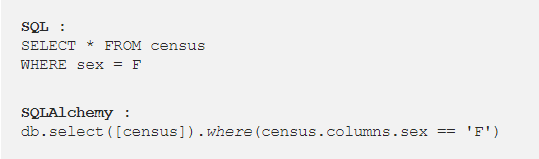
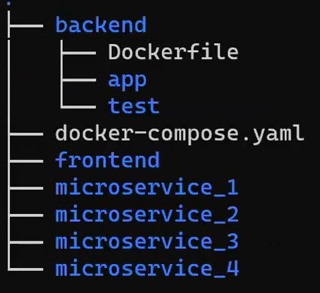
**EASS שיעור 9- SQL, DATACLASS, אלמנטים נוספים בפייתון, מבוא לdocker compose**

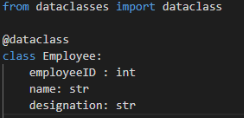
SQL (Structured Query Language)- שפה לניהול בסיסי נתונים יחסיים.  
מנועים שממשים את השפה- PostgreSQL, MySQL, SQLite.  
כדי לפנות לבסיס הנתונים נשלח למנוע בקשות HTTP שיכילו queries.  
SQLAlchemy- ספרייה (toolkit) בפייתון שמתרגמת פקודות בשפת פייתון לשאילתות ב-SQL, שאותן שולחת למנוע SQL ואת התשובה מתרגמת לפייתון. מתקשר עם כל אחד מהמנועים של SQL (שחלקם הוזכרו מעלה). דוגמה:  


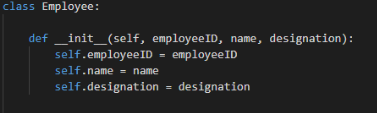
SQLite3- ספרייה בפייתון שמאפשרת עבודה עם קובץ DB ללא שרת נפרד (microservice במקרה שלנו(. <https://docs.python.org/3/library/sqlite3.html>

תצורת פרוייקט- כל מיקרוסרוויס הוא תיקייה, כמו גם ה-frontend וה-backend. לכל אחד מהם צריך ליצור קונטיינר משלו לכן בכל אחת מהתיקיות יהיה Dockerfile וגם טסטים. את הכל יחבר docker-compose שהוא מסוג .yaml (ניתן להיעזר באתרים לביצוע format לקובץ yaml כמו <https://jsonformatter.org/yaml-formatter> )  
  
בREADME יש להוסיף תמונה של דיארגמת בלוקים המתארת את כל חלקי הפרוייקט וכיצד הם מתקשרים (הייתה דוגמה בשיעור וגם בשיעור 7)

Dataclass:

Dataclass הוא class שמכוון לאכסון מידע, כלומר יותר פשוט ונוח למתכנת ליצור בו משתנים.

יצירת Dataclass דורשת ייבוא מהספרייה dataclasses. כל משתנה שניצור בו יהיה לו בנאי (\_\_init\_\_) אוטומטי:  


לעומת זאת, ב-class רגיל צריך מתודת בנאי \_\_init\_\_:  


Pydantic מזכיר בצורתו Dataclass.

Decorator:

מתודה שמקבלת מתודה אחרת, "עוטפת" אותה, מוסיפה לה פונקציונליות ומחזירה אותה.

נשתמש בסימן @ ואחריו שם ה-decorator כדי לגרום למתודה להיות עטופה על ידיו.

דוגמה:

>>> def make\_pretty(func): #This is the decorator

>>> def inner():

>>> print("I got decorated")

>>> func()

>>> return inner

>>> @make\_pretty

>>> def ordinary(): #This is a regular function that's going to be decorated

print("I am ordinary")

>>> ordinary()

I got decorated

I am ordinary

כל הפונקציות שהשתמשנו בהן ב-fastAPI עטופות ע"י decorators (@app.get וכו')

ניתן גם ליצור decorator מסוג class.

מתודת \_\_call\_\_: מתודה ב-class שמגדירה מה יקרה כשיקראו לו, לדוגמה MyClass()

Generator:

פונקציה שמחזירה אובייקט שניתן לבצע עליו איטרציות, כלומר אובייקט שניתן לקחת כל פעם ערך אחד שלו ולבצע עליו פעולה כלשהי. לדוגמה- לולאה על רשימה של 10 מספרים.

כל פונקציה שמכילה את הפעולה yield לפחות פעם אחת היא generator. yield היא פעולה כמו return , רק שהיא לא מסיימת את פעילות הפונקציה (כמו return) אלא רק עוצרת אותו ושומרת את המצב הקיים עד שניתן להמשיך.

דוגמה:

def my\_gen():

n = 1

print('This is printed first')

# Generator function contains yield statements

yield n

n += 1

print('This is printed second')

yield n

n += 1

print('This is printed at last')

yield n

for item in my\_gen():

print(item)

ומה שמוחזר:

This is printed first

1

This is printed second

2

This is printed at last

3

ניתן לראות את פעולת השמירה של my\_gen בין הדפסה להדפסה גם בעזרת פונקציית next בדוגמה הבאה:

>>> a = my\_gen()

>>> next(a)

This is printed first

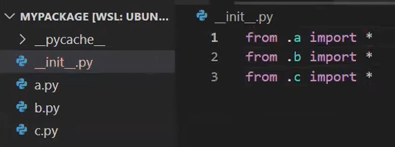
1

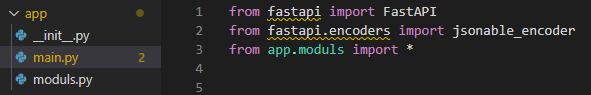
>>> next(a) #Value 1 is saved while generator is paused.

