy
לולאות צפופות מהירות	numba / Cython
יישום ארוך-חי, הרבה אובייקטים	PyPy

13.9 Anti-pattern → הדרך הטובה יותר

לא טוב: רשימה כתור

```
# what NOT to do
queue = []
queue.append(1)
queue.append(2)
item = queue.pop(0)  # O(n) -- shifts the whole list!
```

טוב: deque לתור

```python
from collections import deque

queue = deque()
queue.append(1)
queue.append(2)
item = queue.popleft()  # O(1)
```

לא טוב: ספירה ידנית במילון

```python
# what NOT to do
counts = {}
for word in words:
    if word in counts:
        counts[word] += 1
    else:
        counts[word] = 1
```

טוב: defaultdict או Counter

```
from collections import deque, defaultdict, Counter
from dataclasses import dataclass
from datetime import datetime

@dataclass(slots=True)
class Order:
    order_id: str
    table_id: int
    items: list[tuple[str, int]]  # (dish_id, qty)
    status: str = "pending"
    created_at: datetime = datetime.now()

# TODO: implement the system
```


לא טוב: אובייקטים נפרדים למספרים

```
# what NOT to do
points = [(x, y) for x in range(1000) for y in range(1000)]
# a million tuples + 2M int objects = lots of memory
```

טוב: array או numpy למערכים מספריים

```
import array
import numpy as np

# array: Pure Python, compact
coords = array.array("d", [0.0] * 2_000_000)  # 1M points * 2

# numpy: vectorized, fast
coords = np.zeros(2_000_000, dtype=np.float64)
```

13.10 תרגיל קצר

תרגיל: בחירת מבנה נתונים ליישום אמיתי

נתונה מערכת שמנהלת הזמנות במסעדה. לכל הזמנה: `order_id`, `table_id`, `items` (רשימת `(dish_id, qty)`), `status` (`pending`, `preparing`, `ready`, `served`), `created_at`.

1. איזה מבנה תבחרו לייצוג הזמנה בודדת? נמקו.
2. איך תאחסנו את כל ההזמנות הפעילות כדי לאפשר: הוספה מהירה, חיפוש לפי `order_id`, עיבוד לפי סדר הגעה (FIFO)?
3. איך תספרו כמה מנות מכל סוג הוזמנו היום?
4. (רמה שניה) אם המערכת צריכה לעבד 10,000 הזמנות בדקה; מה תשנו?

התחלה

```
from collections import deque, defaultdict, Counter
from dataclasses import dataclass
from datetime import datetime

@dataclass(slots=True)
class Order:
    order_id: str
    table_id: int
    items: list[tuple[str, int]]  # (dish_id, qty)
    status: str = "pending"
    created_at: datetime = datetime.now()

# TODO: implement the system
```

פרק 14: מכניקת ייבוא וטעינה עצלה

מערכת הייבוא של פייתון חזקה וגמישה; אבל גם מקור לבעיות ביצועים (זמן אתחול), תלויות מעגליות (circular imports) והפתעות בזמן ריצה. בפרק זה נבין איך `import` עובד בפועל, וכיצד `sys.modules` ו־`importlib` מאפשרים שליטה דינמית; ונראה טעינה עצלה (lazy imports) כפתרון לבעיות אמיתיות בפרוד.

נדגים גם ייבוא דינמי: `__import__` ו־`importlib.import_module` מקבלים שם מודול כמחרוזת; ונגיע ל-API היציב של `importlib.metadata` ו־`importlib.resources` ב־3.12+.

14.1 איך import עובד: sys.modules, __import__, ומציאת מודולים

כשכותבים `import foo`, פייתון מבצעת:

1. בודקת `sys.modules["foo"]`; אם קיים, מחזירה אותו (סינגלטון למודול).
2. מחפשת מודול בשם `foo` בנתיבי החיפוש (`sys.path`).
3. יוצרת אובייקט מודול חדש, מכניסה ל־`sys.modules`.
4. מריצה את קוד המודול (מבצעת הגדרות, ייבואים פנימיים).
5. מחזירה את המודול (או קושרת שמות עם `from ... import`).

sys.modules -- מטמון המודולים

```python
import sys

# first import -- loads and executes
import json
print("json" in sys.modules)  # True

# second import -- returns from cache instantly
import json as json2
print(json is json2)  # True -- exactly the same object

# can remove from cache (to force a reload)
del sys.modules["json"]
import json  # loads again
```


הערה 14.1. `sys.modules` הוא מילון גלובלי. מחיקת מפתח ממנו גורמת לטעינה מחדש בייבוא הבא; שימושי בבדיקות ובכלים לפיתוח. אבל היזהרו: מודולים אחרים שכבר ייבאו את המודול הישן ממשיכים להחזיק הפניה אליו.

