# מג'אווה לפייתון

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 if \_\_name\_\_ == "\_\_main\_\_" :

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](https://www.python.org/dev/peps/pep-0008/#maximum-line-length)**" 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 עם אותו השם, אבל ממחלקות שונות.   
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 הפונקציה,  
לצורך הדוגמא נניח שהמודול cow מכיל רק פונקציה אחת והיא moo שמדפיסה למך את המילה moo :**

**#cow.py  
>>> def moo():  
... print('moo')**

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

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

**לפעמים יהיו modules גדולים שנרצה מהם רק פונקציה ספציפית, למשל מהספרייה math נרצה להשתמש רק במשתנה pi, ולא בכל הפונקציות בספרייה.  
במקרה כזה נוכל להגדיר אילו אובייקטים ספציפיים לקחת מהספרייה עם המילים השמורות from ו- import :**

**>>> from math import pi**

**במקרה כזה במשתנה pi יהפך להיות משתנה גלובלי בפרוייקט הנוכחי שלנו, ולא נכיר אותו כmath.pi כפי שהיה קורה במקרה בו היינו מייבאים את כל הספרייה:**

>>> from math import pi

>>> pi

3.141592653589793

>>> math.pi

NameError: name 'math' is not defined

נוכל גם להגדיר את השימוש בספריות או אובקיטים של הספרייה במרחב שם חדש ע"י שימוש במילה as:

>>> import math as m

>>> m.pi

3.141592653589793

>>> from math import pi as PI

>>> PI

3.141592653589793

***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(והbitwise-).   
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.  
*הערה:* בבניית מחלקה חדשה, במידה ולא הגדרנו את האופרטור של המחלקה: '\_\_eq\_\_', אז האופרטור '==' יתפרש באותו אופן כמו האופרטור 'is'.  
  
*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.   
הbitwise 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: *, כי רק במקום 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 :* .  
  
not- *מסומן ב- '~' והוא מסמן שלילת הביטוי, כלומר כל מה שערכו אחד יהפוך להיות אפס וכנ"ל אפס יהפוך לאחד,  
למשל: .*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:*.   
  
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: *.*shift right- *שמסומן ב-'<<' ,והוא מעתיק פעמיים את הביט השמאלי ביותר ומוריד את הביט הימני ביותר מהביטוי, למשל:.  
וכפי שאמרנו קודם, לג'אווה יש אופרטור נוסף שאין לפייתון והוא האופרטור '<<<', שדומה לshift left רק בכיוון ההפוך, למשל:   
 .  
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}.

... """

וגם באיזה פורמט להציג את המשתנים:

>>> val = 12.3

>>> print(f'{val:.2f}')

12.30

>>> print(f'{val:.5f}')

12.30000

>>>

>>> a = 300

>>> # hexadecimal

>>> print(f"{a:x}")

12c

>>> # octal

>>> print(f"{a:o}")

454

>>> # scientific

>>> print(f"{a:e}")

3.000000e+02

***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 mutableכלומר ניתנת לשינוי וtuple- 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** (*condition*1)

{

*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 ה else 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

*שאלה למחשבה:* בפייתון אין לולאת do while , האם ניתן לממש בצורה יעילה לולאה כזאת או לפחות שיטה לסמלץ אותה?

לולאות עם מספר סופי של איטרציות- מוכרות גם בשם לולאות for.   
בג'אווה יש שני סוגים של לולאות for, הראשונה היא לולאה שבנויה משלושה חלקים: חלק אחד נקודות התחלה ,חלק שני תנאי להתקדמות, חלק שלישי התקדמות בכל איטרציה:

for(int i=0; i!=arr.length; ++i)  
{  
//code..  
}

סוג שני של לולאות הוא כמין syntactic sugar של הסוג הראשון, והוא לולאת for each , שעוברת באיטרציה על כל משתנה מתוך אובייקט שבנוי איטרציות (ומוכר גם כמשתנה iterable), בדר"כ זה אוספים כמו מערך, סט וכו'.  
 הלולאה בנויה משני חלקים: הגדרת שם לכל משתנה מתוך האוסף, נקודותיים ושם האוסף:

for(int elem: arr){//...

פייתון שונה מג'אווה במקרה זה.  
בפייתון כל לולאת for היא לולאת for each בעצם,   
מה הכוונה? כל פעם שאנחנו מבצעים איטרציה המוגבלת במספר מסוים, צריך להגדיר מתוך איזה אוסף, או יותר מדויק משתנה iterable, אנחנו מבצעים את הלולאה, למשל כשרוצים לעבור על כל איבר ברשימה אנחנו משתמשים באופרטור in כדי להכריז על הגדרה של הלולאה:

>>> arr = [1,'2',(3,4)]  
>>> for elem in arr:  
... print(elem)  
1  
'2'  
(3,4)

אם כך איך ניתן לבצע לולאה קלאסית כמו בג'אווה?   
אז זהו, כדי ליצור לולאה של טווח מסויים נצטרך 'ליצור' אובייקט iterable של אותו טווח.  
עכשיו לכאורה יכולנו לחשוב, רגע אז כל פעם שנרצה ליצור אובייקט של טווח מסוים נצטרך להגדיר רשימה שאיבריה הם המספרים של הטווח, זה לא הגיוני! עדיף כבר להשתמש בלולאת while עם משתנה חיצוני שעולה באחד כל פעם!  
ישנה דרך נוחה, אפילו יותר מלולאות בג'אווה: לפייתון יש פונקציה שנקראת range() שמחזירה אובייקט שניתן לעבור עליו באיטרציה (לא אוסף), של מספרים מסוים.   
כברירת מחדל חייב לשלוח לפונקציה ארגומנט אחד של הטווח(נקודת עצירה), אך ניתן להוסיף לו גם ארגומנט של נקודת התחלה, וגם משתנה של קפיצות בין איטרציה לאיטרציה, למשל נניח שאנחנו רוצים לקפוץ שלושה מספרים כל איטרציה במקום קפיצות של מספר אחד.  
סדר הפרמטרים בפונקציה הוא קודם כל נקודת ההתחלה, הטווח וקפיצות:

>>> for n in range(10): #n=0 -> n=9  
... #...  
>>> for n in range(2,10): #n=2 -> n=9  
... #...  
>>> for n in range(2,10,2): #n=2-> n=8 iff n%2==0  
... #...

