קורס פיתון

Self.Py
shalomvanunu@mekifh.edum.org.il

	# סיכום
0.1למה ללמוד פייתון ?	שפת הפיתוח הנפוצה ביותר שפת קוד פתוח
0.2מבנה הקורס self.py	
1.1פייתון שפה עילית	בחירת השפה לכל פרויקט שפה עילית High Level, בחירה של רכיבים אך פחות שליטה בכל רכיב. שפה נמוכות קרובה יותר לשפת מכונה קומפלציה והידור Linking שפות הנכתבות בText ומועברות למכונה עי Interperter . מתרגמת לשפת מכונה. שפות עיליות Java, C#, Python
1.2 סביבת העבודה של פייתון	נוהל התקנת פיתון. לינק
1.3קדימה, לעבודה !	כתיבת היערות לשורת הקוד ע״י סימן # ספריות הקוד של פיתון שנכתבו. חלקן באות עם התקנת פיתון וחלקן ניתן לייבא עי הפקודה import. לדוגמא פקודה Cos Import math Math.cos(3.14) תיעוד ראשי של פיתון. לינק
2.1משתנים בפייתון - חלק א ׳	שמירת מידע ע״י משתנים variable_name = value כל משתנה מצביע על אובייקט, איזור בזיכרון של המחשב. טיפוסים שונים float,str,int
2.2משתנים בפייתון - חלק ב ׳	טיפוס Bool, יכול לקבל 2 ערכים True False כתיבת ביטוי לוגי מספר 0 מייצג את הערך הבולאני False אופרטורים נוספים : == שוויון =! אי שוויון אופרטור not קשר לוגי **גם וגם** AND קשר לוגי **או** OR קריאות הקוד עי ריווח המידע. אופרטור in בדיקת הכלה של מחרוזת בתוך מחרוזת

	פקודת help. תיעוד בלינק
2.3קלט משתמש ומניפולציות על משתנים	קליטת מידע מהמשתמש Input (‘ text on screen’) הזנת המידע לתוך משתנה Keyboard = input(‘text on screen’) חילוף בין שתי משתנים Var1 , Var2 = Var2, Var1
2.4 קונבנציות בשמות משתנים	קונבקצית כיתבה – סט מוסכם לרישום תקין. מסמך PEP המגדיר זאת . לינק Case_sensetive קובעת שני אובייקטים שונים המצביעים על ערך אחר הגדרת משתנה עי אותיות קטנות בלבד אם יותר ממילה אחת נדרש קו תחתון _ אותיות גדולות רק כאשר מדובר בקבוע בחירת שם המשתנה המשקף את המהות
3.1יצירת מחרוזות	מחרוזת str טיפוס נתונים המייצג תווים. מצויין ב ׳ ׳ או ״ ״ . באם יש סימן ׳ או ״ באמצע המחרוזת, באם המחרוזת מכילה ׳ נתחום ב ״ ״ ולהיפך באם קיימים הרבה גרשיים נמקם \. שירשור מחרוזות ע״י + אפשר להכפיל מחרוזת ע״י * הסבת משתנה למחרוזת ()str
3.2הדפסת מחרוזות	שיטות להדפיס מחרוזות ארוכות \n שורה חדשה \t הזחת שורה כדי לסמן לפיתון להשתמש ב \ כתו רצוי נסמן \\. סימון מחזורת עם שלוש מרכאות ״״״ מאפשר לשבור טקסט מבלי ׳ \n.
3.3גישה לתווים וחיתוך מקטעים במחרוזת	מיקום של תו מתחיל מהסיפרה אפס - 0 פניה למיקום ע״י [ ] סוגריים מרובעים ניתן לספור מציזו השני כלומר במינוס -2,-1 חיתוך מחרוזת string[start: stop] יש לשים לב ללא מיקום ערך מצביע stop חיתוך בקפיצות string[start: stop: step] שינוי ערך מחזורת ע״י שינוי הצבעה לערך חדש
3.4עבודה עם אובייקטים Targil3.4.1 Targil3.4.3	פונקציה ומתודה בלוק של קוד הממש פעולה כלשהי Function_name(parameter) ייבוא ספריה import math שימוש בפונקציה שורש ()math.sqrt מתודה היא פונקציה אשר מוגדרת עבור אובייקט מטיפוס מסוים, ומבצעת פעולות עליו או באמצעותו. Object.method_name(parameter) האובייקט הוא פרמט Built-in Functions String Method הצגת כל המתודות dir(str)
4.0עבודה עם עורך טקסט	מעבר מאינטרפרטר interpreter לעורך טקסט
4.1בלוקים	שורת פקודה המסתיימת ב :

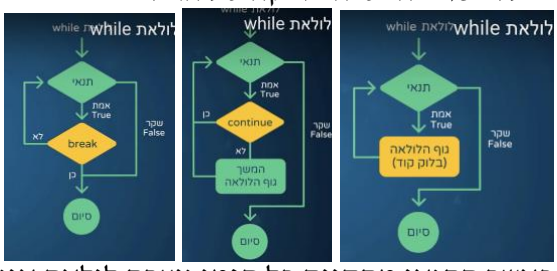
	אחכ רצף פקודות אשר נכתבות בהזחה בבלוק הוספת פקודת pass המאפשרת דילוג על הקוד

4.2בלוק תנאי
 	
תרגיל 4.2.1
 	
תרגיל 4.2.3
	פקודת תנאי if
	If ( ) :
	Command
	else:
	Command
	הגדרת תנאי ביניים elif
	מתבצע אם שום אחד מהמצבים לא התקיים
	If ( ) :
	Command
	elif:
	Command