14.2 תלויות מעגליות (Circular Imports) ולמה הן קורות

תלות מעגלית קורית כש `a.py` מייבא `b.py` ו־`b.py` מייבא `a.py` (ישירות או דרך שרשרת). בגלל שפייתון מכניסה מודול ל־`sys.modules` **לפני** הרצת הקוד שלו, המודול השני רואה מודול "חצי מוכן".

תלות מעגלית -- בעיה

```python
# a.py
print("a.py: start")
from b import something_from_b
print("a.py: end")

# b.py
print("b.py: start")
from a import something_from_a
print("b.py: end")

# run: python a.py
# a.py: start
# b.py: start
# a.py: start  (re-entered -- still half-built)
# ImportError: cannot import name 'something_from_b'
#              from partially initialized module 'b'
```


אסטרטגיות למניעת תלויות מעגליות

1. **מבנה מחדש**; הוציאו קוד משותף למודול שלישי `common.py`.
2. **ייבוא מקומי**; העבירו `import` לתוך הפונקציה שצריכה אותו (נטען רק בקריאה).
3. **ייבוא מותנה**; `if TYPE_CHECKING:` לטיפוסים בלבד (לא רץ בזמן ריצה).
4. **ABC / Protocol**; הגדירו חוזה במודול נפרד, מימושים מייבאים אותו (לא להפך).

14.3 `TYPE_CHECKING`; ייבואים רק לבדיקת טיפוסים

`typing.TYPE_CHECKING` הוא קבוע ששווה `True` רק בזמן ריצת כלי בדיקה (mypy, pyright), ו־`False` בזמן ריצה רגילה.

TYPE_CHECKING מונע ייבואים כבדים בזמן ריצה

```python
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    # loaded only by mypy/pyright -- not at runtime!
    from heavy_dependency import HeavyClass
    from .models import User

def process(
    user: "User",
) -> "HeavyClass":  # string = forward reference
    # at runtime: User and HeavyClass are not imported
    return HeavyClass(user.name)
```


הערה 14.2. מ־Python 3.7 אפשר להוסיף `from __future__ import annotations`, וכל ההערות הופכות למחרוזות (PEP 563): כלי טיפוס עדיין מבינים אותן, וזמן הריצה לא משלם על בנייתן. אל תסתמכו רק על זה: `TYPE_CHECKING` נשאר הדרך הנכונה כשצריך ייבוא של מודול שלם שרק כלי הבדיקה רואים. (ורמז לעתיד: ב־3.14 ההערות נדחות לפי PEP 649 במקום להפוך למחרוזות — ההמלצה לא משתנה.)

14.4 טעינה עצלה (Lazy Imports) עם importlib

טעינה עצלה דוחה את הייבוא עד לרגע שהאובייקט באמת ניגשים אליו. זה מקצר זמן אתחול משמעותית ביישומים גדולים.

(+3.7) importlib.util.LazyLoader

```
import importlib.util
import sys

spec = importlib.util.find_spec("heavy_module")
loader = importlib.util.LazyLoader(spec.loader)
spec.loader = loader
lazy_module = importlib.util.module_from_spec(spec)
sys.modules["heavy_module"] = lazy_module  # placeholder only
loader.exec_module(lazy_module)            # nothing loads yet

import heavy_module   # found in sys.modules -- still lazy
# the real module is imported on first attribute access:
heavy_module.some_function()  # actually loads here
```

דפוס מעשי: Import Lazy ידני פשוט

```python
# utils/lazy_import.py
import importlib
import sys
from types import ModuleType
from typing import Any

class LazyModule(ModuleType):
    def __init__(
        self, name: str, real_name: str | None = None,
    ):
        super().__init__(name)
        self._real_name = real_name or name
        self._module: ModuleType | None = None

    def _load(self) -> ModuleType:
        if self._module is None:
            self._module = importlib.import_module(
                self._real_name,
            )
            # update sys.modules for future imports
            sys.modules[self.__name__] = self._module
        return self._module

    def __getattr__(self, name: str) -> Any:
        return getattr(self._load(), name)

    def __dir__(self) -> list[str]:
        return dir(self._load())

# usage:
# my_lazy = LazyModule("my_lazy", "heavy_module")
# sys.modules["my_lazy"] = my_lazy
# from my_lazy import some_func  # loaded on demand
```

הערה 14.3. בפייתון 3.14 נוספה תמיכה ניסיונית ב־`-L` / `PYTHONLAZYIMPORTS` (PEP 1165) לטעינה עצלה אוטומטית של ייבואים ברמה עליונה. שווה מעקב — אבל הפתרון הידני למעלה עובד בכל גרסה, בלי תלות בניסיוני.