ניתן גם להציג טווח הפוך:

>>> for n in range(10,1,-1):  
... # n=10 -> n=1

גם בלולאת for ניתן להשתמש ב-break,continue וגם לולאה יש else בדיוק כמו בלולאת while.

פונקציות-

בחתימה של פונקציות ג'אווה יש ארבעה חלקים: סוג הפונקציה(פרטית, ציבורית..), ערך החזרה, שם הפונקציה ופרמטרים, ואח"כ יבוא בלוק הקוד של הפונקציה.   
מתי שהפונקציה מוגדרת void אין ערך חזרה, וכאשר מגדירים ערך חזרה ספציפי מגדירים אותו בראש הפונקציה כך:

public static void print\_upper(String str) //The function doesn't return value  
{  
 System.out.print(str.toUpperCase());  
}  
  
public static String return\_upper(String str) //Returns String   
{  
 upper\_str= str.toUpperCase();  
 return upper\_str;  
}

כמו כן ,היות וכל מסמך בג'אווה הוא מחלקה, אז ניתן להגדיר אם הפונקציה היא פרטית או ציבורית, כלומר האם אפשר להשתמש בפונקציה גם ב-modules חיצוניים, או רק במחלקה עצמה.  
כפי שכבר יצא לנו לראות קובץ פייתון שונה ממבנה קובץ ג'אווה, וניתן להגדיר פונקציות ומשתנים מחוץ למסגרת המחלקה, בדומה לפונקציות בשפות תכנות כמו c++.  
חתימה של פונקציה בפייתון מחולקת לשלושה חלקים, שהם בעצם שתיים: החלק הראשון הוא להגדיר שעכשיו אנחנו כותבים פונקציה עם-def (כלומר הפונקציה defined), כמו שצריך להגדיר על לולאות ותנאים, גם כאן צריך להגדיר למתכנתים אחרים שהבלוק הבא שיבוא הוא פונקציה; החלק השני הוא שם הפונקציה, והשלישי והאחרון הוא הפרמטרים של הפונקציה, אח"כ יבוא הבלוק שבו תוגדר הפונקציה:

>>> def print\_upper\_str(string):  
 print(string.upper())

כשרוצים להחזיר ערך מוגדר מהפונקציה גם משתמשים במילה השמורה return:

>>> der return\_upper\_str(string):  
 return string.upper()  
>>> print(return\_upper\_str("what the fuction returns?"))  
WHAT THE FUNCTION RETURNS?

אבל בניגוד לג'אווה פונקציות **תמיד** מחזירות ערך ו-return רק מגדיר מה הערך המוחזר, במידה ולא הגדרנו הערך המוחזר יהיה אובייקט מטיפוס None:

>>> print(print\_upper\_str("what is the value of the function?"))  
WHAT IS THE VALUE OF THE FUNCTION?  
None

***פרמטרים וארגומנטים-***הרבה מתכנתים מתבלבלים אז כדי לישר קו: מבחינת טרמינולוגיה כשמדברים על פרמטרים מתכוונים לאילו ערכים הפונקציה מקבלת, בדוגמא למעלה הפונקציה מקבלת משתנה שקוראים לו string, וצריכה להיות לו פונקציה שקוראים לה upper() (פייתון לא מחייבת לשלוח דווקא משתנה מטיפוס מחרוזת העיקר שיהיה לאובייקט את הפונקציה).   
כשמדברים על ארגומנטים מתכוונים לערך שאנחנו שולחים לפונקציה מתוך התוכנית, למשל בדוגמא למעלה הארגומנט שנשלח לפונקציה print\_upper\_str() הוא המחרוזת :"what is the value of the function".  
כששולחים ארגומנטים לפונקציה, הפונקציה תקבל אותם לפי סדר הפרמטרים שכתובים בחתימתה, למשל:

>>>def func(A ,B, C):  
... print(f'A is {A}, B is {B}, C is {C}')  
>>> func(12 , 13, 14)  
A is 12, B is 13, C is 14

משום ששלחנו קודם את 12 כארגומנט לפונקציה, הפונקציה שמה את הערך הראשון לפרמטר A , והארגומנט השני לפרמטר B וכו'.  
אבל ניתן לשלוח את הארגומנטים לא לפי סדר הופעתן בחתימת הפונקציה, אם נגדיר לאיזה פרמטר אנחנו שולחים איזה ערך בקריאה לפונקציה:

>>> func(C=14,A=12,B=13)  
A is 12, B is 13, C is 14

בדומה ל c++ , גם בפייתון אפשר לתת ערך דיפולטיפי לפרמטרים במידה והמשתמש לא שלח ערך לארגומנטים של הפונקציה:

def func2(A ,B = 3, C = 14):  
... print(f'A is {A}, B is {B}, C is {C}')  
>>> func2(12 , 13)  
A is 12, B is 13, C is 14  
>>> func2(13,C=22)  
A is 12, B is 3, C is 22  
>>> #func2(C=12,13,B=12)->error, A must be the first argument

כשהפרמטר הדיפולטיבי של הפונקציה הוא משתנה immutable אין לנו בעיה, אבל כשהערך הוא טיפוס mutable זה כבר יותר משונה, ניקח למשל את הפונקציה הבאה:

>>> def stam\_list(my\_list=[]):  
. . . my\_list.append('###')  
. . . return my\_list  
>>> stam\_list()  
['###']

הערך הדיפולטיבי של הפונקציה הוא רשימה ריקה, לכאורה עד כאן הכל כפי שציפינו, אבל מה יקרה אם נפעיל את הפונקציה שוב באופן ריק:

>>> stam\_list()  
['###','###']  
>>> stam\_list()  
['###','###','###']