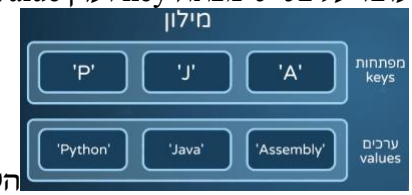
5.1פונקציות בפייתון	רצף של פקודות שאוספים אותם לבלוק אחד ומבצעים משימה מוגדרת def my_function(): def my_function(parameter) קריאה לפונקציה עם פרמטר My_function(argument)
5.2משתנים לוקאליים וגלובליים תרגיל 5.2.1	טווח ההכרה של משתנה Scope משתנה גלובלי – המוכר בתוכנית כולה משתנה לוקאלי- משתנה בתוך הפונקציה למשתנה לוקאלי יש עדיפות לא מומלץ להשתמש במשתנים גלובאליים בפונקציות שינוי ערך משתנה גלובלי בתוך פונקציה global anumal = “fish”
5.3פלט וקלט של פונקציה תרגיל 5.3.4 תרגיל 5.3.5 תרגיל 5.3.6	החזרת הערך של פונקציה ע״י מוגדר פלט פונקציה יכולה להחזיר יותר מערך אחד return value1,value2 ניתן להגדיר ארגומנטים ברירת מחדל My_function(argument=20)

5.4תיעוד, הפונקציה main
וקונבנציות כתיבה
	תיאור של פונקציה מתאר את המטרה והשימוש בה
	תיעוד הינו docstring
	שיטות תיעוד
	
	יצירת קובץ ספרייה משלנו
	Main()
	`def main():`
	`    code`
	
	`if __name__ == "__main__":`
	`    main()`
	
	הקפידו שלכל פונקציה תהיה מטרה אחת ברורה.
	נדאג ששם הפונקציה ישקף את המטרה שלה, וכך גם התיעוד שלה.
	שהקלט והפלט של הפונקציה יהיו ברורים והשימוש בהם יהיה אינטואיטיבי.
	בתוכנית עם יותר מפונקציה אחת, הפרידו ביניהן באמצעות שורה ריקה.

6.1 מהי רשימה?
 	
6.1.2 תרגיל
	רשימה היא מבנה נתונים
	סימון רשימה List = []
	רשימה יכולה להיות עם נתונים מסוגים שונים
	מיקום הנתונים מתחיל באפס 0
	ניתן להכניס רשימה בתוך רשימה ולגשת למיקום האיבר עי List[][]
	קיימת פונקציה list()
	הכנסת ערכים לרשימה עי הכנסת ערך מחרוזת
	
	דוגמה להשמת איברי רשימה למשתנים
	
	`>>> cool_trick = [1, 2, 3]`
	`>>> a, b, c = cool_trick`
	`>>> a`
	`1`
	`>>> c`
	`3`

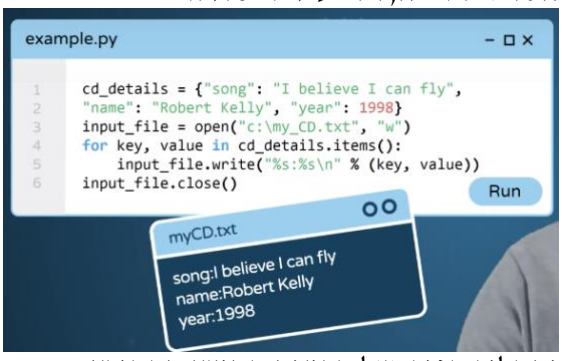
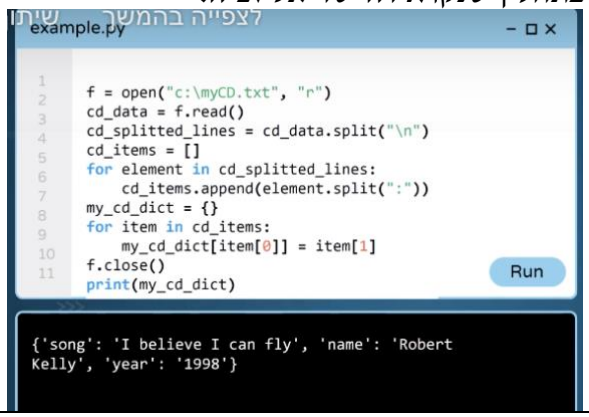
6.2 רשימות ומחרוזות הן רצפים 6.2.2 תרגיל 6.2.3 תרגיל	רשימות ומחרוזות מיקום ברשימה הינו איבר מיקום במחרוזת הוא תו אפשר לרשום מיקום חיובי ושלילי גם חיתוך Slicing עובד בשני המקרים הוספת איברים ברשימה יש לתחום ב [] ספירת איברים ע״י len
6.3 פעולות נפוצות על רשימות 6.3.1 תרגיל 6.3.2 תרגיל	בדיקת כמות איברים ברשימה len() הוספת איבר לרשימה list.append(“ “) או הוספת האופרטור list += [“new”] שינוי איבר מתבצע ע״י הכנסת ערך חדש למיקום list[] = “new” מחיקת ערך ברשימה list.remove(“item”) בדיקת ערך ברשימה עי פקודה in בדיקת כמות הפעמים לערך ברשימה list.count() מיון רשימה list.sort() מהקטן לגדול ערך הגדול ברשימה max(list) רשימה חדשה וממויינת sorted(list) ניתן למיין לפי מפתחות sorted(list , key=len)
7.1 לולאת while 7.1.3 תרגיל 7.1.4 תרגיל	כל עוד מתקיים התנאי, מתקיימת הפעולה בלולאה. הוא ייצא ממצב זה רק כאשר התנאי משתנה. While <condition>: Command בכל איטרציה ישנה בדיקה של התנאי  כאשר התנאי מתקיים כל הזמן נוצרת לולאה אינסופית . Break Continue פקודות