14.5 importlib.import_module; ייבוא דינמי בזמן ריצה

כששם המודול מגיע כמחרוזת (מקונפיג או מתוסף), `importlib.import_module` טוען אותו בזמן ריצה.

import_module -- טעינת תוספים דינמית

```
import importlib
from typing import Protocol

class Plugin(Protocol):
    name: str
    def execute(self, data: dict) -> dict: ...

def load_plugins(plugin_names: list[str]) -> list[Plugin]:
    plugins = []
    for name in plugin_names:
        try:
            module = importlib.import_module(f"plugins.{name}")
            # expect a class named PluginClass in the module
            plugin_class = getattr(module, "PluginClass")
            plugins.append(plugin_class())
        except (ImportError, AttributeError) as e:
            print(f"Failed to load plugin {name}: {e}")
    return plugins

# config: plugins = ["auth", "logging", "metrics"]
plugins = load_plugins(["auth", "logging", "metrics"])
for p in plugins:
    print(p.name, p.execute({}))
```


14.6 importlib.resources; גישה למשאבים בתוך חבילות

קריאת קבצי נתונים (SQL, תבניות, הגדרות) שארוזים בתוך החבילה; בלי נתיבי קבצים שבירים.

importlib.resources (+3.9 API יציב)

```python
from importlib.resources import files, as_file

# assume this structure:
# mypkg/
#   __init__.py
#   data/
#     queries.sql
#     config.json

# read as text (returns str)
sql = files("mypkg.data").joinpath("queries.sql").read_text(
    encoding="utf-8",
)

# read as bytes
config_bytes = files("mypkg.data").joinpath(
    "config.json",
).read_bytes()

# if a real file path is needed (for C libraries etc.)
with as_file(
    files("mypkg.data").joinpath("config.json"),
) as path:
    # path is a temporary pathlib.Path
    external_lib.load_config(str(path))
# the temporary file is deleted at the end of the block
```


הערה 14.4. `importlib.resources.files` (+3.9) מחליף את `open_text` הישן; הוא עובד גם עם `zip`, namespace packages וכל loader תואם.

14.7 importlib.metadata; חבילות מותקנות ו-metadata (+3.8)

קריאת גרסה, תלויות, נקודות כניסה (entry points); בלי pkg_resources הכבד.

importlib.metadata

```python
from importlib.metadata import version, metadata, entry_points

# current package version
print(version("httpx"))  # "0.28.1"

# full metadata (from pyproject.toml / setup.cfg)
meta = metadata("httpx")
print(meta["Summary"])     # "The next generation HTTP client."
print(meta.get_all("Requires-Dist"))  # list of dependencies

# entry points -- plugin discovery
eps = entry_points(group="myapp.plugins")
for ep in eps:
    print(ep.name, ep.value)  # auth = myapp.plugins...
    plugin_class = ep.load()  # loads the object
```


14.8 אתחול מהיר: טכניקות מעשיות

צ'קליסט לקיצור זמן אתחול

1. **העבירו ייבואים כבדים לפונקציות**; נטענים רק כשצריך.
2. **השתמשו ב-`TYPE_CHECKING`** לטיפוסים בלבד.
3. **הימנעו מקוד רץ ברמת מודול**; חישובים, חיבורי DB, קריאות רשת. עטפו בפונקציות `init()`.
4. **השתמשו ב-`importlib.util.LazyLoader`** לספריות ענק נדירות.
5. **בדקו עם `python -X importtime`**; מראה זמן טעינה לכל מודול.
6. **שקלו PyPy**; JIT מאיץ אתחול חוזר.

פרופיילינג זמן ייבוא

```
# python -X importtime -c "import myapp" 2>&1 | head -30
# example output:
# import time: self [us] | cumulative | imported package
# import time: 123  |  123 | _frozen_importlib
# import time: 456  |  579 | _frozen_importlib_external
# import time: 7890 | 8469 | numpy
# import time: 3456 | 11925 | pandas
# import time:       123 |      12048 | myapp
# ...
```

14.9 מבנה פרויקט להימנעות מתלויות מעגליות

מבנה מומלץ

```
myproject/
├── pyproject.toml
├── src/
│   └── myproject/
│       ├── __init__.py           # public imports only
│       ├── core/                 # core, no external deps
│       │   ├── models.py         # dataclasses, Protocols
│       │   ├── protocols.py      # Protocol definitions
│       │   └── exceptions.py     # exception hierarchy
│       ├── services/             # business logic
│       │   ├── __init__.py
│       │   ├── user_service.py   # imports from core only
│       │   └── order_service.py
│       ├── adapters/             # external implementations
│       │   ├── __init__.py
│       │   ├── postgres_repo.py  # imports core.protocols
│       │   └── http_client.py
│       └── plugins/              # plugins loaded at runtime
│           ├── __init__.py
│           └── loader.py         # importlib.import_module
└── tests/
```

הערה 14.5. הכלל: **תלויות זורמות פנימה**; `adapters` מייבאים `core`, `services` מייבאים `core`, `core` לא מייבא כלום מהפרויקט. זה מונע מעגלים ומאפשר DI נקי.