רגע אבל הגדרנו את הערך הדיפולטיבי להיות רשימה ריקה, איך זה הגיוני שאנחנו מוספים עוד ערכים לאותה רשימה? זה לא מה שרצינו שיקרה!  
זה משום שערכים דיפולטיבים בפייתון נקבעים **רק פעם אחת** כשהפונקציה מוגדרת.  
בעצם מה שאנחנו מגדירים מאחורי הקלעים זה מצביע למשתנה בטיפוס הדיפולטיבי שאנחנו מצפים שיהיה.   
כך למשל בפונקציה func2() הגדרנו מצביעים לטיפוס מסוג int כפרמטרים ,ולכן הערך הדיפולטיבי של המצביעים לא ישתנה, כי int הוא משתנה immutable, אבל במקרה של ערך דיפולטיבי לרשימה או לאובייקט mutable הערך של הפרמטר כן ישתנה:

>>> def stam\_list(my\_list=[]):  
. . . print(id(list.append))  
. . . my\_list.append('###')  
. . . return my\_list  
>>> stam\_list()  
41305130  
['#']  
>>> stam\_list()  
41305130  
['#','#']

אז איך היינו פותרים את הבעיה הזאת? כיצד בכל זאת ניתן להגדיר רשימה ריקה כפרמטר?  
נצטרך להגדיר את הדיפולט של האובייקט כ-None שהוא משתנה immutable, ואז פשוט נבצע בדיקה אם הפרמטר שלנו הוא None, אם כן אז נשנה את המצביע לאובייקט מטיפוס רשימה ריקה:

>>> def stam\_list(my\_list=None):  
. . . if my\_list==None:  
. . . my\_list=[]  
. . . my\_list.append('###')  
. . . return my\_list  
>>> stam\_list()  
['#']  
>>> stam\_list()  
['#']

עוד הבדל מהותי בין פייתון לג'אווה הוא בכמות המשתנים שניתן להחזיר מפונקציה- בג'אווה ניתן להחזיר רק אובייקט אחד ופייתון ניתן להחזיר בלי הגבלה, וההשמה של האובייקטים תהיה לפי הסדר שהפונקציה מחזירה:

>>> def get\_str\_int\_lst():  
... return "my string", 1, ['my string',1]  
...  
>>> my\_str,my\_int,my\_list = get\_str\_int\_lst()  
>>> my\_str  
'my string'  
>>> my\_int  
1  
>>> my\_list  
['my string', 1]

בעצם מה שהפונקציה מחזירה הוא tuple עם שלושה אובייקטים שונים, לכן יש ',' בין כל משתנה שחוזר מהפונקציה, באותו האופן היה ניתן להחזיר אותם כ- ("my string", 1, ['my string',1]).

***סקופ של משתנים בפונקציה-****שאלה:* מה יודפס בפונקציה הבאה?

>>> x = 10  
>>> def f(x):  
... x+=1  
... print(f" x is {x}, and its id is{id(x)}")  
>>> f(x)  
>>> print(f" x is {x}, and its id is{id(x)}")

תשובה:

x is 11, and its id is 8791421687744  
...  
x is 10, and its id is 8791421687776

אז בעצם כשאנו שולחים אובייקט הוא לא משנה את ערכו? האם אנחנו באמת שולחים העתק של האובייקט או את האובייקט עצמו?

>>> def f2(x):  
... print(f"x is {x}, and its id is{id(x)}")  
>>> f2(x)  
x is 10, and its id is 8791421687776  
>>> print(f" x is {x}, and its id is{id(x)}"))  
x is 10, and its id is 8791421687776

הפונקציה מעתיקה את הערכים שקיבלה מהארגומנט לתוך מצביע חדש, ואז שני המצביעים(זה של הפונקציה והמשתנה המקורי) באמת פונים לאותו מקום בזיכרון, וכנ"ל אם x היה שווה אובייקט מטיפוס mutable - אם הינו משנים את ערכו של x במהלך הפונקציה, השינוי היה ניכר רק במצביע של הפונקציה ולא באובייקט שמחוץ לו.  
אבל אם היינו מפעילים מתודה של אובייקט mutable בפונציה ,למשל append() של רשימות, אז השינוי היה ניכר גם בפונקציה וגם מחוץ לה:

>>> my\_lst =[0]  
>>> def null\_lst(lst):  
... lst=None  
...  
>>> def add\_one\_lst(lst):  
... lst.append(1)  
...  
>>> def change\_first\_cell(lst, val):  
... lst[0]=val  
...  
>>> null\_lst(lst)  
>>> lst  
[]  
>>> add\_one\_lst(lst)  
>>> lst  
[1]  
>>> change\_first\_cell(lst,0)  
>>> lst  
[0]

לכאורה היינו מצפים שהפונקציה לא תשנה את האובייקט, כפי שהיא לא שינתה אותו לNone בפונקציה הראשונה,  
אבל בפונקציה השנייה והשלישית היות והשינוי הוא בזיכרון, כלומר באובייקט עצמו, ולא שינוי של המצביע, אז הוא ניכר גם מחוץ לפונקציה.  
  
 האם אנחנו יכולים להשתמש במשתנה גלובאלי (משתנה שלא מוגבל בבלוק מסוים) בתוך פונקציה וגם לשנות אותו?  
לכאורה ניתן לגשת לאובייקט גלובלי גם מחוץ לפונקציה:

>>> def f3():  
... print(f"x is {x}, and its id is{id(x)}")  
...  
>>> f3()  
x is 10, and its id is 8791421687776  
>>> print(f" x is {x}, and its id is{id(x)}"))  
x is 10, and its id is 8791421687776

אבל האם ניתן גם לשנות אותו? אם ננסה בשיטה הנאיבית נגלה שהשינוי לא יהיה כמבוקש:

>>> def f4():  
... x=11  
... print(f"x is {x}, and its id is{id(x)}")  
...  
>>> f4()  
x is 11, and its id is 8791421680014  
>>> print(f" x is {x}, and its id is{id(x)}"))  
x is 10, and its id is 8791421687776