	Break מציבה תנאי נוסף ואם הוא מתקיים ישנה יציאה מהלולאה, באם לא תתבצע איטרציה נוספת Continue מציבה תנאי נוסף, באם מתקיים גורמת לגישה ישירות לאיטרציה הבאה מבלי להמשיך. באם לא ישנו המשך
7.2 לולאת for 7.2.1 תרגיל 7.2.4 תרגיל 7.2.5 תרגיל	זאת לולאה שמבצעת בלוק של קוד פעם אחת עבור כל איבר שמופיע ברצף כלשהו. For <var> in <sequence> <Do Something> פונקציית range() המייצרת סדרת מספרים for num in range(5) print (num) סדרה עם ערך התחלה וסיום range(start,stop,step) עם מרווח הקפיצה step
8.1 טאפל (tuple)	טיפוס למבנה נתונים בפיתון– Tuple רשימה כזו מסומנת ב (). My_tuple = () הפרדה בין הערכים עי , יצירת מערך Tuple גם עי הפרדה של , (פסיק) בלבד **יצירת Tuple עם ערך אחד** יהיה עי הוספת פסיק אחרי האיבר my_tuple = (20,) פעולות הדומות לרשימה Len(tuple) Tuple[0] טאפל tuple הינה רשימה שלא ניתנת לשינוי immutable . **לא ניתן להוסיף , לשנות ולמחוק** בניגוד לרשימה. אחרי יצירה יש אפשרות רק לגשת אליו. יתרונות Tuple : שמירת נתונים שלא אמורים להשתנות What is the difference between `sort()` `sorted()` ? The primary difference between the list `sort()` function and the `sorted()` function is that the `sort()` function will modify the list it is called on. The `sorted()` function will create a new list containing a sorted version of the list it is given.
8.2 דוגמאות לשימוש בטיפוס טאפל	**השמה בעזרת טאפל** tuple assignment (var1 ,var2 ,var3) = (value1 ,value2 , value3) **החזרת ערכים מפונקציה** return value1, value2 **יצירת תבנית למחרוזת** string formatting נסמן % עבור ערך של משתנה, ובסופו %(value1 value2) משתנה מחרוזת יסומן %s ומשתנה מסוג מספר int יסומן %d פורמט חדש לרישום נמצא בלינק
8.3 מילון	מילון Dict זהו מבנה נתונים בפיתון

[תרגיל 8.3.2](#) [תרגיל 8.3.3](#)	עובד על בסיס מפתח key וערךvalue המקושר אליה  הערכים מצויינים עי מפתחות.  הגדרת מילון ע״י { } ערכים במילון מורכבים ממפתח וערך המופרדים : הפרדה בין איברים עי , (פסיק) גישה למילון עי dict[key] **גישה הינה מערך למפתח בלבד** מילון הוא mutable כלומר **ניתן לשינוי** **הוספת** ערך dict_name[key] = value **עידכון** ערך מתבצע עי השמת ערך עם הפניה לאותו מפתח dict_name[key] = value **מחיקת** ערך עי del dict_name[key] או עי dict_name.pop(key) **הצגת** המפתחות dict_name.key() **הצגת** הערכים dict_name.values() **הצגת** המפתחות והערכים כצמד dict_name.items()

[עבודה עם קבצים בפייתון 9.1](#)
 	
[תרגיל 9.1.1](#)
 	
[תרגיל 9.1.2](#)
	עבודה עם קבצי txt
	החזרת אובייקט File_object = open(file,mode)
	fileהינו מחרוזת הגישה לקובץ
	mode מציג מצבי עבודה.
	R מצב Read קריאה
	w מצב write לכתיבה
	
	`input_file = open(r"c:\telephones.txt", "r")`
	כדי להימנע ממצב של הזחה t\ נדרש להוסיף r
	
	סגירת הקובץ File_object = close()
	
	ניתן להפעיל מנגנון קוד שאינו דורש סגירת הקובץ. [לינק](#)
	
	``with open(file,mode) as input_file:
	    #do something with the file;```
	
	קריאת תוכן הקובץ לתוך משתנה
	file_object = open(file,mode)
	file_content = file_object.read()
	הפרדת שורה תתבצע עי הסימון n\
	
	[מתודות נוספות לקריאת תוכן של קובץ](#)

[כתיבה לקבצים בפייתון 9.2](#) [תרגיל 9.2.2](#) [תרגיל 9.2.3](#)	החזרת אובייקט File_object = open(file,mode) fileהינו מחרוזת הגישה לקובץ mode מציג מצבי עבודה. w מצב write לכתיבה [סוגי גישה בפתיחת קבצים](#) שלב ראשון פתיחת הקובץ

	Chat_input_file = open(‘c:\chat.txt’,’w’) שלב שני כתיבה לקובץ Chat_input_file.write(‘the sentence to write’) שלב שלישי סגירת התוכנית Chat_input_file.close() כשפותחים קובץ בסוג גישה ,"w" פייתון דורסת את התוכן שלו. באם נכתוב טקסט חדש.
[9.3 תבניות של קבצים](#)	קיים פורמט לקבצים במבנה סדור כדיי לתרגם אובייקטים בפייתון לפורמט שאפשר לשמור בקובץ, ואפילו להעביר אותו ברשת מחשבים, נלמד על תהליך **הסריאליזציה**, או בעברית סדרות.  ותהליך הפוך, של קריאת המידע מהקובץ, בתהליך שנקרא **דה-סריאליזציה.** 

קורס פיתון

Next.Py

סיכום

כתיבת קוד בשורה קצרה

One Liner

רשימה של רצפים מאפשר לנו לבצע עיבוד
בפייתון קיימות פונקציות מובנות שמבצעות עיבוד ייחודי על רצפים
פונקציית map מאפשרת לנו לבצע את רצף הפעולות בלולאת בשורת קוד אחת בלבד.

```
result = map(function, sequence1, [sequence2,...])
```

האובייקט שהיא מחזיקה הינו איטרטור, לכן יש להציגה עם המרה

סינון מרשימה את כל האיברים הנדרשים ויצירת רשימה חדשה , וקיימת בפונקציה המובנית filter.

```
filter(filter_function, sequence)
```

Filter עוברת על כל איבר ברצף ומחזירה רצף חדש המורכב רק מהאיברים שפונקציית הסינון החזירה True.

```
# List of file names.
my_files = ["Dance remix.mp3", "work.docx", "lists.pdf",
"cheatsheet.pdf", "Shallow.mp3", "security-lesson.ppt", "The good
doctor.srt", "lecture slides.pptx", "It gets better (with time).mp3",
"exam.docx"]

# Filter mp3 files using the function "filter".
def is_mp3_file(filename):
    return filename.endswith(".mp3")

filter(is_mp3_file, my_files)
```