14.10 Anti-pattern → הדרך הטובה יותר

לא טוב: ייבוא כבד ברמת מודול

```
# what NOT to do -- app/__init__.py
import heavy_ml_library  # 500ms load time
import pandas as pd      # 200ms
import requests

# now every `import app` takes 700ms+ even if ML is not needed
```

טוב: ייבוא עצל בתוך פונקציה

```
# good: app/__init__.py -- clean and fast
__version__ = "1.0.0"

def get_ml_model():
    import heavy_ml_library  # loaded only when called
    return heavy_ml_library.load_model()

def get_dataframe():
    import pandas as pd
    return pd.DataFrame()
```

לא טוב: תלות מעגלית דרך מודלים

```python
# what NOT to do
# models/user.py
from services.email import send_email  # circular!

class User:
    def notify(self):
        send_email(self.email, "Hello")

# services/email.py
from models.user import User  # circular!

def send_email(to: str, body: str):
    ...
```

טוב: Protocol במודול נפרד

```python
# good: core/protocols.py
from typing import Protocol

class Notifier(Protocol):
    def send(self, to: str, body: str) -> None: ...

# models/user.py
from core.protocols import Notifier

class User:
    def __init__(self, notifier: Notifier):
        self.notifier = notifier
    def notify(self):
        self.notifier.send(self.email, "Hello")

# services/email.py
from core.protocols import Notifier

class EmailNotifier:
    def send(self, to: str, body: str) -> None:
        ...
```


14.11 תרגיל קצר

תרגיל: מערכת תוספים עם טעינה עצלה

1. צרו חבילה `myapp.plugins` עם שני תוספים: `logger` ו־`metrics`.
2. כל תוסף מגדיר מחלקה `Plugin` עם `name` ו־`setup(config)`.
3. מימשו `PluginLoader` ש:
   - ◇ מקבל רשימת שמות תוספים מקונפיג.
   - ◇ טוען אותם דינמית עם `importlib.import_module`.
   - ◇ תומך בטעינה עצלה: `get_plugin(name)` טוען רק בבקשה ראשונה.
4. (רמה שניה) הוסיפו `entry_points` ב-`pyproject.toml` כדי לאפשר תוספים חיצוניים.

מבנה התחלתי

```
# pyproject.toml
[project.entry-points."myapp.plugins"]
logger = "myapp.plugins.logger:LoggerPlugin"
metrics = "myapp.plugins.metrics:MetricsPlugin"

# myapp/plugins/loader.py
import importlib
from importlib.metadata import entry_points
from typing import Protocol

class Plugin(Protocol):
    name: str
    def setup(self, config: dict) -> None: ...

class PluginLoader:
    def __init__(self):
        self._plugins: dict[str, Plugin] = {}
        self._loaded: set[str] = set()

    def discover(self) -> list[str]:
        return [
            ep.name
            for ep in entry_points(group="myapp.plugins")
        ]

    def get_plugin(self, name: str) -> Plugin:
        if name in self._loaded:
            return self._plugins[name]
        # TODO: dynamic + lazy loading
        ...

    def setup_all(self, configs: dict[str, dict]) -> None:
        for name, config in configs.items():
            self.get_plugin(name).setup(config)
```

פרק 15: סיכום ופרויקט מסכם

פרק זה אוסף את העקרונות לכלל אחד ומציע פרויקט מסכם קטן שמחבר את רוב מה שלמדנו: Protocols, היררכיית חריגות, DI managers, context, בדיקות, אריזה ולוגינג — הכול בפרויקט אחד בר־הרצה. אם חיכיתם לרגע שבו כל החלקים מתחברים למשהו אחד שלם, הנה הוא. עשרת העקרונות הבאים הם גם checklist לפני כל דיפלוי — מסקנה של לילות רבים של תיקונים עד הבוקר. קחו אותם, השתמשו בהם, והעניקו לעצמכם כמה לילות שקטים.

15.1 עשרת העקרונות המרכזיים

אם לקחתם מהספר רק רשימה אחת, שזו תהיה. לא צריך לשנן אותה בעל פה: תישארו נאמנים לעיקרון; והפרטים יחזרו מעצמם. מדפיסים את הרשימה הזו בתחילת כל פרויקט חדש, ומוצאים בה בכל פעם נקודה ששכחו לבדוק. בדקו אותה מראש.

