מג'אווה לפייתון חלק א'

מבנה של סקריפט פייתון-

בדומה לתוכניות של ג'אווה, לכל פרויקט יש כמה מסמכים או סקריפטים, ולכל מסמך יש את הסיומת שלו- בג'אווה: '.java' או '.py' בפייתון.
לתוכניות בשפות סטטיות יש כמה קבצים וקובץ ראשי המכיל פונקציית main שרץ בתחילת התוכנית "ומנהל" אותה.
בפייתון לעומת זאת המפרש מריץ סקריפטים החל מהמסמך הראשון ועד המסמך האחרון לפי סדר מסוים ,ואין צורך בפונקציה ראשית שפועלת בתחילת התוכנית,
אך על כל פנים, ניתן להגדיר פונקציה ספציפית כנקודת תחילת התוכנית, זה שימושי בעיקר כדי להבין כיצד התוכנית עובדת.
הדרך הנהוגה להגדרת פונקציה ראשית היא ע"י "__main__" == __name__ if :

```
def main():
    print("Hello World!")

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

סקריפט ו-module -

כדי להבין מה זה __name__ נצטרך להבין תחילה איך להריץ קבצי פייתון-
ישנם שתי דרכים (עיקריות) להורות לפייתון להריץ את קבצי הפייתון:
פייתון היא שפה שמפעילה "מפרש" שעובר שורה אחר שורה של הקוד ומבצע אותה או שומר בזיכרון את הפעולה של אותה שורה.
עם התקנת השפה על המחשב אנחנו מתקינים גם מצב "אינטראקטיבי", שהוא תוכנית שרצה בזמן אמת, מחכה לפקודות בשפת פייתון ומבצעת אותם.
כדי להפעיל את המצב האינטראקטיבי, צריך רק להפעיל את הפקודה python או python3 בטרמינל, והוא יקרה למצב הנ"ל.
נוכל לייבא למצב הקיים גם modules של פייתון שכתבנו מראש וכך להפעיל אותם מתוך התוכנית האינטראקטיבית ע"י הפקודה import ושם הmodule או הpath שלו (במידה והטרמינל לא נמצא באותה תיקייה של ה-module).
הערה: כדי לצאת ממצב אינטראקטיבי צריך להפעיל את הפונקציה exit() שיוצאת מתהליך וחוזרת לטרמינל, ניתן גם ללחוץ ctr+d.

דרך נוספת להריץ את הקוד היא כסקריפט, ואז המשתמש צריך להזין בטרמינל, בתיקייה של הקבצים את הפקודה:
'python3 name_of_the_file.py' (הפקודה היא python3 או python ואז שם הסקריפט בסיומת '.py').

סקריפט הוא קובץ טקסט של פייתון(קובץ '.py') שמכיל קוד פייתון שמטרתו לרוץ ישירות ע"י המשתמש.
לעומתו module הוא קובץ טקסט (קובץ '.py') המכיל קוד פייתון ומטרתו להיות תוכנית שמיובאת לתוכניות פייתון אחרות, אז בעצם ההבדל בניהם הוא שהראשון נועד להרצה והשני נועד לייבוא לתוכנית רצה.

לפעמים נרצה להשתמש בסקריפט אחד בתוך סקריפט אחר כmodule.
בזכות המשתנה __name__ נוכל לקבוע אם נרצה להריץ את הקובץ כסקריפט או לייבא אותו כmodule.
כשאנחנו מריצים את הקובץ כסקריפט המשתנה __name__ יהיה שווה למחרוזת "__main__" אבל כשמריצים את הקוד כmodule בתוך תוכנית אחרת, אז ערך המשתנה הוא שם הקובץ.

עכשיו מה שיקרא אם נריץ את הסקריפט הוא שהמפרש יבדוק מה ערכו של __name__ ואם המשתנה יהיה שווה

__main__ הוא יבין שהקוד נקרא כסקריפט, ויריץ אותו החל מנקודה שאותה צוינה כנקודת ההתחלה, למשל למעלה קבענו שהתוכנית תתחיל מהפונקציה main.
אך אם המשתנה לא שווה ל- __main__ המפרש יבין שזהו module בתוך תוכנית אחרת, ובמקרה כזה לא יהיה צורך להגדיר נקודת התחלה, כי התוכנית שמריצה את הmodule תקבע באילו משתנים היא רוצה להשתמש מתוך הקובץ.

בלוקים וסיומות של פקודה-

בשפות כמו ג'אווה אנחנו מציינים סיום פקודה בנקודה פסיק ';', ותחילת קטע וסיומו עם סוגרים מסולסלים '{ }'.
כל קטע קוד המתחיל בסוגר מסולסל (פותח) ומסתיים בסוגר מסולסל (סוגר) נקרא בלוק.
הסוגריים מאפשרים לתוכנה לזהות היכן נגמר הקטע, אך הם לא מחייבים לשמור על איזשהו סדר, מה שבהרבה מקרים גורם למתכנתים מתחילים ,אבל לא רק, ליצור קוד מבולגן שקשה לעקוב אחר הלוגיקה שלו, דבר שמקשה על מתכנתים חדשים "להיכנס" לתוכנית.
פייתון היא שפה שבנויה בהתאם לאיזשהו מניפסט שמחייב אותה, לכן כדי להקפיד על עיקרון "קוד נקי" וכדי שהשפה תהיה דומה ככל הניתן לשפה אנושית, בלוק בפייתון מצוין בנקודתיים ,ירידת שורה ובהזחות במקום בסוגרים עגולים ונקודה פסיק.
סיומות של פקודה לא נגמרות עם איזשהו סימן מיוחד אלא פשוט בירידת שורה, מה שמקנה לשפה מראה של כותרת ופירוט או רשימת סופר, שהיא בהחלט יותר אנושית מהמבנה המוכר של שפות תכנות כמו ג'אווה.
דוגמא:

```
public class Test { public static void main(String args[]) {
        String array[] = {"Hello, World", "Hi there, Everyone", "6"};
        for (String i : array) {System.out.println(i);}}}
```

קוד חוקי בג'אווה שמדפיס למסך כל אחת מהמחרוזות במערך array, הקוד לא מחויב לחוקי אסתטיקה קפדניים במיוחד.
ואותו הפונקציונליות בדיוק בפייתון:

```
stuff = ["Hello, World!", "Hi there, Everyone!", 6]
for i in stuff:
    print(i)
```

כל שורה היא פקודה נפרדת, וכל בלוק בנוי מכותרת (במקרה הזה ההצהרה על לולאה), נקודתיים, ותחילת הבלוק בשורה מתחת עם רווח מתחילת מיקום הכותרת.
במבנה כזה נוצרת איזושהי היררכיה- כל הפקודות שנחשבות שוות אחת לשנייה, כלומר מוכלות באותו הבלוק, יתחילו מאותה נקודה רק בשורות נפרדות, כך שבמקרה כמו הקוד המצוין לעיל, אם נרצה להוסיף פקודה שתבוא בסוף הלולאה, נוכל לזהות אותה בקלות גם אם אנחנו לא כותבי הקוד, כי היא פשוט תתחיל מאותה נקודה שהתחילה הכותרת של הלולאה:

```
 stuff = ["Hello, World!", "Hi there, Everyone!", 6]
for i in stuff:
    print(i)
print("end")
```

הזחות הן דבר מרכזי בפייתון ואם הפקודה שבאה באותו הבלוק לא זהה ברווח לשאר הפקודות בבלוק, או שהיא לא בדיוק במרחק המתאים מהכותרת המפרש לא יכול לזהות את הפקודה, או שהוא יזהה אותה בבלוק אחר.

הערות (comments)-

כתיבת הערות בקוד עוזרת לתאר את תהליך החשיבה של המתכנת, עוזרת לו ולאחרים להבין יותר מאוחר את כוונתו בכתיבת שורות ספציפיות או הקוד בכללותו, עוזרת במציאת שגיאות ותיקונם, שיפור הקוד ושימוש בו (אינטגרציה) בפרויקטים אחרים.
בג'אווה יש שני סוגים של הערות: הערת שורה שאותה אנחנו מציינים עם '//' והיא מגדירה שכל מה שבא מהסימון של שני הקווים האלכסוניים ועד סוף השורה יחשב כהערה ולא יקומפל ע"י המהדר;
או הערת בלוק (הערה של כמה שורות) שאותה אנחנו מסמנים עם */ בתחילת בלוק ,הערה, ובסוף הבלוק אנחנו סוגרים עם /* (יש עוד כמה סוגים כמו הערות javadocs אך אלו שני סוגי הערות המרכזיים).

בפייתון סימון הערות הוא בצורה שונה, כשרוצים לעשות הערות של שורה אחת משתמשים בתו '#' .

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

במדריך "style guide for python code" כתוב שגודל השורה המומלץ הוא כ 72 תווים.
במידה ואנחנו חורגים מהגודל מומלץ לפצל את הערות לכמה שורות של הערות או לבלוק הערות , מה שמוביל אותנו לסוג השני של הערות של פייתון-הערות בלוק:
בשביל הערות בלוק כותבים בהתחלה " " " (שלושה מרכאות) ומסיימים את בלוק גם בשלושה מרכאות:

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

טיפ: נהוג להתחיל קובץ פייתון בכמה שורות של הערות, שורות אלה מציינות מידע אודות הפרויקט, מטרת הקובץ, מי המתכנת ורישיון התוכנה.
בדר"כ הערות כאלה מסוגננות בצורה הבאה:

```
#-----------------------------------------------------------------
#demonstrates how to write ms excel files using python-openpyxl
#
#(C) 2015 Frank Hofmann, Berlin, Germany
#Released under GNU Public License (GPL)
#email email@email.com
#-----------------------------------------------------------------
```

יש עוד סוג של הערות הפייתון והוא docstring .
docstring היא הערה שמוסיפים מתחת לכותרת של פונקציה, מחלקה, שיטה של מחלקה או module,
והיא מסייעת לצרף הערות לחלקים בפרויקט כך שגם מחוץ לפרויקט יהיה ניתן לקרוא עליהם:

```
def add(value1, value2):

    """Calculate the sum of value1 and value2."""

    return value1 + value2
```

add היא פונקציה שמקבלת שני ערכים ומחזירה את הסכום שלהם.
הוספנו לה docstring ועכשיו נוכל לבדוק מה הפונקציה עושה מבלי להשתמש בה:

```
>> print add.__doc__

   Calculate the sum of value1 and value2.
```

יבוא ספריות חיצוניות-
אחד ההבדלים הבולטים בין ג'אווה לפייתון הוא בהתייחסות ל-modules שונים.
בג'אווה כל מסמך מוגדר כמחלקה חדשה, ורק המסמך הראשי מכיל קובץ main.
זה שכל מסמך נחשב לאובייקט עוזר למנוע בעיות של ambiguous בקוד, כלומר שימוש בשתי מתודות או יותר של מחלקות

שונות עם אותו השם.
פייתון יותר דומה ל-c/c++ בדבר הזה, וניתן להגדיר פונקציות שלא נחשבות למתודות של מחלקות ספציפיות, ואז כאשר מייבאים את ה-module של הפונקציה ניתן לקרוא לפונקציה בשמה אבל צריך להגדיר מאיפה הגיע הפונצייה,
לצורך הדוגמא נניח שהמודול cow מכיל רק פונקציה אחת והיא moo שמדפיסה למך את המילה moo :

```
#moo.py
>>> def moo():
...  print('moo')
```

עכשיו נניח ואנחנו כשנייבא אותו נוכל להשתמש בפונקציה במודול ולהגדיר מהיכן הגיע:

```
>>> import cow
>>> moo.moo()
'moo'
```

מוסכמות-
אלו רק מוסכמות, אין חובה לציית להם, אבל אם אתם מתכננים לעבוד עם אנשים אחרים שאמורים לקרוא את הקוד שלכם מומלץ להסכים על מוסכמות בניכם.
במדריך הרשמי של פייתון מצוינות כמה מוסכמות בנוגע לכתיבת קוד נכון בפייתון:
* לא להשתמש באות l (האות א"ל קטנה)או I (האות אי"י גדולה)או באות O (האות או"ו גדולה או קטנה) כייצוג שם של משתנה, היות ובחלק מהפונטים קשה להבדיל בין האותיות האלה למספרים אחד או אפס.

* כל המזהים חייבים להיות כתובים בascii ואמורים להיות כתובים באנגלית בלבד.

* שמות של modules אמורים להיות כתובים באותיות קטנות בלבד, להשתמש בקו תחתון במקרה שרוצים שהשם שלו יהיה בנוי מכמה מילים למשל my_first_project.py

* שמות של מחלקות אמורות להיות במבנה של CapWord כלומר להתחיל באות גדולה, וכל פעם שרוצים להוסיף מילה חדשה לשם המחלקה נוסיף אותו עם אות גדולה למשל: MyClass .

* חריגות הן מחלקות בפייתון (נראה בהמשך) ולכן שמן יהיה כשם של מחלקה. במידה והחריגה היא שגיאה נהוג להוסיף את המילה Error לסיפא של שמה, למשל: ZeroDevisionError .

* שמות של פונקציות צרכים להיות באותיות קטנות עם הפרדה של '_' בין מילים, למשל: " def print_hello():"

* שמות של משתנים (גלובליים או לוקלים) צריכים להיות כמו שמות של פונקציות, יוצא דופן הוא משתנה גלובלי של module שכדאי לסמן שהוא לא לשימוש הכלל אלא משתנה פרטי של המודול, במקרה כזה נסמן את המשתנה ב-"__name__",ומשתנה קבוע.

* משתנים קבועים מציינים באותיות גדולות עם ' _ ' שמפריד בין מילים: MAX_VALUE, TOTAL וכו'.

אופרטורים –

אופרטורים הם פונקציות מיוחדות של שפות תכנות שמטרתן לשפר את קריאות התוכנית, את הדמיון בינה לבין טקסט מתמטי, לוקי או שפה טבעית, או כדי להדגיש משמעות של פעולה כלשהי.
את האופרטורים ניתן לחלק לארבעה קבוצות עיקריות - אריתמטיים, השמה, השוואה, ולוגים.

כמו כן יש קבוצות של אופרטורים מיוחדים כגון: אופרטורים של זהות או שייכות, אופרטורים של בסיסי נתונים ו-bitwise .

אופרטורים אריתמטיים-
אופרטורים אריתמטיים הם כל אותם אופרטורים שאנחנו משתמשים בהם להגדרת פעולות מתמטיות יסודיות.
בג'אווה יש את הפעולות המתמטיות הבסיסיות:
חיבור שמסומן ב- '+' , חיסור '-' , כפל '*' , חילוק '/' , ושארית חלוקה המוכרת בשם modulus '%'.
המתכנתים של ג'אווה גם הוסיפו כמה syntactic sugar (תחביר קצר ומובן יותר לפעולות שדורשות הרבה תווים)-
הוספת אחד לסכום '++', והפחתת אחד מהסכום '- -'.
בפייתון לעומת זאת הפעולות האריתמטיות היסודיות של השפה הן יותר מתקדמות, למשל ניתן למצוא חזקה שמסומנת ב-'**' , או ערך תחתון של חלוקה (מה שהיינו עושים לו קאסטינג ל-int בג'אווה) המסומן ב-'//', אך הsyntactic sugar של ג'אווה הוא לא חלק מהשפה.
סיכום הפעולות האריתמטיות של פייתון:

+	חיבור	x+y
-	חיסור	x-y
*	כפל	x*y
/	חילוק	x/y
%	Modulus	x%y
**	חזקה	x**y
//	ערך תחתון של חלוקה	x//y

אופרטורי השמה-
אופרטורים של השמה הם כל אותם האופרטורים שמכניסים ערך לתוך משתנה.
בתיאוריה קיים רק אופרטור אחד של השמה לג'אווה ופייתון והוא האופרטור '=',
אך עם הזמן פותחו עוד כמה syntactic sugars לשפות רבות שעוזרות לקצר תהליכים כגון += שהוא מקצר תחבירים של חיבור והשמה , למשל במקום לכתוב x= x+y , נוכל לכתוב x+=y , וכנ"ל לגבי כל אחד מהאופרטורים האריתמטיים(וה-bitwise).
גם בג'אווה וגם פייתון במקרה זה ניתן להשתמש ב-syntactic sugars הזה.

אופרטורי השוואה-
כל אותם האופרטורים שנועדו כדי לתת לנו אינדיקציה של גודל או סוג לערכים של המשתנים.
אנחנו משתמשים באופרטורי השוואה כדי למדוד האם אובייקט מסוים הוא גדול, קטן ,שווה ערך, או לא שווה ערך מאובייקט אחר, ומקבלים ערך "אמת" או "שקר" במידה והביטוי נכון.
בפייתון ובג'אווה האופרטורים זהים: '<' -הערך הימני גדול יותר, '>' -הערך השמאלי גדול יותר, '==' -שני הערכים שווים, ו- '!=' –הערכים אינם שווים.
בנוסף יש כמה syntactic sugars לשפה שהם שילוב של שני אופרטורים גדול/קטן ו-שווה: '<=' הערך הימני גדול **או** שווה, ועל אותה הדרך רק עם הערך השמאלי ב- '>='.

אופרטורים לוגיים-
אופרטורים לוגיים הם כל אותם אופרטורים שמגדירים לנו נכונות בין ביטויים, כלומר הם מחזירים "אמת" אם ביטוי מסוים או כמה ביטויים נכונים, ו"שקר" אחרת.
שלא כמו האופרטורים הקודמים הסינטקס של האופרטור שונה, אך התוכן שלו זהה, למשל
אם יש לנו שני ביטויים(או יותר) ואנחנו רוצים לבדוק ששני הביטויים עם ערך אמת, בג'אווה נעשה את זה עם האופרטור "וגם" && ובפייתון ממש נכתוב and ,
אם נרצה לוודא שלפחות ביטוי אחד נכון, נשתמש באופרטור 'או' שבג'אווה מצוין כ-'||' ובפייתון ממש כותבים "or" ,
ואם נרצה לוודא שההפך של ביטוי הוא מה שקורה נשתמש באופרטור not שבג'אווה אנחנו מציינים אותו ב-'!' ובפייתון

ממש כותבים not:

האופרטור	ג'אווה	פייתון
and	x<5 && y<7	x<5 and y<7
or	x<5 || y<7	x<5 or y<7
not	!(x<5 && y<7)	not(x<5 and y<7)

אופרטורי זהות ושייכות-

אופרטורי זהות-

אופרטורי זיהוי הם אופרטורים לבדיקה האם שני אובייקטים מצביעים לאותו מקום.
בג'אווה כשאנו בונים מחלקה חדשה ומגדירים אובייקט שמושם לו ערך המחלקה, לדוג' Person p = new Person() , אז p במקרה זה הוא לא אובייקט מסוג Person אלא מצביע לאובייקט מסוג Person.
יש לזה הרבה יתרונות, למשל במקום לשלוח לפונקציה פרמטר מטיפוס אובייקט ואז היא תעתיק אותו, כפי שהיא עושה במשתנים פרימיטיביים כמו int וכו', שמעתיקה אותם ומחזירה ערך אך לא משנה את הפרמטר שנשלח , נשלח לה מצביע למשתנה ואז השינוי יהיה בזיכרון מה שיחסוך מקום(העתקה של אובייקט כבד לוקחת זמן ומקום נוסף בזיכרון), והשינוי יהיה ניכר.
אבל יש לכך גם חסרונות למשל בשימוש באופרטור '==' על אובייקט מורכב תתבצע בדיקה על המצביע ולא על הערך שהוא מחזיק, מה שאומר שהבדיקה תהיה לפי המיקום בזיכרון של המצביע.
בפייתון לעומת זאת כל המשתנים הם מצביעים, וניתן להגדיר למחלקות אופרטורים כמו ב ++c, כפי שנראה בהמשך, לכן השימוש ב-'==' יכול להיות ממש לפי ערך ולא לפי מיקום בזיכרון, אבל כדי שלא תישלל האפשרות לבדוק מצביעים גם לפי המיקום שלהם בזיכרון יש את האופרטור is או האופרטור is not , כך למשל נוכל לבדוק את הדבר הבא:

```
>> x=3
>> z=x
>> y=z
>> y is x
      True
```

וכנ"ל נוכל לבדוק חוסר התאמה עם is not.

אופרטורי שייכות-

הם אופרטורים בלעדיים לפייתון שבודקים האם ערך מסוים נכלל בקבוצה כלשהי.
כדי לבצע את הבדיקה משתמשים במילה השמורה in או ב- not in כדי לבדוק חוסר שייכות
למשל:

```
>> primes_num_under_ten = [2 ,3 ,5 ,7]
>> 8 not in primes_num_under_ten
      True
```

אופרטורי Bitwise-

אופרטורי bitwise הם אופרטורים שפועלים על מספרים בינריים.
לפעמים נצטרך לבצע חישוב ברמת הביטים על משתנים, למשל בפרוטוקולי תקשורת לחישוב checksums וכדו', או באלגוריתמי דחיסה והצפנה.
האופרטורים בג'אווה ובפייתון זהים במקרה זה, למעט אופרטור אחד כפי שנראה בהמשך.
bitwise מבצעים פעולות לוגיות על ביטים- מחשבים, שמתקשרים בשפה בינארית, מחשיבים ערך כ"שקר" אם ערכו הוא 0 אחרת (1) ערכו אמת, או יותר מדויק אם יש זרם הערך 1, ואם אין זרם הערך הוא 0.
מכאן שפעולות שאנחנו מגדירים על ביטויים לוגים ניתן לבצע גם בצורה בינארית.
לפייתון וג'אווה יש שישה אופרטורי bitwise משותפים:

and – שדומה לסימון של 'וגם' בביטויים בוליאניים בג'אווה –'&' ,
ובדומה גם כאן ערכו אמת אמ"מ לשני הביטויים יש ערך 1, אחרת הערך שחוזר הוא 0, ואם מדברים על מספר המורכב מכמה ביטים, אז רק אם שני הביטים שבאותו המקום(באותו החזקה של 2) עם ערך 1, אז התוצאה תקבל ערך אחד באותו המקום, למשל: $5\&1 = 0101 \ \& \ 0001 = 0001 = 1$, כי רק במקום 2^0 לשני הביטים יש ערך 1.

or- שמסומן ב- ' | ', והוא פועל בצורה דו לאופרטור 'או' בביטויים בוליאניים, כלומר מקבל ערך 1 אם לפחות אחד משני הביטים באותו המקום עם הערך אחד: $5 \mid 1 = 0101 \mid 0001 = 0101 = 5$.

not- מסומן ב- '~' והוא מסמן שלילת הביטוי, כלומר כל מה שערכו אחד יהפוך להיות אפס וכנ"ל אפס יהפוך לאחד, למשל: $\sim 5 = \sim 0101 = 1010 = 10$.

xor- פעולה לוגית שמסומנת ב-^ ,והיא מגדירה שהערך הוא אחד רק אם **באחד** מהביטויים יש ערך אחד באותו המקום, כלומר בעוד ש-*or* מגדיר ערך גם אם שני הערכים הם 1 , *xor* מגדיר ערך אחד אם רק אחד משני הביטויים הוא אחד, למשל: $5 \wedge 1 = 0101 \wedge 0001 = 0100 = 4$.

פעולות נוספות ברמת ה-bitwise הן:
shift left- שמסומן ב-'<<', והוא משמש לדחוף את הביטוי ביט אחד שמאלה ע"י הוספה של המספר אפס לביט הכי ימני והוצאה של כל הביט השמאלי ביותר מהביטוי, למשל: $9 << 1 = 1001 << 1 = 0010 = 2$.
shift right- שמסומן ב-'>>' ,והוא מעתיק פעמיים את הביט השמאלי ביותר ומוריד את הביט הימני ביותר מהביטוי, למשל:$9 \gg 1 = 1001 \gg 1 = 1100 = 12$.
וכפי שאמרנו קודם, לג'אווה יש אופרטור נוסף שאין לפייתון והוא האופרטור '>>>', שדומה ל*shift left* רק בכיוון ההפוך, למשל:
$9 \gg 1 = 1001 \gg 1 = 0100 = 4$.
בכל הדוגמאות לעיל השתמשנו בדוגמא ב-*shift* לאחד אבל ניתן לעשות *shift* גם ל2 ואז המספרים ינועו שתיים ימינה או שמאלה וכו'.

טיפוס נתונים *(DATA TYPES)* –

טיפוס נתונים הוא מושג המגדיר מה הערכים שכל סוג של משתנה יכול לקלוט, ובאילו דרכים.
בג'אווה לכל משתנה חייב להיות טיפוס נתונים מוגדר מראש כדי שהמהדר יוכל לזהות אותו בזמן קומפילציה, למשל כשנרצה להגדיר משתנה שערכו מספר שלם נשתמש בטיפוס הנתונים int או long , אך בפייתון אין הגדרה למשתנים, וכל משתנה הוא בעצם מטיפוס נתונים אחד- מצביע, כך שהוא יכול להיות כל טיפוס נתונים שהוא, ואף להשתנות בזמן אמת מטיפוס אחד לאחר.
זה לא אומר שלא קיימים טיפוסי נתונים נוספים בפייתון , גם לפייתון יש טיפוסי נתונים וניתן גם להגדיר טיפוסים ע"י בניית מחלקות ופונקציות חדשות, וכדי לזהות את סוג הטיפוס משתמשים בפונקציה type() שמקבלת כארגומנט משתנה ומחזירה מה הטיפוס שלו.
את טיפוסי הנתונים המוגדרים מראש של פייתון ניתן לחלק לכמה קבוצות:

מספרים (numeric types)- טיפוסים מסוג מספרים ניתן לחלק לשלושה סוגים: טיפוסי מספרים שלמים שמוכר כ-int, מספרים ממשיים שמוכר כ-float, ומספרים מרוכבים (complex numbers)שאנחנו מגדירים אותם עם האות j:

```
>> type(2)
      <class 'int'>
>> type(2.0)
      <class 'float'>
>> type(2j)
      <class 'complex'>
```

Boolean – משתנים בוליאניים המוכרים לנו מג'אווה מהווים טיפוס עצמאי בפייתון שערכו הוא 'אמת' או 'שקר'.
בניגוד לג'אווה, מציינים את המשתנים באותיות גדולות בתחילת המילה כך שערך אמת הוא בעצם True ושקר הוא False:

```
>> type(True)
      <class 'bool'>
```

מחרוזות- בפייתון אין טיפוס מסוג char אך יש מחרוזות.
את המחרוזות בפייתון ניתן להגדיר או במרכאות כפולות(") או במרכאות רגילות('), ואין העדפה בין שתיהן, ובלבד שיהיה אחידות בקוד.
הסיבה שניתן להשתמש בשני סוגי המרכאות הוא בשביל שימוש של אחד מהתווים (" או ') תוך כדי כתיבת מחרוזת ללא שימוש ב-\ ,מה שמכער את הקוד, וכבר ראינו כמה נראות היא ערך עליון במניפסט של פייתון.
כך למשל נוכל לכתוב: "I don't care" , במקום לכתוב 'I don\'t care'.
פייתון היא שפה שרגישה להזחות והורדת שורות, לכן ,בניגוד לג'אווה, הגדרה של מחרוזת רגילה תעשה בשורה אחת, אך ניתן להגדיר גם מחרוזות של כמה שורות עם שלושה מרכאות פותחות ושלושה סוגרות:

```
string= ' ' ' This is going to be a really long string,
way more than the usual ' ' '
print(string)
```

אופרטורים של מחרוזות- למחרוזות יש שני אופרטורים מיוחדים- שרשור וחזקה.
שרשרות מאפשר לנו לחבר בין כמה מחרוזות ולקבל מחרוזת חדשה:

```
>> str1= "Hello"
>> str2="World"

>> str3= str1+str2
>> print(str3)
    "Hello World"
```

חזקה מאפשר לשרשר את המחרוזת לעצמה כמה פעמים:

```
>> a= 'a'
>> print(a*4)
    'aaaa'
```

פונקציות של מחרוזות- מחרוזות הן אובייקט שלא ניתן לשנות ישירות בדומה למחרוזות בשפה C (נראה בהמשך), אך כן ישנן פונקציות של המחלקה str שמשנות את ערך המחרוזות, למשל הפונקציה upper() שמחזירה את אותה מחרוזת באותיות גדולות, או הפונקציה replace שמקבלת שתי תתי מחרוזות ,אחת מתוך המחרוזות המקורית והשנייה מחליפה אותה:

```
>> string= "Hello World"
>> prtint(string.replace("Hello","Bye"))
    "Bye World"
>> print(string.upper())
    "HELLO WORLD"
```

הפונקציות לא מחליפות את המחרוזת המקורית אלא מחזירות מחרוזת חדשה.
ניתן למצוא את הרשימה המלאה של פונקציות המחלקה כאן:
https://www.w3schools.com/python/python_ref_string.asp

F string-
לפעמים נרצה לבנות את המחרוזת שלנו כך שתכיל בתוכה משתנים שהגדרנו קודם לכן.
בשיטה הישנה היינו פשוט משרשרים למחרוזת משתנים:

```
>> name = "Tuna"
>> str = "Hello  " +name+ "."
```

זאת שיטה מעולה אם אין הרבה משתנים מסוגים שונים, אך אם רוצים לערב הרבה משתנים שחלקם מטיפוסי נתונים שונים , יש צורך בשרשור ארוך עם המרות של משתנים, מה שהרבה פעמים לא נראה טוב ולא עולה בקנה אחד עם המניפסט של פייתון.
לכן מפתחי השפה הוסיפו שיטה לכתוב מחרוזות בפורמט נח יותר ע"י המתודה format() שעובדת בצורה דומה ל-printf של c, רק שהמשתנים נכתבים עם סוגרים מסולסלים במקום בתווים כמו '%d' :

```
>> first_name = "Eric"
>> last_name = "Idle"
>> age = 74
>> profession = "comedian"
>> affiliation = "Monty Python"
>> print(("Hello, {first_name} {last_name}. You are {age}. " +
>>        "You are a {profession}. You were a member of {affiliation}.") \
>>        .format(first_name=first_name, last_name=last_name, age=age, \
>>                profession=profession, affiliation=affiliation))
'Hello, Eric Idle. You are 74. You are a comedian. You were a member of Monty Python.'
```

בסוגרים מכריזים על שמות המשתנים שיופיעו, וב- '.format()' מציינים איזה ערך יש לכל משתנה במחרוזת.
כפי שניתן לראות המתודה פתרה כמה בעיות נראות, אך עדיין מוסיפים הרבה קוד מיותר וארוך ,ובאמת החל מפייתון 3.6 נוספה טכניקה חדשה לשפה- fstring, שהיא בדיוק כמו המתודה פורמט, רק שהיא לא מחכה לפרמטרים:

```
>> name = "Eric"
>> age = 74
>> f"Hello, {name}. You are {age}."
'Hello, Eric. You are 74.'
```

עם fstring ניתן גם לבצע פעולות בתוך הגדרת המחרוזת והתוצאה תושם במחרוזת:

```
>> what_is=f"455*698 = {455*698}"
>> print(what_is)
 '455*698 = 317590'
```

ניתן גם להגדיר מחרוזות ארוכות ב-fstring:

```
>> message = f"""
...     Hi {name}.
...     You are a {profession}.
...     You were in {affiliation}.
... """
```

מבני נתונים-

(*הערה*: במסמך זה לא נתעמק בפונקציות של מבני נתונים שונים, זה יעשה במסמך "מבנה נתונים בפייתון".)

מבנה נתונים הוא דרך לאחסון כמות נתונים במשתנה אחד.
בג'אווה יש רק סוג אחד של מבנה נתונים שלא מצריך ייבוא של ספריות מיוחדות והוא מערך.
גם מחרוזת של ג'אווה היא בעצם סוג של מערך של תווים.
לפייתון יש מגוון גדול יותר של מבנה נתונים המגיעים עם השפה, וניתן לחלק אותם לקבוצות:

רצפים- נקראים כך משום שהם רציפים בזיכרון, וכוללים: tuple ו-list ,שהם בעצם סוג של מערך כמו בג'אווה, רק שאפשר להכניס להם ערכים מכמה מטיפוס נתונים בכל אינסטנס של אובייקט.
כך למשל ניתן לבנות רשימה שבנויה מאינטג'רים ומחרוזות למרות שהם מטיפוסי נתונים שונים.
ההבדל העיקרי בין tuple ו-list הוא שרשימה היא mutable כלומר ניתנת לשינוי ו-tuple הוא immutable כלומר איך שהוא מוגדר כך הוא יישאר (נראה בהמשך מה ההבדל המהותי בין השניים).
בשביל ליצור אובייקט מטיפוס רשימה נצטרך לעטוף רצף של אובייקטים שמופרדים ב-',' עם סוגרים מרובעים, וב-tuple האובייקטים עטופים בסוגריים עגולים:

```
>> lst = [1 ,'2' ,3.0]
>> tup =(1 ,'2' ,3.0)
>> type(lst)
     <class 'list'>
>> type(tup)
    <class 'tuple'>
```

לרצפים יש אופרטורים ייחודים להם: [ ] – בשביל לראות ערך ספציפי, ובניגוד למערך בג'אווה אפשר להתחיל מהסוף ע"י הכנסה של ערך שלילי לסוגריים; [:]- בשביל לראות מערך ספציפי עד ערך ספציפי אם לא מגדירים ערך בצד הימני של הנקודתיים הערך הדיפולטיבי הוא עד סוף האוסף, ואם לא מגדירים בצד השמאלי הערך הוא תחילת האוסף; [: :]- בשביל לראות מערך ספציפי עד ערך ספציפי עם קפיצה מסוימת:

```
>> lst=[1,2,3,4,5]
>> lst[0]
    1
>> lst[-1]  #the last cell of the list
  5
>> lst[0:2] #from lst[0] to lst[2]
     [1,2]
>> lst[0::2]
     [1,3,5]
```

מחרוזת היא סוג של tuple, היא כמין tuple רק של מחרוזות בגודל אחד, וכל אופרציה שניתן לבצע על רצפים ניתן לבצע גם במחרוזת, למשל להגיע לתו השלישי : string[2], או לקפוץ בין תווים של המחרוזת: string[0:3:2].
רק לרשימה ניתן להוסיף פריטים חדשים ולהסיר, באמצעות המתודה append() (שמוסיפה איבר לסוף הרשימה), ו-remove() של המחלקה list :

```
>> prime_lst=[1,2,3,5]
>> prime_lst.append(7)
>> prime_lst.remove(1)
>> prime_lst
[ 2 ,3 ,5 ,7 ]
```

sets- סט הוא אוסף של נתונים לא רציפים בזיכרון ולא ממוינים.
כל איבר בסט הוא ייחודי (אין חזרתיות של אברים) וחייב להיות בלתי ניתן לשינוי (immutable),אבל הסט עצמו הוא לא משתנה immutable .
בפייתון מכריזים על סט בדיוק כמו שמכריזים על רשימה או tuple רק עם סוגריים מסולסלות:

```
>> my_set = {1, 2, 3}
```

בכל סט ניתן להכניס מס איברים ככמות הזיכרון, ואין הגבלה על סוג מסוים של טיפוסים ובלבד שיהיו טיפוס immutaple , כלומר לא ניתן להכניס רשימה או מילון, או סט עצמו למשל:

```
>> my_set= {1 ,(2 ,3), '4'}
```

בשביל ליצור סט ריק לא ניתן להשתמש בסוגריים מסולסלים ריקים, כי זאת קריאה למילון ריק, במקום נשתמש בקונסטרקטור ריק של סט:

```
>> a = {}
>> b = set()
>> print(f"a={type(a)} , b= {type(b)}")
     "a=<class 'dict'>, b=<class 'set'>"
```

סטים הם אובייקטים הניתנים לשינוי , אך משום שאינם מסודרים בצורה רציפה אין משמעות לאינדקסים ,לכן לא נוכל לגשת או לשנות איבר ספציפי בסט כפי שהיינו עושים ברשימה.
אך ניתן להוסיף אלמנטים נוספים עם הפונקציה add(), או אם נרצה להוסיף את אלמנטים מתוך רשימה או מסט אחר (או כל אובייקט אוסף mutable) נוכל להשתמש בפונקציה update() :

```
>> my_set = {1}
>> my_set.add((3,2))
>> my_set.update([4,5])
>> my_set
     {4, 5,(3,2),1}
```

מילון- מילון הוא אוסף לא רציף של נתונים, שערכיו מסודרים לפי מפתח בערך.
בג'אווה מילון מוכר כטבלת גיבוב (hash table).
המילון הוא אופטימלי להחזרת ערכים בסיבוכיות נמוכה כאשר ידוע המפתחות שלהם.
הכרזה על מילון דומה להכרזה על סט, אך כל איבר במילון בנוי משני חלקים, חלק ראשון הוא המפתח- איזשהו משתנה (שחייב להיות מטיפוס נתונים ממשפחת ה-immutable), נקודתיים ומשתנה "ערך" שיכול להיות מכל טיפוס שהוא:

```
>> my_dict= {"some key": "some value", 'other key': 1, (1,2): "tuple key"}
```

כדי לקבל ערך מסוים מהמילון נשתמש האופרטור '[]' , ונכניס לתוכו את הפתח:

```
>> val = my_dict["some key"]
>> val
'some value'
```

שינוי ערכים יהיה כמו ברשימות רק שבמקום אינדקס מכניסים את המפתח:

```
>> my_dict['other key'] = 2
```

על כל אחד ממבנה הנתונים ניתן להשתמש בפונקציה `len()` בשביל לקבל את אורך האוסף (כמה אלמנטים יש לו):

```
>> my_dict = {"some key":1, 'other key': 2"}
>> my_set = {1 ,2}
>> my_str = "1,2"
>> my_lst = [1,2]
>> my_tup = (1,2)
>> message = f"""
...     {len(my_dict)},
...     {len(my_lst)},
```

```
...     {len(my_str)},
...     {len(my_tup)},
...     {len(my_set)},
... """
>>print(masseg)
2,2,3,2,2
```

-NoneType

ברוב שפות התכנות יש משתנה הנקרא null, והוא מציין לרוב מצביע למקום לא מוגדר בזיכרון, כלומר אובייקט שעדיין לא הוגדר, והוא בעצם המיקום ה-0 בזיכרון.
בפייתון לעומת זאת משתמשים באובייקט מסוג None ולא null שמשמש למטרה זהה אך הוא שונה מהותית מ-null.
None הוא טיפוס נתונים בפני עצמו, כלומר הוא מחלקה הבאה עם השפה:

```
>> x = None
>> type(x)
<class 'NoneType>
```

משתנים mutable ו- immutable-

יש שתי משפחות של אובייקטים בפייתון- mutable ו- immutable.
כל משפחה מגדירה האם האובייקט יכול להשתנות(mutable), או שהוא אובייקט קבוע מרגע שהוא נוצר (immutable)
מה הכוונה? כאשר יוצרים משתנה חדש בפייתון בעצם יוצרים מצביע על מקום מסוים בזיכרון, אובייקט immutable הוא אובייקט שכל פעם שנשים אותו כערך למשתנה ונבצע על המשתנה פעולה כלשהי המשתנה יצביע למקום חדש בזיכרון.
באמצעות הפונקציה id() נוכל לראות להיכן בזיכרון מצביע המשתנה:

```
>> x = 1
>> id(x)
     8971453144736
>> x+=1
>> id(x)
     8971453144768
```

ניתן לראות בדוגמא שכאשר הוספנו +1 למשתנה הוא שינה את המיקום בזיכרון אליו הוא הצביע.
שאלה: מה יהיה הפלט של הקוד הבא?

```
>> str1 = "String"
>> str2 = 'String'
>> str1 is str2
```

אובייקט mutable לעומת זאת הוא אובייקט ששומר על המיקום שלו בזיכרון גם לאחר שבוצעה עליו פעולה כלשהי:

```
>> lst = [1,2,3]
>> id(lst)
     4601088
>> lst.append(4)
>> id(lst)
     4601088
```

ניתן לראות כאן שהמצביע לרשימה עדיין מצביע לאותו מיקום, למרות שהרשימה הוסיפה אלמנט חדש.
עוד משהו שיש לאוסף ממשפחת mutable הוא היכולת לשנות אובייקט ספציפי מתוך אוסף האובייקטים, למשל ברשימה בדוגמא לעיל אם נרצה לשנות תא ספציפי, למשל lst[2], השינוי יתבצע ולא נקבל שגיאה, אך באוספים שהם immutable

כמו מחרוזת או tuple לא ניתן לשנות תא ספציפי.
שאלה למחשבה: האם ניתן ליצור אובייקט פרימיטיבי (כמו int, float, str וכו') שיהיה mutable?

Casting והמרות- לפעמים נרצה לשנות בין טיפוסי נתונים של משתנים, למשל קיבלנו מחרוזת של מספרים, ואנחנו רוצים לבצע פעולות אריתמטיות עליהם.
בג'אווה כאשר נרצה לשנות בין טיפוסי נתונים נעשה זאת באמצעות casting, שבה אנחנו מגדירים בסוגרים את טיפוס הנתונים אותו אנחנו רוצים לפני המשתנה שנרצה לשנות:

```
public class Main {
  public static void main(String[] args) {
    double myDouble = 9.78;
    int myInt = (int) myDouble; // Manual casting: double to int
    System.out.println(myDouble);   // Outputs 9.78
    System.out.println(myInt);      // Outputs 9
  }
}
```

דרך נוספת היא ע"י השמה של טיפוס נתונים פשוט יותר בתוך טיפוס מורכב, כך שהמשתנה הפשוט יוסיף על עצמו עוד כדי להגיע לרמה מעל.
ההיררכיה של המשתנים היא: byte->short->char->int->long->float->double

```
public class Main {
 public static void main(String[] args) {
   int myInt = 9;
   double myDouble = myInt; // Automatic casting: int to double
   System.out.println(myInt);      // Outputs 9
   System.out.println(myDouble);   // Outputs 9.0
 }
}
```

אם המשתנים הם לא מההיררכיה שציינו לעיל , אז צריך להשתמש בפונקציות מיוחדת כדי להמיר בין טיפוסי משתנים, למשל כדי להפוך מחרוזת למספר צריך להשתמש בפונקציה Parse של מחלקת Integer.
פייתון שונה קצת,
משום שפייתון שפה מונחת עצמים שלימה לאובייקטים מטיפוס מחרוזת ומספרים ניתן לעשות המרות אחת לשנייה ע"י הבנאי של המחלקה, כך למשל אם נרצה לעשות המרה בין int ל-float נשתמש בקונסטרקטור של float על המשתנה:

```
>> x = 3
>> y = str(x)
>> z = float(y)
>> print ('x=',x,'type(y)',type(y),'z=',z)
      x=3 type(y)=<class 'str'> z=3.0
>> complex(z)
   <3+0j>
```

כמובן שכדי לעשות המרה ממחרוזת למספר, על המחרוזת להיות בפורמט שניתן להפוך אותו למספר.
כמו כן ניתן לבצע המרה בין משתנים שהם באותו מעמד, למשל ניתן לבצע המרה בין מחרוזת לרשימה ,או בין רשימה ל-tuple וכו', אבל לא ניתן לבצע המרה בין map לרשימה או מילון וכו' .
עבור אובייקטים ל\מטיפוס שהם לא מאותו מעמד (למעט מחרוזת ומספרים) נצטרך להשתמש בפונקציות עזר.
הערה: המרה בין רשימה למחרוזת לא תמיד מניבה את התוצאה הרצויה, מומלץ להשתמש בזה להמרה בניהם:
https://www.geeksforgeeks.org/python-convert-list-characters-string/?ref=lbp

המרה למשתנה בוליאני- כל אובייקט ניתן להמיר למשתנה בוליאני והערך יהיה True, למעט מקרים של אוספים ריקים, אפס או משתנה שערכו None:

```
>> my_dict={}
>> my_list=[]
>> my_str=""
>> my_tup=()
>> my_set=set()
>> message = f"""
...     {bool(my_dict)},
...     {bool(my_list)},
...     {bool(my_str)},
...     {bool(my_tup)},
...     {bool(my_set)},
...     {bool(None)},
...     {bool(0)}
... """
>>print(massege)
False,False,False,False,False,False,False
```

בקרת זרימה בפייתון-

כברירת מחדל מחשב מבצע פקודות לפי סדר כתיבתן, אך לפעמים נרצה לשנות את אופן הפעלת הפקודות שיהיו יותר מבוקרות, למשל נרצה לבצע פעולה מסוימת מספר פעמים, או לבצע פעולה בתנאי שגם פעולה אחת התבצעה כראוי.
בג'אווה יש שני סוגים של בקרות זרימה: תנאים ולולאות, בפייתון בנוסף לשני הסוגים האלה יש גם פונקציות callback, אבל על כך בנושא נפרד.

תנאים- בשפות תכנות פקודת תנאי שלרוב נקראת פקודת if היא פקודה שמתבצעת אם ורק אם מתקיים(או לא) תנאי מסוים.
בג'אווה הפקודה if היא כמו פונקציה שמקבלת ערך בוליאני ומבצעת את הקטע קוד המגיע לאחר מכן שסגור בבלוק ('{}'),או השורה שמופיע בדיוק לאחר הפקודה.
את הפקודה הרבה פעמים ניתן לסייג לכמה תנאים שמתחלקים לפונקציה else ולסוג של פונקציה מקוננת else if, else מתבצעת במקרה בו לא מתבצע התנאי ונרצה שיתבצע משהו אחר כברירת מחדל, ו-else if היא הוספת תנאי נוסף:

```
if (condition1)
{
  Statement1;
}
else if(condition2)
{
 statment2
}
else
{
  StatementDefault;
}
```

בפייתון הסינטקס דיי דומה.
if נכתב באופן דומה רק שהביטוי שלפיו אנחנו מבצעים את הבלוק מתקבל ישר(ולא כמו פרמטר לפונקציה), אחריו המבנה יהיה זהה למבנה של בלוק בפייתון- נקודתיים, ואז פקודה חדשה ברווח מתחילת ההתניה:

```
>> if 3 > 2:
. . print("3 > 2")
'3 > 2'
```

גם else של פייתון זהה לשל ג'אווה רק בסינטקס של פייתון, וה-else if נכתב כ- elif ומשם כמו ב-if רגיל:

```
>>> if a > 10:
...  print("a>10")
... elif a < 10:
...  print("a<10")
... else:
...  print("a==10")
```

יש גם syntactic sugar לכתיבת התניה מקוצרת של if ו- else בלבד:

```
>> a = 10 if a>10 else 0:
```

a יקבל את הערך 10 אם הוא גדול מעשר אחרת הוא יקבל 0.
בג'אווה יש גם אפשרות לביצוע התניה עם switch ו-cases , בפייתון **אין** אפשרות כזאת.
שאלה למחשבה: האם ניתן לממש התנית switch ו- cases בפייתון או לפחות למצוא לה תחליף ראוי?

לולאות- לפעמים נרצה לבצע בלוק קוד מסוים כמה פעמים, וכמתכנתים טובים תמיד נשאף לבצע את הדרך הקלה והפשוטה ביותר שנוכל.
לולאה מאפשרת לבצע את אותו הקטע קוד בכמה איטרציות.
כמו כל דבר בחיים, או לפחות במסמך זה, גם לולאות מתחלקות לכמה סוגים או יותר נכון לשניים:
לולאות עם מספר איטרציות לא מוגדר- הלולאה תבצעה כל זמן שלא הוגדר אחרת.
ולולאות עם מספר סופי של איטרציות.

לולאות עם מספר איטרציות לא מוגדר מוכרות גם בשם לולאות while – כל זמן שלא שתקיים התנאי המסוים תבצע את קטע הקוד שמופיע מתחת.
גם לג'אווה וגם לפייתון האיטרציות פועלות בצורה דומה, והשוני הוא כמו השינוי בין if של השפות, כלומר בג'אווה נכתוב את שם הפקודה while ואח"כ בסוגרים את התנאי של הלולאה ולמטה בלוק עם הפעולה שאמורה להתבצע.
ובפייתון נכתוב while, התנאי (בלי סוגריים) ובלוק שיתבצע באיטרציות:

```
//while loop in java
while(x>2)
{
   x++;
}

#while loop in python
>>> while x>2:
...  x+=1
```

לולאת while תמיד מתקדמת לפי תנאי בוליאני ולכן נוכל לבצע ביטויים כגון:

```
>>> a = [1,2,3]
>>> while a:
```

```
...  print(a.pop(-1))
3
2
1
```

pop() היא פונקציה של רשימות, והיא מוציאה את האיבר האחרון מהרשימה, וכפי שכבר ראינו רשימה ריקה נחשבת כ-False.

בפייתון ובג'אווה ללולאות יש שתי מילות מפתח שמסיימות את האיטרציה של הלולאה(או את כל הלולאה) לפני הזמן, והן continue ו- break .

* continue- מסיימת את האיטרציה הנוכחית של הלולאה ,ומקפיצה יש לראש הלולאה מבלי לבצע את הפקודות הבאות לאחר פקודת continue.

* break- מסיימת את הלולאה לחלוטין.

```
>>> n = 5
>>> while n>0:
...  n-=1
...  if n==3:
...   continue
...  if n==1:
...   break
...  print(n)
4
2
```

בפייתון אפשר להשתמש ב-else אחרי לולאות while, מה שמאפשר ביצוע של סדר פקודות שיבואו בתום הלולאה:

```
>>> num1 = 1
>>> num 2 = 0
>>> n = 5
>>> while n>5:
...  temp = num1
...  num1+=num2
...  num2=temp
...  #print(num2)
...  n-=1
... else:
...  num1=1
...  num2=0
>>> print(f"num1={num1},num2={num2}")
'num1=1,num2=0'
```

שאלה: מה היה מודפס לו לא היינו מסמנים את print(num2)?

מה בעצם מוסיף לנו ה"פיצ'ר" הזה? הרי בכל מקרה אחרי הלולאה היו מתבצעים הפקודות בשורות הבאות אחריה.

אז זהו שהצהרת הelse מאפשרת לבצע את הפעולות המוגדרות בבלוק שלה רק אם הלולאה נגמרה בצורה טבעית, כלומר אם התבצעה break במהלך הלולאה, המפרש לא יבצע את השורות של הפקודה else:

```
>>> n = 5
>>> while n>5:
...  n-=1
...  if n==2:
...   break
... else:
...  n=10
```

```
>>> print(n)
2
```

[לחלק של הפונקציות: פונקציות בפייתון, שלא כמו בג'אווה תמיד מחזירות ערך. בעוד פונקציית void של ג'אווה לא מחזירות ערך בכלל, פונקציות שלא משתמשות ב-return מחזירות מאחורי הקלעים ערך None.
לדוגמא לזה ניתן לראות מהפונקציה print() שמדפיסה למסך, אך לא מחזירה ערך לכאורה:

```
>> print(print("null function?"))
   null function?
   None.
```

https://realpython.com/null-in-python/

לחלק של המונחה עצמים: פונציות מיוחדות של מחלקות כמו __str__ ומימוש אופרטורים
בטיפול בשגיאות לא לשכוח להוסיף על with]