פונקציית reduce , הפעולה חוזרת על הרצף ומצמצמת אותו כדי ערך יחיד

```
import functools

functools.reduce(function, sequence [,initializer])
```

הפרמטר initializer מאפשר להגדיר ערך התחלתי לפונקציה
דוגמא בלינק

דוגמא לחישוב סכום כולל ברשימה:

```
import functools

def add(x, y):
    return x + y

shopping_list = [200, 120, 330, 50]
print(functools.reduce(add, shopping_list))
```

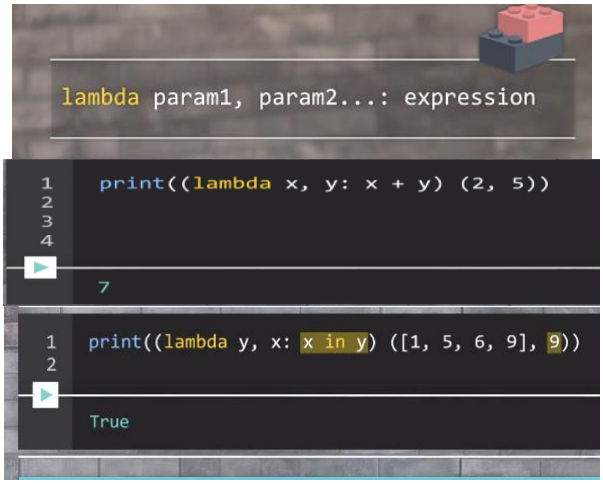
```
700
```

דוגמא למציאת הערך הקטן ברשימה:

```
import functools

def f(a, b):
    if a < b:
        return a
    else:
        return b

print(functools.reduce(f, [47, 11, 42, 102, 13]))
```

[1.2 פונקציות למדא](#)

פונקצית למדא lambda
מימוש פונקציה קצרה שאנחנו צופים שלא נשתמש בה שוב בתכנית.




```
print((lambda x, y: x + y) (2, 5))
```

```
7
```

```
print((lambda y, x: x in y) ([1, 5, 6, 9], 9))
```

```
True
```

ניתן להשתמש בפונקציית למדא lambda כארגומנט לפונקציה נוספת

```
my_list = [0, 1, 2, 3, 4, 5, 6, 7]
print(list(filter(lambda x: x % 2 == 0, my_list)))
```

עוד דוגמא הינה מיון רשימה עפ"י האיבר השני

```
list_of_tuples = [(2, 2), (3, 4), (4, 1), (1, 3)]
print(sorted(list_of_tuples, key=lambda x: x[1]))
```

```
[(4, 1), (2, 2), (1, 3), (3, 4)]
```

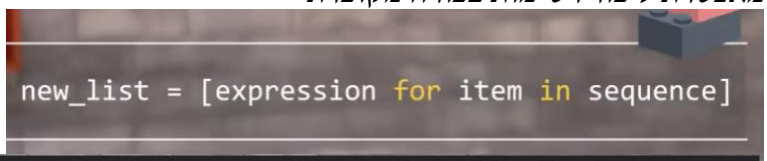
שימוש נוסף לפונקציית lambda
הוא הגדרה שלה כאיבר ישירות בתוך מבנה נתונים,
כמו טאפל, מילון או רשימה

```
calc_sqrt_list = [lambda x: x ** 2, lambda x: x ** 3, lambda x: x ** 4]
for func in calc_sqrt_list:
    print(func(3))
```

```
9
27
81
```

1.3 הרכבת רשימות

קיימת צורת כתיבה מקוצרת שנקראת "הרכבת רשימה" Comprehension List
מאפשרת ליצור רשימות בצורה מקוצרת




```
my_money_list = [x ** 2 for x in range(100)]
print(my_money_list)
```

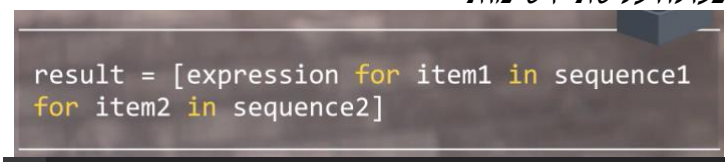
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289,
324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841, 900, 961, 1024,
1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600, 1681, 1764, 1849, 1936, 2025,
2116, 2209, 2304, 2401, 2500, 2601, 2704, 2809, 2916, 3025, 3136, 3249, 3364,
3481, 3600, 3721, 3844, 3969, 4096, 4225, 4356, 4489, 4624, 4761, 4900, 5041,
5184, 5329, 5476, 5625, 5776, 5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056,
7225, 7396, 7569, 7744, 7921, 8100, 8281, 8464, 8649, 8836, 9025, 9216, 9409,
9604, 9801]
```

שימושים אפשריים
הפעלת מתודה מורכבת על איבר

```
letters = ['a', 'b', 'c']
upper_letters = [x.upper() for x in letters]
print(upper_letters)
```