1. **טיפוסים הם כלי**: Protocols ו־TypedDict במקום Any והנחות.
2. **שגיאות מדברות**: היררכיית חריגות ייעודית, ExceptionGroup לעיבוד קבוצות.
3. **לוגים מובנים**: JSON עם IDs, correlation בלי `print`.
4. **משאבים מנקים את עצמם**: RAII ExitStack, managers, context.

5. **מקביליות נבחרת, לא נגררת**: threads/asyncio/processes לפי העבודה, timeouts תמיד.
6. **מדידה לפני ייעול**: cProfile, py-spy, tracemalloc.
7. **תלות בכיוון המופשט**: DI ידני, composition root יחיד.
8. **פרויקט שהוא חבילה**: src/ + pyproject.toml.
9. **בדיקות על התנהגות**: Fake (מימוש מדומה)s ו־contract tests, mock רק בגבולות.
10. **מוכן בפרוד**: health checks, graceful shutdown, feature flags.

15.2 פרויקט מסכם: שירות המרת מטבעות

הגיע הזמן להוכיח שהתיאוריה עובדת גם ביחד, לא רק כל פרק לבדו. הפרויקט המסכם (שירות המרת מטבעות) בנוי בדיוק כמו בעבודה: שכבות מופרדות, בדיקות על התנהגות, ולוג שמישהו אחר יוכל לקרוא בלי להרים טלפון. תזהו כל רכיב שתפגשו בקוד; זה כל החומר של הספר, בקומץ שורות.

Converter: שירות שממיר סכום בין מטבעות, לפי rate שמגיע מספק חיצוני.

Application-ו Domain

```python
# converter/domain.py
from dataclasses import dataclass

@dataclass(frozen=True)
class Money:
    amount: float
    currency: str

# converter/service.py
from typing import Protocol

from converter.domain import Money

class ConversionError(Exception):
    pass

class RateUnavailableError(ConversionError):
    pass

class RateProvider(Protocol):
    def rate(self, frm: str, to: str) -> float: ...

class Converter:
    def __init__(self, rates: RateProvider) -> None:
        self.rates = rates

    def convert(self, money: Money, to: str) -> Money:
        try:
            r = self.rates.rate(money.currency, to)
        except RateUnavailableError as exc:
            raise ConversionError("rate service down") from exc
        return Money(money.amount * r, to)
```

Infrastructure: ספק אמיתי וספק fake

```python
# converter/providers.py
from converter.service import RateUnavailableError

class HttpRateProvider:
    def __init__(self, session) -> None:
        self.session = session

    def rate(self, frm: str, to: str) -> float:
        resp = self.session.get(
            f"https://api.x.io/rate?frm={frm}&to={to}"
        )
        if not resp.is_success:
            raise RateUnavailableError(
                f"status {resp.status_code}"
            )
        return float(resp.json()["rate"])

class FakeRateProvider:
    def __init__(self) -> None:
        self.rates = {("usd", "usd"): 1.0, ("usd", "eur"): 0.9}

    def rate(self, frm: str, to: str) -> float:
        if (frm, to) not in self.rates:
            raise RateUnavailableError("pair unknown")
        return self.rates[(frm, to)]
```

קומפוזיציה, לוג ול־CLI

```python
# converter/domain.py
from dataclasses import dataclass

@dataclass(frozen=True)
class Money:
    amount: float
    currency: str

# converter/service.py
from typing import Protocol

from converter.domain import Money

class ConversionError(Exception):
    pass

class RateUnavailableError(ConversionError):
    pass

class RateProvider(Protocol):
    def rate(self, frm: str, to: str) -> float: ...

class Converter:
    def __init__(self, rates: RateProvider) -> None:
        self.rates = rates

    def convert(self, money: Money, to: str) -> Money:
        try:
            r = self.rates.rate(money.currency, to)
        except RateUnavailableError as exc:
            raise ConversionError("rate service down") from exc
        return Money(money.amount * r, to)
```

בדיקת contract על שני הספקים

```python
import pytest

from converter.domain import Money
from converter.providers import FakeRateProvider
from converter.service import Converter, RateUnavailableError

@pytest.fixture
def provider():
    return FakeRateProvider()

def test_provider_contract(provider) -> None:
    assert provider.rate("usd", "usd") == 1.0
    with pytest.raises(RateUnavailableError):
        provider.rate("zzz", "xxx")

def test_fake_satisfies_contract():
    test_provider_contract(FakeRateProvider())

def test_converter_roundtrip():
    conv = Converter(FakeRateProvider())
    out = conv.convert(Money(100, "usd"), "eur")
    assert out == Money(90.0, "eur")
```


מה הפרויקט מחבר

DI + Protocol (פרק 1, 6) • היררכיית חריגות עם wrap (פרק 2) • httpx.Client כ־context manager (פרק 3) • בדיקות contract (פרק 8) • pyproject.toml ו־entry point (פרק 7) • לוג מובנה (פרק 2, 9).