וזה הגיוני סהכ כי הפונקציה יוצרת משתנה חדש בשם x שמתאים רק לסקופ שלה וערכו 11   
ואם במקום לשים ערך ב-x היינו מנסים לשנות את ערכו בפעולה אריתמטית היינו מקבלים שגיאה:

>>> def f5():  
... x+=1  
... print(f"x is {x}, and its id is{id(x)}")  
...  
>>> f5()  
Traceback (most recent call last):  
 File "<stdin>", line 1, in <module>  
 File "<stdin>", line 2, in f  
UnboundLocalError: local variable 'x' referenced before assignment

אז כדי להשתמש במשתנים גלובליים בתוך הסקופ של הפונקציה צריך להשתמש במילה השמורה global ואז להצהיר איזה משתנה אנחנו מנסים לשנות:

>>> def f5():  
... global x  
... x=0  
... x+=1  
... print(f"x is {x}, and its id is{id(x)}")  
...  
>>> f3()  
x is 1, and its id is 87914216874341  
>>> print(f" x is {x}, and its id is{id(x)}"))  
x is 1, and its id is 87914216874341

***פרמטר עם כוכבית-***   
נניח אנחנו רוצים לבנות פונקציה שמחשבת ממוצע של כמה מספרים.   
אינטואיטיבית נדרוש שהפונקציה תקבל את הארגומנטים ואז תחלק את סכומם במספר הארגומנטים:

>>> def avg(a=0,b=0,c=0):  
... sum= a+b+c  
... return sum/3

קל לראות שאנחנו ניתקל בבעיה כאשר המשתמש לא יכניס את אחד הארגומנטים, כי אז הסכום יהיה בין שני מספרים, ואנחנו מבצעים ממוצע כאילו היו שלושה מספרים.   
ואפילו יותר מזה אם נרצה לבצע ממוצע עם יותר משלושה לא נוכל.  
הדרך הקלה לטפל בבעיה היא ע"י שליחה של רשימה או tuple , ואז לחלק את סכום האיברים של האוסף באורך האוסף:

>>> def avg(nums):  
... sum = 0  
... for num in nums:  
... sum += num  
... return sum/len(nums)

הבעיה שאנחנו מצפים מהמשתמש להבין שהוא צריך לשלוח רשימה או tuple וזה לא ממש אינטואיטיבי.   
במקרה כזה נוכל להשתמש בפרמטר כוכבית- כשלפני שם של פרמטר מופיע התו '\*' אנחנו יכולים לדעת שהמשתנה עוטף בתוכו כמה משתנים לתוך tuple, כלומר הארגומנטים שנשלח יתאספו לאובייקט מטיפוס tuple ,ואז נוכל להפעיל עליו פונקציות של tuple:

>>> def f(\*args):  
... print(args)  
... print(type(args), len(args))  
... for x in args:  
... print(x)  
...   
>>> f(1, 2, 3)  
(1, 2, 3)   
<class 'tuple'> 3  
1  
2  
3

ולכן המימוש של פונקציית הממוצע יראה כך :

>>> def avg(\*nums):  
... sum = 0  
... for num in nums:  
... sum += num  
... return sum/len(nums)

למי שיצא לתכנת בשפת c או c++ בטח מבין מה הרעיון , זה מזכיר מאוד שליחה של מצביע לפונקציה, אבל כאן בדיוק נכנס החלק המוזר (בערך), ניתן להשתמש באופרטור כוכבית גם לארגומנטים, במקרה כזה הפעולה שתבצעה תהיה הפוכה, כלומר הפונקציה תפרק את המשתנים שהתקבלו מהאוסף שלהם:

>>> def sum\_of\_three(num1,num2,num3):  
... sum = num1 + num2 + num3  
... return sum  
...  
>>> tup\_int=(1,2,3)  
>>> sum\_of\_three(\*tup\_int)  
6  
>>> list\_int = [4,5,6]  
>>> sum\_of\_three(\*list\_int)  
15

טוב זה לא לגמרי מוזר, כי גם ב-c כששולחים מצביע לפונקציה צריך להגדיר את טיפוס הפרמטר ב-\*, וכדי להשתמש בערך של מצביע אפשר להשתמש ב-\* (לרוב משתמשים באופרטור '<-'), פשוט בפייתון לא מגדירים את טיפוס המשתנה בהכרזתו אז זה נראה מעט מוזר.   
עוד משהו מעניין בקשר לפירוק של המשתנים, הוא נעשה גם אם מפרקים כמה משתנים שכל אחד הוא אוסף בפני עצמו:

>>> def my\_sum(\*args):  
... result = 0  
... for x in args:  
... result += x  
... return result  
...  
>>> list1 = [1, 2, 3]  
>>> list2 = [4, 5]  
>>> list3 = [6, 7, 8, 9]  
>>> print(my\_sum(\*list1, \*list2, \*list3))  
45

ואם זה לא מספיק, ניתן לפרק לשלושה חלקים עם האופרטור:

>>> my\_list = (1, 2, 3, 4, 5, 6)  
>>> a, \*b, c = my\_list  
>>>  
>>> print(a)  
1  
>>> print(b)  
[2, 3, 4, 5]  
>>> print(c)  
6

כמו שקיים אופרטור כוכבית קיים גם אופרטור של שתי כוכביות.  
אופרטור של שתי כוכביות דומה לאופרטור כוכבית, רק שהוא עוטף את הארגומנטים במילון במקום ב-tuple.   
בשביל לשלוח ארגומנטים למילון צריך להגדיר שם של משתנה(שמשמש כמפתח) והערך שהוא שווה, כך האופרטור ידע לפרק את הארגומנטים בפונקציה:

>>> def f(\*\*kwargs):  
... print(kwargs)  
... print(type(kwargs))  
... for key, val in kwargs.items():  
... print(key, '->', val)  
...  
>>> f(e=1, r=2,j=3)  
{'e': 1, 'r': 2, 'j': 3}  
<class 'dict'>  
e -> 1  
r -> 2  
j -> 3