```
['A', 'B', 'C']
```

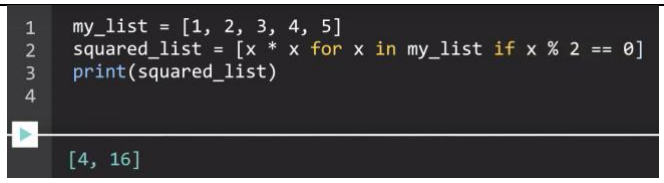
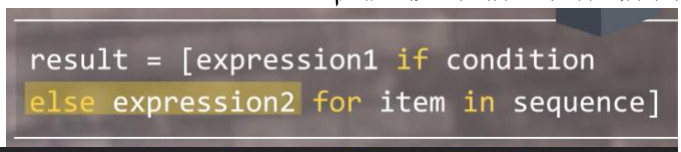
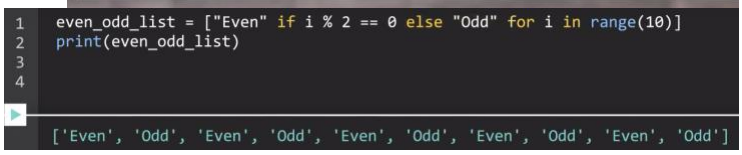
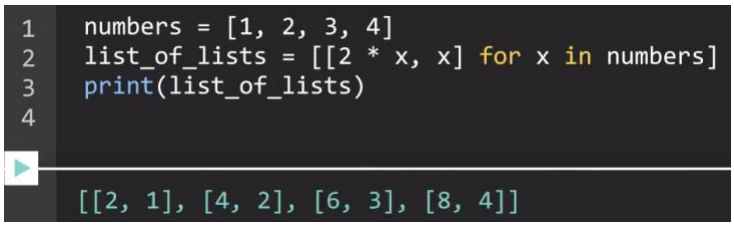
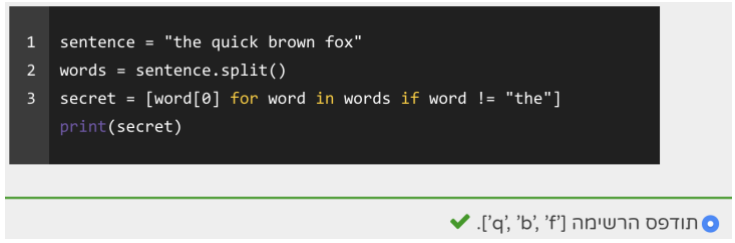
פעולה על שתי רשימות



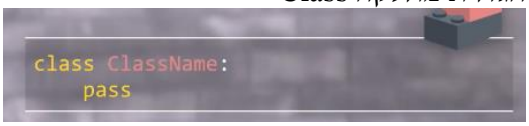

```
list1 = [1, 2, 3]
list2 = [5, 6, 7]

mult_list = [x * y for x in list1 for y in list2]

print(mult_list)
```

```
[5, 6, 7, 10, 12, 14, 15, 18, 21]
```

עם שילוב של תנאי

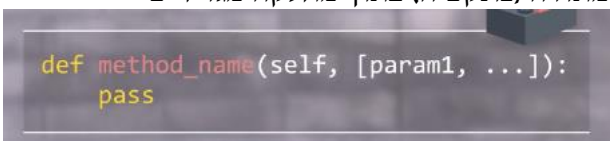
	 ביצוע עם מספר תנאים  או הרצה באם התנאי לא מתקיים   ייצור רשימה של רשימות  
יחידה 2.1 - מבוא לתכנות מונחה עצמים	גישת תכנות מונחה עצמים בעזרת תכנות מונחה עצמים נוכל להגדיר עצמים מסוגים שונים .על ידי תיאור התכונות והפעולות שהם יכולים לבצע. עצם הוא כל דבר שניתן לתאר אותו בעזרת .תכונות ופעולות ייחודיות שהוא יכול לבצע שילוב התכונות והפעולות נקבל מחלקה מחלקה היא התבנית הכללית המתארת עצם , וכל פרט ספציפי נקרא "מופע מחלקה" או בקיצור "מופע" המופעים הם ישות מסוימת של אותה תבנית זאת אומרת שבעזרת אותה מחלקה, אנחנו יכולים ליצור מספר רב של מופעים. לאחר שיצרנו מופעים, אפשר להפעיל את הפעולות שלהם – פעולות אלו נקראות מתודות מופע. מתודות אלה הן למעשה פונקציות.

הגדרת מחלקה Class



```
class ClassName:
    pass
```

מתודה (פונקציה) בתוך מחלקה מגדירים



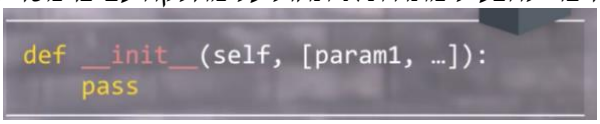
```
def method_name(self, [param1, ...]):
    pass
```

כתיבת מתודה והפעלתה

```
class BankAccount:

    def greet(self, name):
        print("Welcome", name)

def main():
    michals_account = BankAccount()
    michals_account.greet("Michal")
    amirs_account = BankAccount()
    amirs_account.greet("Amir")

main()
```

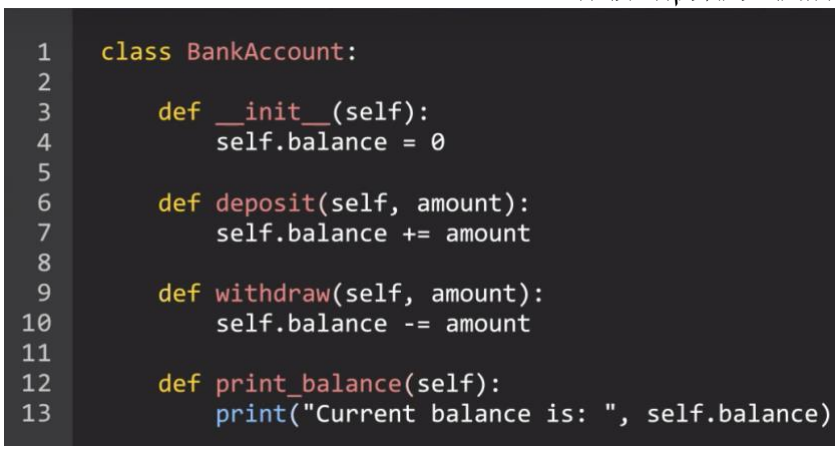
```
Welcome Michal
Welcome Amir
```

רצוי להפעיל מתודת איתחול על מחלקה עם פרמטרים



```
def __init__(self, [param1, …]):
    pass
```

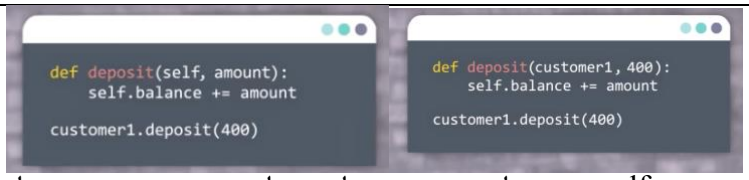
דוגמא למחלקה מלאה