החלק החשוב: כל שכבה כאן ניתנת להחלפה בנפרד, בלי לשבור את השאר. זה לא קסם; זה מבנה. קסם זה מה שקורה כששוכחים את המבנה, ואז מנסים לתקן הכל בלילה שלפני הדיפלוי.

15.3 כלים מומלצים

אלה לא "הכלים הנכונים" היחידים בעולם; אלה כלים שעובדים בפרוד. החלפתם באחרים? מצוין, העיקר שתעשו את העבודה. רק אל תהפכו את בחירת הכלים לתחביב: להחליף linter שלוש פעמים בשבוע לא מומלץ.

משימה	כלי
טיפוסים	mypy, pyright
בדיקות	pytest, hypothesis, pytest-asyncio
לוג מובנה	structlog
observability	OpenTelemetry
אריזה	hatch, poetry, flit
איכות קוד	ruff, black
פרופיל	cProfile, py-spy, scalene

15.4 מה ללמוד הלאה

הספר נגמר; והעבודה שלכם רק מתחילה; ואלה חדשות מצוינות. אם הגעתם עד כאן, יש לכם בסיס שמרבית האנשים שקוראים לעצמם "מפתחים" לא חולמים עליו. הנה כיוונים מומלצים להמשך.

- אופטימיזציה עמוקה: Cython, Rust עם pyo3.
- typing מלא של קוד async, Self ב־Protocols ו־TypeIs להיצרות טיפוסים (מ־3.13).
- OpenTelemetry מלא: SDK משלך, exporters, sampling.
- serverless מול long-running: תבניות שונות לאותם עקרונות.
- ספרים מומלצים: Fluent Python, Architecture Patterns with Python, Robust Python.

הערה 15.1. קוד שמחזיק מעמד הוא לא קוד יפה; הוא קוד שקל לבטל, להחליף ולפרוס מחדש. טיפוסים, בדיקות ולוגים הם לא המטרה; הם הדרך לגלות מהר שטעיתם. מי שחושב שהקוד שלו מושלם, כנראה שעדיין לא דיפלוי בפרוד. בהצלחה.

נספח: כלים, קישורים ותבניות

נספח זה מרכז את כלי העבודה המומלצים, קישורי העמקה, ותבניות קוד מוכנות להעתקה לתוכן. חשבו עליו כעל ערכת העזרה הראשונה שפותחים כשתקועים, או רשימת קניות שסורקים לפני פרויקט חדש. הרשימה מתקצרת עם הזמן, לא כי זוכרים יותר, אלא כי הנספח זוכר בשבילכם.

16.1 רשימת כלים וספריות

כל הכלים בטבלה הם open source, חינמיים וטובים מספיק עבור חברות שבהן עובדים עשרות ומאות מהנדסים. אל תיתנו למישהו לשכנע אתכם שצריך "פתרון מסחרי" כדי לבנות קוד טוב. את הטיעון הזה שומעים בדרך כלל ממי שיש לו אינטרס למכור לכם.

כלי	תחום	שימוש עיקרי
mypy	טיפוסים	בדיקת טיפוסים סטטית
pyright	טיפוסים	בדיקת טיפוסים (מהיר)
pytest	בדיקות	fixtures, markers
hypothesis	בדיקות	property-based
pytest-asyncio	בדיקות	tests async
structlog	לוג	לוגינג מובנה
OpenTelemetry	obs	tracing+metrics
ruff	איכות	formatter+linter
hatch	אריזה	environments
poetry	אריזה	lock קפדני
py-spy	ביצועים	פרופיל תהליך חי
scalene	ביצועים	CPU+זיכרון

מינימום התקנה

```
$ pip install -e ".[dev]"
# adds: pytest hypothesis pytest-asyncio structlog
#       ruff mypy
```

שורה אחת, והארגז כולו מתקין את עצמו. ככה צריך להרגיש התחלה של פרויקט: לא רשימת הוראות ידניות, אלא צעד ראשון שקט.

16.2 קישורי העמקה

התיעוד הרשמי מרתיע בהתחלה; אבל הוא המקור היחיד שלא משקר. כשקוראים blog post ומתפעלים מטריק מגניב, לכו לאמת אותו מול התיעוד. תגלו כמה פעמים ה"טריק" הזה הוא בעצם הבנה שגויה; הפרוד משלם על טעויות כאלה.