במקרה כזה המפתחות צריכים להיות מוגדרים כמילה לא שמורה בזיכרון, למשל אי אפשר לשלוח 1=1 לפונקציה וזה יחשב כשגיאה.  
וכפי שאפשר לפרק ערכים באופרטור '\*' , אפשר לפרק אותם באופרטור גם '\*\*', הדבר המעניין הוא שאם נרצה לשלוח את המפתחות לפונקציה במקום את הערכים נוכל באופרטור '\*':

>>> def f(a, b, c):  
... print(F'a = {a}')  
... print(F'b = {b}')  
... print(F'c = {c}')  
...  
>>> d = {'a': 'foo', 'b': 25, 'c': 'qux'}  
>>> f(\*\*d)  
a = foo  
b = 25  
c = qux  
>>> f(\*d)  
a = a  
b = b  
c = c

ניתן גם להשתמש באופרטור כדי לפרק כמה מילונים למילון אחד:

>>> my\_first\_dict = {"A": 1, "B": 2}  
>>> my\_second\_dict = {"C": 3, "D": 4}  
>>> my\_merged\_dict = {\*\*my\_first\_dict, \*\*my\_second\_dict}  
>>> print(my\_merged\_dict)  
{'A': 1, 'B': 2, 'C': 3, 'D': 4}

אם רוצים להשתמש בשני סוגי האופרטורים בפונקציה חשוב לשמור שקודם יבוא פרמטר עם אופרטור \* ואח"כ פרמטר עם '\*\*':

>>> def f(a, b, \*args, \*\*kwargs):  
... print(F'a = {a}')  
... print(F'b = {b}')  
... print(F'args = {args}')  
... print(F'kwargs = {kwargs}')  
...  
>>> f(1, 2, 'foo', 'bar', 'baz', 'qux', x=100, y=200, z=300)  
a = 1  
b = 2  
args = ('foo', 'bar', 'baz', 'qux')  
kwargs = {'x': 100, 'y': 200, 'z': 300}

*שאלה:* מה יודפס במקום הסימני שאלה?

>>> def tup\_dict(\*tup,\*\*dict):  
... print(f'{type(tup)}, {type(dict)}')  
... print(dict)  
... print(tup)  
...  
>>> lst=(1,2,3)  
>>> dic={'a':1,'b':2}  
>>> tup\_dict(\*lst,dic)  
?  
>>> tup\_dict(\*lst,\*\*dic)  
?  
>>> tup\_dict(\*lst,\*dic)  
?

***ביאורים (annotations) לפונקציות-***מלבד docstring שראינו אותו בפרק 'מבנה של סקריפט פייתון', החל מפייתון 3.0 ,קיימת בשפה עוד אפשרות להגדיר [metadata](https://he.wikipedia.org/wiki/%D7%9E%D7%98%D7%90-%D7%93%D7%90%D7%98%D7%94) על הפונקציה, וזה באמצעות annotations.  
אז מה זה בעצם הביאורים האלה? הביאורים מאפשרים לנו להציג אילו ערכים הפונקציה אמורה לקבל, ומה אמורה להחזיר.   
כדי להגדיר 'ביאור' נוסיף נקודתיים אחרי הגדרת המשתנה וטיפוס המשתנה:

>>> def f(a: int, b: str) -> float:  
... print(a,b)  
... return 3.5  
...

נוכל לראות את הביאורים ע"י המשתנה \_\_annotations\_\_ של הפונקציה:

>>> f.\_\_annotations\_\_  
{'a': <class 'int'>, 'b': <class 'str'>, 'return': <class 'float'>}

נשים לב שמה שקיבלנו מהמשתנה \_\_annotations\_\_ הוא לא מחרוזת כמו docstring, אלא מילון, וכמילון ניתן לגשת גם לכל ערך בנפרד:

>>> f.\_\_annotations\_\_['a']  
<class 'int'>  
>>> f.\_\_annotations\_\_['b']  
<class 'str'>  
>>> f.\_\_annotations\_\_['return']  
<class 'float'>

ניתן אפילו להוסיף יותר מידע על כל אובייקט, למשל נרצה להוסיף מה אמור להיות התפקיד של כל משתנה ומה הערך שהפונקציה מחזירה אמור לייצג:

>>> def area(  
... r: {  
... 'desc': 'radius of circle',  
... 'type': float  
... }) -> \  
... {  
... 'desc': 'area of circle',  
... 'type': float  
... }:  
... return 3.1415926 \* (r \*\* 2)  
...  
>>> area(2.5)  
19.63495375  
>>> area.\_\_annotations\_\_  
{'r': {'desc': 'radius of circle', 'type': <class 'float'>}, 'return': {'desc':  
'area of circle', 'type': <class 'float'>}}  
>>> area.\_\_annotations\_\_['r']['desc']  
'radius of circle'  
>>> area.\_\_annotations\_\_['return']['type']  
<class 'float'>

בדוגמא לעיל כל אחד מהמשתנים (r והערך חזרה) הוא מילון בפני עצמו.  
אם נרצה גם להגדיר ערך לפרמטר נעשה זאת לאחר הגדרת הannotation:

>>> def f(a: int = 12, b: str = 'baz') -> float:

... print(a, b)

... return(3.5)  
...

>>> f.\_\_annotations\_\_

{'a': <class 'int'>, 'b': <class 'str'>, 'return': <class 'float'>}

>>> f()

12 baz

3.5

האם ה-annotations מחייבים? האמת היא שלא, ניתן לשלוח ארגומנט מטיפוס שונה מזה של הביאור של הפרמטר ולא נקבל שגיאה.   
המטרה של הביאורים האלו היא בעיקר ליצור דוקומנטציה טובה יותר לקוד- הפרמטרים ומטרתם, מה הפונקציה אמורה להחזיר וכו'.   
אבל כן יש דרך להשתמש ב-annotations כדי לכפות על טיפוסים וזה ע"י ספריות עזר כמו inspect שיכולה לקחת ולבדוק את טיפוסי המשתנים שנמצאים בפונקציה ולהשוות לטיפוסים ב-annotations , במידה והטיפוס שונים זה כבר ביד המתכנת להחליט מה לעשות.  
להלן דוגמא לשימוש בספרייה:

>>> def f(a: int, b: str, c: float):  
... import inspect  
... args = inspect.getfullargspec(f).args  
... annotations = inspect.getfullargspec(f).annotations  
... for x in args:  
... print(x, '->',  
... 'arg is', type(locals()[x]), ',',  
... 'annotation is', annotations[x],  
... '/', (type(locals()[x])) is annotations[x])  
...  
>>> f(1, 'foo', 3.3)

a -> arg is <class 'int'> , annotation is <class 'int'> / True

b -> arg is <class 'str'> , annotation is <class 'str'> / True

c -> arg is <class 'float'> , annotation is <class 'float'> / True

>>> f('foo', 4.3, 9)

a -> arg is <class 'str'> , annotation is <class 'int'> / False

b -> arg is <class 'float'> , annotation is <class 'str'> / False

c -> arg is <class 'int'> , annotation is <class 'float'> / False

>>> f(1, 'foo', 'bar')

a -> arg is <class 'int'> , annotation is <class 'int'> / True

b -> arg is <class 'str'> , annotation is <class 'str'> / True

c -> arg is <class 'str'> , annotation is <class 'float'> / False

הערה: מרבית הדוגמאות מהפרק נלקחו מתוך האתר: https://realpython.com/  
realpython הוא אתר מעולה ללמוד פייתון בכל רמה ,ומומלץ בחום לעיון ולהרחבה מעבר להיקף החומר הנלמד בשיעור.

***פונקציות למדא -***פונקציות למדא (או למבדא) הן פונקציות אנונימיות יותר תמציתיות אך יותר מוגבלות מבחינת סינטקס מפונקציות רגילות.   
המבנה של פונקציית למבדא הוא: תיאור(המילה למדא) , פרמטרים והביטוי - lambda x : x+4  
בפייתון גם פונקציות הן אובייקט ולכן נוכל לשים למשתנה פונקציה כערך:

>>> x = lambda a : a + 10  
>>> print(x(5))  
15  
>>> type(x)  
<class 'function'>

ניתן גם להפעיל את הפונקציה עם כתיבתה:

>>> (lambda a : a + 10)(2)  
12

פונקציות למדא יכולות לקבל גם כמה פרמטרים שמוגדרים בתוך רשימה ללא סוגרים, ניתן גם להגדיר לפונקציה ערכים דיפולטיבי:

>>> z = lambda a=1,b=2,c=3 : a + b + c  
>>> z(1,2)  
6

לפונקציית למדא האחרונה שהגדרנו ,שלא הכנסנו למשתנה, אנחנו יכולים לקרוא במפרש עם '\_':

>>> lambda first\_name, last\_name: f"My name is: {first\_name} {last\_name}"  
>>> \_("Tom","Pythonovitz")  
My name is Tom Pythonovitz

הערה: המילה השמורה '\_' יעבוד רק אם אנחנו מריצים את הפונקציה במפרש של התוכנית הראשית, זה לא יעבוד לנו אם נשתמש בפונקציה של module .   
השימוש בפונקציית למדא נעשה בדר"כ בתוך פונקציה ממעלה גבוה יותר:

>>> def high\_ord\_func(x , func):  
... return func(x)+x  
...  
>>> high\_ord\_func(2, lambda x: x\*\*2)  
6  
>>> high\_ord\_func(2, lambda x: x+2)  
6  
>>> high\_ord\_func(2, lambda x: x%2)  
2

בפייתון ההבדל בין פונקציות למדא לפונקציה רגילה הוא לכאורה הבדל של syntactic sugar בעיקר , שכן את אותן פעולות שניתן לעשות בפונקציות למדא ניתן לעשות בפונקציה רגילה:

>>> def func(x):  
... return x+x  
...  
>>> high\_ord\_func(2, func)  
6

אז למה להשתמש בפונקציות למדא בכלל?   
הכוח של פונקציות למבדא נראה יותר דרך שימוש בפונקציות כפונקציות אנונימיות בתוך פונקציות  
נגיד יש לנו פונקציה ,כמו בדוגמא למעלה, שמקבלת כפרמטר פונקציה אחרת, ושהפרמטר הזה יוכפל פי כמה, מספר לא מוגדר של פעמים:

>>> def myfunc(n):  
... return lambda a : a \* n  
...  
>>> mydoubler = myfunc(2)  
>>>  
>>> print(mydoubler(11))  
22

או שנוכל להשתמש באותה פונקציה כתבנית לכמה פונקציות:

>>> def myfunc(n):  
... return lambda a : a \* n  
...  
>>> mydoubler = myfunc(2)  
>>> mytripler = myfunc(3)  
>>> print(mydoubler(11))  
22  
>>> print(mytripler(11))  
33

***פונקציות build-in בפייתון-***   
השפה מגיע עם כמה פונקציות בנויות מראש שאת חלקם כבר יצא לנו לראות.  
מרבית הפונקציות הבנויות מראש של פייתון ניתן לחלק לכמה כמה "קבוצות":

פונקציות קלט ופלט-  
הפונקציה המוכרת ביותר לפלט היא print() שמדפיסה למסך את מחרוזת שהיא כארגומנט.  
לפונקציה כמה פרמטרים: ערך- שהוא tuple של כמה אובייקטים , לרוב מחרוזת, שמופרדים בפסיקים אחד מהשני.   
sep – שמגדיר מה ההפרדה שתהיה בין כל משתנה, איך לו הגדרה כברירת מחדל ;   
end- מה יודפס בסוף, כברירת מחדל זה ירידה '\n' ;  
file- לאן להדפיס את הערך , כברירת מחדל הוא יודפס ל- sys.stdout אבל ניתן גם להדפיס לקובץ.   
flush- הקלטים של הפונקציה נשמרים בחוצץ (buffer) עד שמודפסת שורה חדשה, לפעמים נרצה "להדיח" אותם לפני שתודפס שורה חדשה.