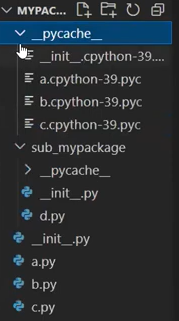
This is printed second

2

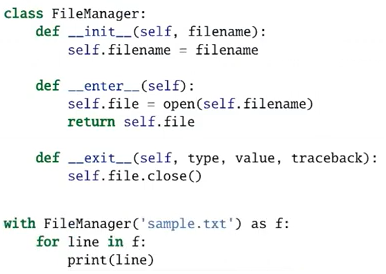
Package- מודל פייתוני שיכול להכיל מודלים אחרים ואף packages אחרים. את ה-package צריך להתקין לוקלית (עם pip install) ואז לייבא בעזרת import בקובץ פייתון שבו נרצה אותו.   
על מנת ליצור package עליו להכיל קובץ \_\_init\_\_.py , קובץ זה יכול להיות ריק או להכיל קוד. בכל תיקייה נוספת שניצור בתוך ה-package צריך להיות עוד קובץ \_\_init\_\_ .  
ניתן לייבא את פונקציונליות שיש בקבצי .py אחרים שכתבנו בpackage לקובץ ה\_\_init\_\_ (ניתן לעשות את זה באופן כללי- לייבא פונקציונליות מקובץ .py אחד לאחר) בצורה הבאה:

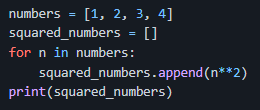
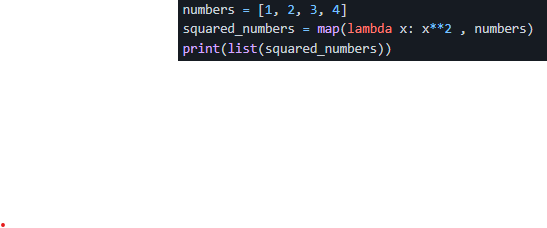
  
הכוכבית מסמנת שאנו רוצים לייבא את כל הפונקציונליות של הקובץ. ניתן לבחור גם משתנים, פונקציות או class ספציפיים במקום כוכבית.

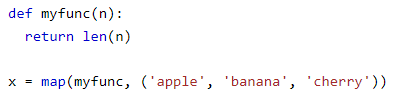
דוגמה נוספת:  
  
בקובץ moduls יצרנו את ה-classes של pydantic. לאחר מכן ייבאנו אותן לקובץ main כדי שנוכל להשתמש בהם. (ניתן גם היה לרשום .moduls במקום app.moduls מכיוון שהmain הוא גם בתיקיית app)

\_\_pycache\_\_- תיקייה שנוצרת שבה נשמרים דברים שנמצאים בcache. לדוגמה- אם נבצע import למשהו הוא יישמר שם כחלק משימוש אוטומטי יעיל בקוד.   
נרשום את תיקייה זו בקובץ gitignore כשנעלה את הקוד ל-github.  


Context manager- אמצעי לניהול משאבים כמו קבצים ו-DB בתוך הקוד. לדוגמה- ניתן באמצעותו לקרוא קובץ .txt בתוך הקוד. כאשר נממש את class FileManager חייב לממש בה את שתי הפונקציות: \_\_enter\_\_ ואת \_\_exit\_\_ .   
נשתמש במילה השמורה with בקריאה לclass הזה פונקציה \_\_enter\_\_ תופעל אוטומטית ובסיום השימוש פונקציה \_\_exit\_\_ תופעל אוטומטית.

דוגמה למימוש פשוט-  


תכנות פונקציונלי- סגנון כתיבת קוד בדומה לצורת כתיבת ביטוי מתמטי, כלומר הקוד מוצג כהרכבה של פונקציות. כדי ליישם זאת נשתמש בפונקציות high-order ו-Lambdas.  
פונקציות high-order הן פונקציות שמקבלות כקלט או מחזירות כפלט פונקציה אחרת (או פונקציות אחרות).  
פונקציית Lambda היא פונקציה אנונימית שמועברת כביטוי לפונקציה אחרת.  
דוגמה לבניית מערך של ערכים ריבועיים ללא תכנות פונקציונלי-  
  
ומימוש של הפעולה באמצעות תכנות פונקציונלי-  


Map- פונקציה שמורה שמקבלת פונקציה ומבנה איטרבילי (כמו רשימה, dictionary, מערך וכו'), ואז מבצעת את הפונקציה על כל אחד מהערכים במבנה האיטרבילי.  
  
ערכו של X בסוף הפעולה [5, 6, 6].

Pandas- ספרייה שנועדה לעיבוד וניתוח נתונים. מאפשרת עבודה עם מבני נתונים כמו dataframe (טבלה) ו-series (סדרה).

Docker compose- מאפשר חיבור של כמה microservices באמצעות קובץ אחד מסוג .yml

נבצע את הבנייה של הimage באמצעות פקודה docker-compose build  
(במקום docker build שעשינו עד כה)

נבצע את יצירת הקונטיינר באמצעות פקודה docker-compose up  
(במקום docker run שעשינו עד כה)

דוגמה לקובץ .yml :

version: "3.9" #The docker-compose version

services:

backend:

build: . # File location. One dot means the location is current file.

ports:

- "8888:80" #8888 is the port in localhost.

redis:

image: "redis:alpine"