- typing: docs.python.org/3/library/typing

- docs.python.org/3/reference ExceptionGroup:
- structlog.org :structlog
- docs.pytest.org pytest:
- hypothesis.readthedocs.io :hypothesis
- opentelemetry.io :Python OpenTelemetry
- mypy.readthedocs.io :mypy
- github.com/benfred/py-spy :py-spy
- github.com/plasma-umass/scalene :scalene
- hatch.pypa.io :hatch
- python-poetry.org :poetry
- flit.pypa.io :flit
- 12factor.net :12-factor
- cosmicpython.com :Architecture Patterns with Python

16.3 תבניות קוד שימושיות

הקוד כאן הוא חינם; אתם יכולים להעתיק אותו בלי טיפת רגשי אשם; וזו בדיוק המטרה. עדיף להתחיל מהקרקע הזו מאשר לשבת מול מסך ריק ולשאול "מה כותבים כאן". גם כותבים מנוסים שואלים את השאלה הזו; ההבדל הוא שלא כותבים את התשובה מהזיכרון.

pytest + ruff + pyproject.toml

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "myapp"
version = "0.1.0"
requires-python = "⩾3.11"
dependencies = ["structlog⩾24,<26"]

[project.optional-dependencies]
dev = ["pytest", "hypothesis", "pytest-asyncio",
       "structlog", "ruff", "mypy"]

[project.scripts]
myapp = "myapp.cli:main"

[tool.ruff]
line-length = 88
target-version = "py311"

[tool.pytest.ini_options]
markers = ["slow: long-running"]
```

conftest.py: fixtures משותפים

```python
import pytest

@pytest.fixture
def logger():
    return structlog.get_logger()

@pytest.fixture
def repo():
    return InMemoryRepo()
```

context manager ל־service

```python
from contextlib import contextmanager

@contextmanager
def service(name: str):
    svc = start(name)
    try:
        yield svc
    finally:
        stop(svc)
```

DI מינימלי ללא framework