>>> hello\_world = ['H','E','L','L','O','-','W','O','R','L','D']  
>>> for char in hello\_world:  
... print(char,end ="")  
... else:  
... print("")  
...  
HELLO-WORLD

וכמו שיש פונקציית פלט יש גם פונקציה שמקבלת ערכים- input() , שהפרמטר היחיד שלה הוא מחרוזת שתודפס למסך לפני קבלת הקלט, הקלט שיתקבל הוא מחרוזת:

>>> x= input("Enter your value:")  
Enter your value:123  
>>> x   
'123'

פונקציות המרה- למשל הפונקציה int() , float(), str(), complex(), list() וכו'  
הן למעשה בנאים למחלקה אותה הם מייצגים.  
בעצם מה שקורה זה שאנחנו יוצרים אובייקט חדש מטיפוס האובייקט אותו אנחנו רוצים לקבל מההמרה.  
חלק מההמרות יכולות להיות גם המרות מתוך טבלת Unicode או ascii , בג'אווה לדוגמא אם נמיר char ל-int נקבל את הערך ה-ascii שלו ולא את הפירסור שלו, בפייתון ניתן להשיג את זה עם הפונקציה ord() שמחזירה את הערך של האות בטבלת הunicode, לדוגמא ord('a')==97, להמרה הפוכה נשתמש בפונקציה chr()- chr(97)=='a'.  
פונקציה מיוחדת במחרוזות היא הפונקציה format() שיצא לנו להכיר אותה בהקשר של fstring אבל לא יצא לנו לראות מה היכולות שלה מעבר להצבה של משתנים.  
format() מאפשרת לערוך את המשתנים במחרוזות ולשנות את המבנה שלהם, למשל אם נרצה להציג רק שני מספרים אחרי הנקודה של משתנה float או שנרצה להציג את הנתונים בצורה מסוימת נוכל להשתמש בפונקציה כדי "לייפות" את המחרוזת.  
זה טוב בעיקר כדי לבנות מחרוזת שמייצגת template עבור כמה משתנים, ואז להגדיר מחרוזת אחת ולהשתמש בה כמה פעמים .  
כברירת מחדל אם לא הגדרנו ערך בתוך הסוגריים המשתנים יעברו למחרוזת כסדרם, אך ניתן להגדיר איזה משתנה יעבור לאיזה חלק במחרוזת אם נגדיר להם מספר.

>>> print("{0} love {1}??".format("fishermen" ,"fish"))  
fishermen love fish??  
>>> print("{1} love {0}??".format("fishermen" ,"fish"))  
fish love fishermen??  
>>> string = "Where nothing goes {} just go {}"  
>>>print(string.format("right","left"))  
Where nothing goes right just go left   
>>>print(string.format("left","right"))  
Where nothing goes left just go right

אפשר להשתמש בפונקציה גם בלי להיצמד למחרוזת מסויימת ולקבל מחרוזת חדשה שממירה את התצוגה של המשתנה:

>>> comma\_format = format(4000000, ',')  
>>> binary\_format= format(4,'b')  
>>> precentage\_format = format(0.4,'%')  
>>> scientific\_format = format(4,'e')  
>>> fix\_point= format(4.44444,'.2f')  
>>> print(comma\_format)  
4,000,000  
>>> print(binary\_format)  
100  
>>> print(precentage\_format)  
40.000000%

>>> print(scientific\_format)

4.000000e+00

>>> print(fix\_point)   
 4.44

לרשימה המלאה של הפורמטים האפשריים מומלץ לעיין ב-

<https://www.w3schools.com/python/ref_func_format.asp>  
  
<https://www.geeksforgeeks.org/python-format-function/#:~:text=str.,a%20string%20through%20positional%20formatting>.

פונקציות מתמטיות- פונקציות המבצעות פעולות מתמטיות על אובייקטים.  
הדוגמא הקלאסית היא אופרטורים, אך יש גם פונקציות כמו abs() שמחזיר את הערך המוחלט של המשתנה, או pow() שמבצע את אותה הפעולה כמו האופרטור '\*\*' אבל ניתן גם להגדיר לו שארית חלוקה: pow(2, 2,mod=3)->1, כנראה השימוש בפונקציה הוא עם מימוש שונה משל האופרטור.   
הפונקציה divmod() מקבלת שני מספרים ומחזירה את החלוקה בערך תחתון, ושארית החלוקה שלהם   
למשל   
.divmod(5,6)==(0,5)   
  
חוץ מהפונקציות הנ"ל יש עוד מלא פונקציות מתמטיות בספרייה math כגון: floor, random, factorial() וכו'  
  
  
פונקציות על אוספים- גם פונקציות על אוספים אפשר לחלק לשני סוגים עיקריים: אלה שמחזירים ערך בודד, ואלא שמחזירים אוסף במקום:   
ערך בודד:  
 min(), max() מחזירות את הערך המינימלי והמקסימלי של אוסף;  
sum() -מחזיר את הסכום איברי האוסף; all() – בודק שכל איברי האוסף הם אמיתיים (ראה פרק טיפוסי נתונים בפייתון- המרות לערך בוליאני), ו-any() אם לפחות אחד מהערכים הוא אמיתי; len() –מחזיר את אורך המחרוזת.  
  
מחזיר אוסף:  
הפונקציה range() שכבר ראינו מחזירה לנו אובייקט מסוג iterable.  
הפונקציה frozenset() היא כמין פונקציה ממירה שמקבלת איזשהו אוסף ומחזירה סט קפוא- סט קפוא הוא כמו סט רגיל רק שהוא immutable , כלומר הוא אוסף לא ממוין ולא ניתן לשינוי .  
הפונקציה zip() מקבלת שני אוספים ומחזירה זוגות של tuples משני מכל איבר באוסף:

>>> frozen\_friends\_male = [" Joey", "Chandler", " Ross"]  
>>> frozen\_friends\_female = frozenset( (" Rachel", "Phoebe", "Monica"))  
>>> friend\_zone = zip(frozen\_friends\_male, frozen\_friends\_female)  
>>> print(tuple(friend\_zone))  
((' Joey', 'Monica'), ('Chandler', 'Phoebe'), (' Ross', ' Rachel'))