```
class BankAccount:

    def __init__(self):
        self.balance = 0

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def print_balance(self):
        print("Current balance is: ", self.balance)
```

עוד על מחלקות בלינק

כאשר אנחנו קוראים למתודה של מופע, המופע שעליו היא פועלת מועבר כארגומנט למתודה בעצמו.
זאת אומרת שמאחורי הקלעים פייתון מעבירה למתודה **את מופע המחלקה עליו היא פועלת בתור הארגומנט הראשון**, ואחריו את הארגומנטים הנוספים שבסוגריים.



פרמטר self מגדיר לאיזה מופע של המחלקה מתייחסת או פועלת.

[2.3 - תכנות מונחה עצמים בפייתון - חלק ב](#)

פייתון מאפשרת לקבוע ערך ברירת מחדל לפרמטרים של מתודה במחלקה.
כלומר, זה יהיה הערך שייקבע לפרמטר כל עוד לא העברנו לו ערך אחר.

```
class BankAccount:

    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def print_balance(self):
        print("Current balance is:", self.balance)

def main():
```

```
def main():
    # Account with no previous amount of money.
    no_prev_money_acc = BankAccount()

    # Account with previous amount of money.
    with_prev_money_acc = BankAccount(2000)

    no_prev_money_acc.deposit(1)
    with_prev_money_acc.deposit(1)

    no_prev_money_acc.print_balance()
    with_prev_money_acc.print_balance()

main()
```

```
Current balance is: 1
Current balance is: 2001
```

ניתן לגשת גם לתכונות של מופע גם מחוץ למחלקה

```
    def withdraw(self, amount):
        self.balance -= amount

    def print_balance(self):
        print("Current balance is:", self.balance)

def main():
    my_account = BankAccount()
    my_account.deposit(200)
    my_account.withdraw(20)
    print(my_account.balance)

main()
```

```
180
```

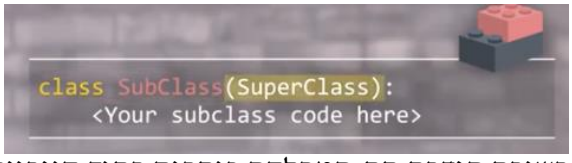
כדי לא להשתמש כך, יש להוסיף קו תחתון לפני שם התכונה

תכונות מחלקה

תכונת מחלקה כותבים בתוך בלוק המחלקה לפני המתודות.

	``python
	class BankAccount:
	    bank_name = "PayPy"
	
	    def __init__(self):
	        self._balance = 0
	
	    def deposit(self, amount):
	        self._balance += amount
	
	    def withdraw(self, amount):
	        self._balance -= amount
	
	    def print_balance(self):
	        print("Current balance is: ", self._balance)
	
	main()
	``
	``python
	def main():
	    print(BankAccount.bank_name)
	
	main()
	``
	``
	PayPy
	``

[2.4 א עקרונות מתקדמים בתכנות מונחה עצמים - הורשה חלק א](#)	כדי לעשות את זה נשתמש במה שנקרא ,"מנגנון ההורשה" באנגלית Inheritance שיאפשר לנו להגדיר מחלקה בעזרת מושגים ממחלקה קיימת ושיחסוך לנו את הצורך ביצירת מחלקה חדשה מאפס. למעשה, כל עצם או מושג בעולם משתייך למספר רב של משפחות בסדר היררכי. המחלקה שמורישה תכונות והתנהגות ,נקראת "מחלקת-העל" superclass. **והמחלקה שיורשת את התכונות וההתנהגויות ,נקראת "תת-מחלקה"** subclass. תת-המחלקה יורשת למעשה את כל המשתנים והמתודות של מחלקת-העל שלה, ואפשר להוסיף לתת-המחלקה משתנים ומתודות נוספות

[2.4 ב עקרונות מתקדמים בתכנות מונחה עצמים - הורשה חלק ב](#)	
	בשורת הגדרת תת-המחלקה כותבים בתוך סוגריים עגולים
	את שם המחלקה ממנה יורשים.
	
	``python
	class Dog(Animal):
	
	    def __init__(self, name, fur_color):
	        Animal.__init__(self, name)
	        self._fur_color = fur_color
	
	    def go_for_a_walk(self):
	        self._fun += 2
	        print("Waff waff! My fun level is rising, and its: ", self._fun)
	``

```
28  def main():
29      fluppy = Dog("Fluppy", "Brown")
30      fluppy.play()
31      fluppy.eat()
32      fluppy.go_for_a_walk()
33
34  main()
```

```
Waff waff! My fun level is rising, and its: 3
```

עוד שיטה לביצוע הגדרת תת-מחלקה ע"י פונקצית super()

```
1  class Dog(Animal):
2
3      def __init__(self, name, fur_color):
4          super().__init__(name)
5          self._fur_color = fur_color
6
7      def go_for_a_walk(self):
8          self._fun += 2
9          print("Waff waff! My fun level is rising, and its: ", self._fun)
```

עוד ניתן לקרוא בלינק. דוגמא לשימוש בחישוב שטחים

נכיר שתי פונקציות שימושיות במיוחד
כשעובדים עם אובייקטים במחלקות

פונקציית isinstance
הפונקציה מקבלת שני פרמטרים:
אובייקט כלשהו וסוג טיפוס,שיכול להיות גם שם של מחלקה.
הפונקציה מחזירה true
אם האובייקט הראשון שהועבר הוא אכן מהטיפוס של השני
או אחד היורשים שלו.
אחרת מחזירה false.

```
1  print(isinstance(2, int))
2
3
4
```

```
True
```

```
1  print(isinstance("text", int))
2
3
4
```

```
False
```

הפונקציה השנייה שנכיר נקראת issubclass.
הפונקציה מקבלת שני פרמטרים שהם שמות של מחלקות
או שמות טיפוסים מובנים, **ומחזירה true** אם הפרמטר הראשון
הוא תת-מחלקה של השני,
אחרת היא תחזיר false.