```python
def build() -> App:
    db = create_pool(os.environ["DATABASE_URL"])
    rates = HttpRateProvider(httpx.Client())
    return App(
        repo=PostgresRepo(db),
        converter=Converter(rates),
    )
```

צעד אחרון

פרקים 1–9 בכלים של נספח זה נותנים לכם קוד שproduction אוהב: טיפוסים שמגנים, שגיאות שמדברות, משאבים שמנקים את עצמם, בדיקות על התנהגות, ושירות שמוכן לכיבוי מנומס. צאו וכתבו דברים שיחזיקו מעמד. כשנתקלים בתקלה לא ברורה, נשמו עמוק, קראו את הלוג, והמשיכו הלאה.

מילון מונחים

Acronyms

Protocol מבנה המגדיר חוזה מבני ללא ירושה מפורשת (structural subtyping). מחלקה המקיימת את החוזה נחשבת תת-טיפוס אוטומטית

TypedDict טיפוס לתיאור מילונים עם מפתחות וערכים מוגדרים מראש, לבדיקת טיפוסים סטטית. בזמן ריצה נשאר dict רגיל

TypeVar משתנה טיפוס המאפשר פולימורפיזם פרמטרי. שומר על הטיפוס הקונקרטי לאורך פעולות

Generic מחלקה פרמטרית בטיפוס אחד או יותר. מאפשרת קוד לשימוש חוזר עם בטיחות טיפוסים

ParamSpec משתנה טיפוס ללכידת חתימת פונקציה (פרמטרים והחזרה). שימושי בדקורטורים ובטיפוסי callable

Subtyping Structural התאמת טיפוסים לפי מבנה (שיטות ושדות) ולא לפי שם או ירושה. הבסיס של Protocol בפייתון

Subtyping Nominal התאמת טיפוסים לפי שם והצהרת ירושה מפורשת. הנהוג ב-Java, C# C++,

ExceptionGroup מיכל לחריגות מרובות (Python 3.11+). מאפשר איסוף וטיפול בכמה כשלונות בו-זמנית עם except*
except* תחביר לטיפול סלקטיבי בחריגות בתוך ExceptionGroup (Python 3.11+). כל except* תופס תת-קבוצה

Logging Structured לוגינג בפורמט מובנה (JSON) במקום מחרוזות חופשיות. מאפשר שאילתות, סינון וניתוח אוטומטי בכלים כמו CloudWatch Loki, ELK,

ID Correlation מזהה ייחודי העובר עם בקשה אחת בכל המערכת. מאפשר מעקב אחר בקשה דרך שירותים רבים

Hierarchy Exception מבנה ירושה של חריגות ייעודיות לכל דומיין. מאפשר תפיסה סלקטיבית לפי רמת חומרה או הקשר

asyncio ספריית הסטנדרט לתכנות אסינכרוני בפייתון. מבוססת על coroutines loop, event ו-tasks

Coroutine פונקציה הניתנת להשהייה וחידוש (מוגדרת עם async .(def יחידת הבסיס של asyncio

TaskGroup מנהל הקשר לקבוצת משימות אסינכרוניות (Python 3.11+). מבטיח ניקוי וסיום מסודר של כולן יחד

Loop Event הלולאה המרכזית המריצה קורוטינות, מטפלת ב-I/O ומנהלת טיימרים. יחידה אחת ל-thread

ThreadPoolExecutor מאגר threads לעבודה חוסמת (blocking I/O). אינו מזרז עבודה CPU-bound בגלל ה-GIL. משתלב עם asyncio דרך run_in_executor

ProcessPoolExecutor מאגר תהליכים לעבודה CPU-bound אמיתית. עוקף את ה-GIL אך עם תקורת תקשורת גבוהה יותר

Injection Dependency דפוס שבו תלויות מסופקות למחלקה מבחוץ במקום להיווצר בתוכה. משפר בדיקתיות והחלפת רכיבים

Architecture Clean ארכיטקטורה בשכבות (Domain, Application, Infrastructure) עם תלות כלפי פנים. העקרון: כלל התלות

Root Composition המקום היחיד באפליקציה שבו נוצרות ומורכבות כל התלויות. בדרך כלל ב-main או ב-factory מרכזי

Pattern Repository הפשטה המסתירה פרטי גישה לנתונים. מחזירה אובייקטי דומיין, לא מבני נתונים גולמיים

pytest מסגרת הבדיקות הסטנדרטית בפועל לפייתון. תומכת ב-fixtures, parametrize, plugins עשירים

Hypothesis ספריית property-based testing. מייצרת אוטומטית קלטים רבים למציאת מקרי קצה

Testing Property-Based בדיקות המציינות תכונות שתמיד נכונות, והכלי מייצר קלטים אקראיים לאימות

Test Contract בדיקה המבטיחה שמימושים שונים של אותו ממשק (Protocol/ABC) מקיימים את אותה התנהגות

Fake מימוש פועל אך פשוט של ממשק (למשל in-memory DB). עדיף על mock כי בודק התנהגות אמיתית

Observability יכולת להבין מצב פנימי של מערכת מהפלטים שלה (logs, metrics, traces). שלושת העמודים

OpenTelemetry סטנדרט פתוח לאיסוף טלמטריה (traces, metrics, logs). ספרייה אחת, ייצוא למערכות רבות

Check Health נקודת קצה הבודקת תקינות שירות (liveness/readiness). בשימוש Kubernetes ו-load balancers

Shutdown Graceful סיום מסודר של שירות: מפסיק לקבל בקשות חדשות, מסיים עיבוד נוכחי, משחרר משאבים

dataclass דקורטור היוצר מחלקות נתונים עם __init__, __repr__, __eq__ אוטומטיים (Python 3.7+)

Descriptor אובייקט המממש __get__, __set__, __delete__ לשליטה בגישה לתכונה. הבסיס של property, classmethod method

Metaclass מחלקה של מחלקה. שולטת ביצירת מחלקות. בשימוש ל-APIים דקלרטיביים, רישום אוטומטי, validation

MRO Method Resolution Order -- סדר חיפוש שיטות בירושה מרובה. פייתון משתמש באלגוריתם C3

__slots__ הגבלת תכונות מותרות במחלקה. חוסך זיכרון, מונע יצירת __dict__, מאיץ גישה לתכונות

Closure פונקציה הלוכדת משתנים מהסקופ המקיף שלה. נשארת נגישה גם אחרי שהסקופ החיצוני הסתיים

Decorator פונקציה המקבלת פונקציה/מחלקה ומחזירה חדשה. תחביר @decorator הוא sugar syntactic

Manager Context אובייקט עם __enter__ ו-__exit__ לניהול משאבים. בשימוש עם statement with

ExitStack מנהל הקשרים דינמי (contextlib). מאפשר הרשמה של callbacks cleanup בזמן ריצה

Generics מחלקות ופונקציות פרמטריות בטיפוס. ראו גם TypeVar, Generic

API Application Programming Interface

CLI Line Command Interface

CPU Central Processing Unit

Injection Dependency **DI**
Object Transfer Data **DTO**

Language Intermediate **IL**
Input/Output **I/O**

Notation Object JavaScript **JSON**

Used Recently Least **LRU**

Order Resolution Method **MRO**

Programming Object-Oriented **OOP**
Mapping Object-Relational **ORM**

Memory Access Random **RAM**
Transfer State Representational **REST**

Agreement Level Service **SLA**
segregation, Interface substitution, Liskov Open-closed, responsibility, Single **SOLID**
inversion Dependency
Language Query Structured **SQL**
Layer Sockets Secure **SSL**

Protocol Control Transmission **TCP**
Security Layer Transport **TLS**

Interface User **UI**
Locator Resource Unfiorm **URL**
Identfiier Unique Universally **UUID**

Interface Gateway Server Web **WSGI**

Language Markup Ain't YAML **YAML**