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.
```

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 בasciii 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 ו-bitwise .

אופרטורים אריתמטיים-
אופרטורים אריתמטיים הם כל אותם אופרטורים שאנחנו משתמשים בהם להגדרת פעולות מתמטיות יסודיות.
בג'אווה יש אנחנו מכירים 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 יותר הרבה יותר 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, ומקבלים ערך "אמת" או "שקר" במידה והם נכונים.
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 ומחזריה 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 לשלול את 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-