```
print(issubclass(Dog, Animal))
print(issubclass(Animal, Dog))
print(issubclass(Animal, int))
```

```
True
False
False
```

עקרונות מתקדמים בתכנות מונחה עצמים - פולימורפיזם

אנחנו רוצים לתת מימוש שונה למתודה מסוימת במחלקה היורשת וגם לרשת את יתר התכונות והמתודות של מחלקת-העל ללא שינוי.
העיקרון שעוזר לנו לעשות את זה הוא עיקרון הפולימורפיזם שמאפשר להתייחס בצורה אחידה לאובייקטים מסוגים שונים כאילו הם מאותו הטיפוס.

כדי לעשות את זה נשתמש בפונקציה המובנית super,
פונקציה שמאפשרת לקרוא למתודה ישירות ממחלקת-העל, בלי לכתוב את שם המחלקה המפורש.

נכתוב את הפונקציה super
לאחריה נקודה, ואז את שם המתודה ששייכת למחלקת-העל שבה אנו מעוניינים.
צורת כתיבה זו למעשה מזמנת את המתודה המקורית ממחלקת-העל

```
        self._fur_color = fur_color

    def go_for_a_walk(self):
        self._fun += 2
        print("Waff waff! My fun level is rising, and its: ", self._fun)


    def eat(self):
        super().eat()
        print("eating a bone!")

def main():
    my_dog = Dog("barkey", "brown")
    my_dog.eat()

main()
```

השיטה הזו מאפשרת לנו **לשנות או להתאים מתודה במחלקה יורשת בלי לשכפל או לכתוב מחדש שורות קוד** שכבר מומשו במחלקת-העל.

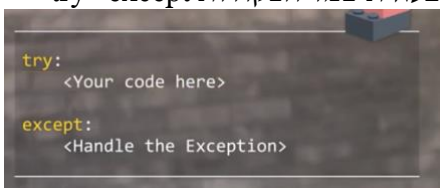
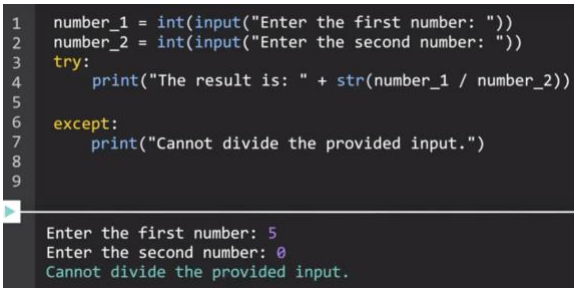
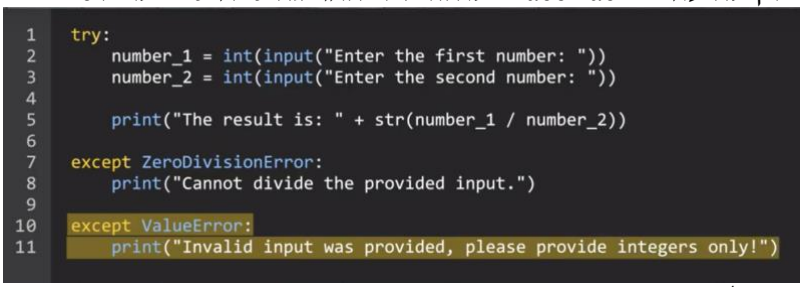
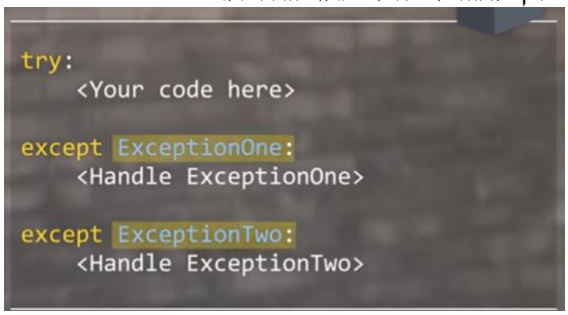
ניתן לדרוס ערך פונקציה של מתודה. ע"י פונקציה dir() ניתן לראות אילו פונקציות קיימות

```
        self._fur_color = fur_color


    def go_for_a_walk(self):
        self._fun += 2
        print("Waff waff! My fun level is rising, and its: ", self._fun)

    def __str__(self):
        return "My name is {} and I am a dog!".format(self._name)

def main(self):
    my_dog = Dog("barkey", "brown")
    print(my_dog)

main()
```

```
My name is barky and I am a dog!
```

3.1 שגיאות וחריגות

שגיאות תחביר הינן בגלל אופן כתיבת קוד , ויש לתקנן עפי כתיבה.
ישנן שגיאות ריצה הנוצרות בגלל מקרי קצה ונתונים לא מתאימים.

	אובייקט חריגה Exception אובייקט שמתאר שגיאה ישנן חריגות מובנות Built In Exception. התכנית קורסת ומציגה פלט Traceback שמתאר מה קרה בקוד ואיזו חריגה נזרקה. פלט ה-Traceback יכול לספר לנו בדיוק מה קרה שורה אחרי שורה. סוגי חריגות בלינק
3.2 טיפול בחריגות	כיצד מונעים מחריגה להקריס את התוכנית איך תופסים חריגה... בעזרת צמד הפקודות try except  דוגמא  הרצת exception רק בסוג מסוים של חריגה. ניתן להיעזר ב TraceBack לזהות את החריגה שנדרשים לתפוס  ניתן להגדיר יותר מחריגה אחת  אם נרצה לבצע פעולות על מופע של חריגה, נעשה את זה בעזרת יצירה של משתנה שמצביע על המופע. כדי ליצור מופע של חריגה

```
number_1 = int(input("Enter the first number: "))
number_2 = int(input("Enter the second number: "))

try:
    print("The result is: " + str(number_1 / number_2))

except ZeroDivisionError as e:
    print("Cannot divide the provided input.")
    print(e)
```

```
Enter the first number: 5
Enter the second number: 0
Cannot divide the provided input.
division by zero
```

במצב שאין חריגה ניתן להריץ קטע קוד אחרי else

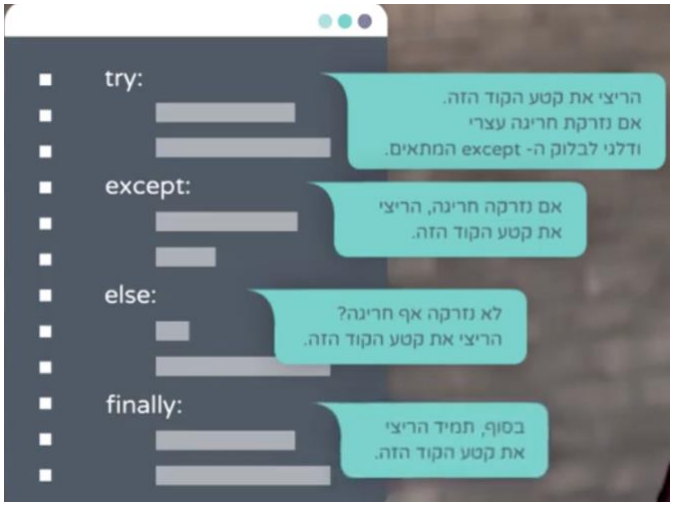
```
NOT_EXISTING_FILE = "imaginaryFile.txt"
EXISTING_FILE = "realism.txt"
def get_file_line_num(file_name):
    try:
        f = open(file_name)
    except IOError:
        print("IO Error with file: ", file_name)
    else:
        print("Number of lines in file: ", len(f.readlines()))
        f.close()
get_file_line_num(EXISTING_FILE)
```

```
Number of lines in file: 5
```

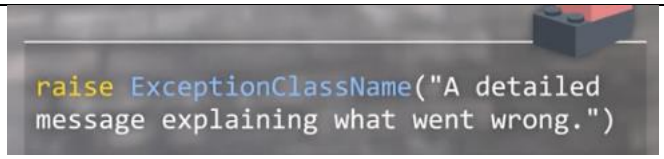
ולבסוף finally

```
try:
    <Your code here>

# except clause is optional
except:
    <Handle the Exception>

# else clause is optional
else:
    <Execute if no exception was thrown>

finally:
    <Always run this code>
```

לסיכום



[3.3 יצירת חריגות בהתאמה אישית](#)	דוגמא ליצירת חריגה יזומה. אנחנו צריכים לממש בקוד תגובה שתמנע מהפונקציה לרוץ במצב של הזנת קלט לא תקין.



דוגמא

```
1  def factorial(n):
2      if n < 0:
3          raise ValueError("Factorial expects non-negative integers.")
4
5      fact = 1
6      for i in range(n, 0, -1):
7          fact = fact * i
8      return fact
9
10 print(factorial(-1))
```

```
Traceback (most recent call last):
  File "fact.py", line 10, in <module>
    print(factorial(-1))
  File "fact.py", line 3, in factorial
    raise ValueError("Factorial expects non-negative integers.")
ValueError: Factorial expects non-negative integers.
```

למזלנו פייתון מאפשרת לנו, ליצור חריגות חדשות לחלוטין.

בניית מחלקה של חריגה

```
1  class FactorialArgumentError(Exception):
2
3      def __init__(self, arg):
4          self._arg = arg
5
6      def __str__(self):
7          return "Provided argument %s is not a positive integer." % self._arg
8
9      def get_arg(self):
10         return self._arg
11
```

שימוש במחלקה

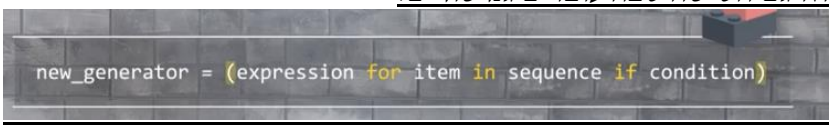

```
12 def factorial(n):
13     if not isinstance(n, int) or n < 0:
14         raise FactorialArgumentError(n)
15
16     fact = 1
17     for i in range(n, 0, -1):
18         fact = fact * i
19     return fact
20
21 result = factorial(-4)
```

```
Traceback (most recent call last):
  File "factorial.py", line 21, in <module>
    result = factorial(-4)
  File "factorial.py", line 14, in factorial
    raise FactorialArgumentError(n)
FactorialArgumentError: Provided argument -4 is not a positive integer.
```

ניתן לתפוס את החריגה שיצרנו ולהציג הודעה

```
        return "Provided argument %s is not a positive integer." % self._arg

    def get_arg(self):
        return self._arg

def factorial(n):
    try:
        if not isinstance(n, int) or n < 0:
            raise FactorialArgumentError(n)

    except FactorialArgumentError as e:
        print("Function Expected positive integer, and instead got %s." %
              type(e.get_arg()))

    fact = 1
    for i in range(n, 0, -1):
        fact = fact * i
    return fact
```

שיטת הפעולה שמחשבת בכל פעם ערך יחיד ,נקראת "חישוב ביטויים עצל"
Lazy Evaluation
השיטה הזו חוסכת לנו מקום בזיכרון של המחשב
והיא שימושית כשעובדים עם כמויות גדולות של מידע,
וזו גם השיטה שבה עובדים גנרטורים.




```
# Do try this at home.
prime_generator = (n for n in range(10 ** 10) if is_prime(n))
for prime_number in prime_generator:
    print(prime_number)
```

```
23
29
31
37
41
43
```

הגנרטור מייצר מספר ראשוני אחד
בכל איטרציה של לולאת ה-for,
ככה הוא לא צריך לחשב את כל הערכים בבת אחת,
אלא הוא פשוט מפיק ערך אחד בכל פעם שמבקשים ממנו
באופן עצל.

כדי להביא את הערך הבא קיימת פונקציית next

```
prime_generator = (n for n in range(10 ** 10) if is_prime(n))
print(next(prime_generator))
print(next(prime_generator))
print(next(prime_generator))
```

```
2
3
5
```