משום שהאוסף מוגדר כ-fozenset הוא יוצא בצורה אקראית  
הפונקציה slice() מאפשרת לקחת ביטוי של האופרטור '[ ]' ולהפוך אותו למשתנה, מה הכוונה?   
נניח אנחנו רוצים רק חלק קטן מאיזשהו אוסף , למשל רק את המקומות הזוגיים בין האינדקס הראשון לחמישי, בשיטה הרגילה היינו עושים: collection[1:5:2] , עכשיו נניח אנחנו צריכים את האוסף הזה ספציפית ואנחנו רוצים לחסוך במקום יקר, אז נוכל ליצור אובייקט מסוג slice שיחזיר שישמור את ההגדרה '[1:5:2]' וכל פעם שנשתמש בו הוא יומר באופרטור להגדרה המקורית:

>>> my\_str = "I can't"  
>>> x = slice(0,5)  
>>> print(my\_str[x])  
I can

הפונקציה sorted() מקבלת אוסף ומחזירה רשימה ממוינת לפי פונקציית מיון שהגדרנו לה, יש לה שלושה פרמטרים- האוסף, key שהוא פונקציית המיון (כברירת מחדל מהקטן לגדול), וreverse- שמגדיר אם להפוך את הרשימה:

>>> my\_tuple = (9,1,8,2,7,3,6,4,5)  
>>> print(f'Normally: {sorted(my\_tuple)}')  
Normally: [1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> print(f'Reversed: {sorted(my\_tuple,reverse=True)}')  
Reversed: [9, 8, 7, 6, 5, 4, 3, 2, 1]

>>> print(f'By function: {sorted(my\_tuple,key=lambda x: x%2!=0)}')  
By function: [8, 2, 6, 4, 9, 1, 7, 3, 5]

הפונקציה filter מקבלת אוסף ופונקציה בוליאנית למיון, ומחזירה אלו איברים באוסף מקיימים את התנאי:

>>> ages = [5, 12, 17, 18, 24, 32]  
>>>  
>>> def myFunc(x):  
... if x < 18:  
... return False  
... else:  
... return True  
...  
>>> adults = filter(myFunc, ages)  
>>>  
>>> for x in adults:  
... print(x)  
...  
18  
24  
32

הפונקציה reversed() מחזירה איטרטור הפוך לאוסף:

>>> alef = "alef"  
>>> fela = revesed(alef)  
>>> for char in fela  
... print(char, end=" ")   
f e l a

פונקציות הערכה- פונקציות שמעריכות ערך של ביטוי מסוים (מחרוזת)-   
eval()- מקבלת ביטוי ומעריכה את ערכו אם הוא ביטוי פייתון תקין, ויש לה שני פרמטרים אופציונליים:locals ו-globals שהם מילונים שמגדירים סוגי ערכים בביטוי:

>>> x = 'print(y)'  
>>> eval(x,{'y':"hello"})  
hello

הפונקציה eval מקבלת רק שורת קוד, הפונקציה exec דומה לה רק שהיא יכולה לקבל גם בלוק שלם:

>>> x = 'name = "Tom"\nprint(name+" "+last\_name)\n'  
>>> exec(x,{'last\_name':"Pythonovitz"})  
Tom Pythonovitz

פונקציות על מחלקות:   
 הפונקציה hasattr()- מחזירה ערך בוליאני אם לאובייקט מסוים יש תכונה מסוימת:

>>> class Person:  
... name = "John"  
... age = 36  
... country = "Norway"  
...  
>>> x = hasattr(Person, 'age')  
>>> print(x)  
True

הפונקציה getattr() כמו הפונקציה hasattr() רק שמחזירה את הערך של האובייקט, ניתן לתת לה ארגומנט שלישי שיהווה את התוצאה אם הארגומנט לא נמצא.

>>> x = getattr(Person, 'age')  
>>> print(x)  
36  
>>> x = getattr(Person, 'page', 'my message')  
>>> print(x)  
my message

הפונקציה vars() מחזירה מילון עם כל הערכים של האובייקט:

>>> x = vars(Person)  
>>> print(x)  
{'\_\_module\_\_': '\_\_main\_\_', 'name': 'John', 'age': 36, 'country': 'Norway', '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Person'objects>,'\_\_weakref\_\_':<attribute'\_\_weakref\_\_' of 'Person' objects>, '\_\_doc\_\_': None}

וכמובן פונקציה שכבר יצא לנו לראות type() שמחזירה את טיפוס המשתנה.

פונקציה שחשוב שנכיר היא help() והיא מביאה לנו מידע על כל אובייקטים של פייתון:

>>> help(Person)  
Help on class Person in module \_\_main\_\_:  
class Person(builtins.object)  
 | Data descriptors defined here:  
 |  
 | \_\_dict\_\_  
 | dictionary for instance variables (if defined)  
 |  
 | \_\_weakref\_\_  
 | list of weak references to the object (if defined)  
 |  
 | ----------------------------------------------------------------------  
 | Data and other attributes defined here:  
 |  
 | age = 36  
 |  
 | country = 'Norway'  
 |  
 | name = 'John'

טיפול בקבצים-

גם בפייתון כמו ג'אווה אפשר לטפל בקבצים, הכוונה שניתן לפתוח קובץ קיים ולכתוב אליו\ לקרוא ממנו , או לייצר אחד חדש, ולתת הרשאות לקובץ.   
בג'אווה כשרוצים להשתמש בקובץ צריך לייבא ספריה ייחודית לכך ,להפעיל פונקציה על הקובץ ולשמור את הנתונים באובייקט מטיפוס File.   
עם פייתון לרוב החיים יותר פשוטים יותר ,וכשזה מגיע לטיפול בקבצים אפשר להחשיב את פייתון כ-"הוואי של שפות התכנות" (אני פשוט מניח שהחיים בהוואי יותר טובים מהוויב שאני, בכל מקרה תוכלו לקחת את המשל לכל מקום שנראה לכם כ-"חיים הטובים" ).  
פייתון מתייחס אחרת לקבצי טקסט ולקבצים בינאריים.   
כל שורה של קוד כוללת רצף של תווים והם יוצרים קובץ טקסט, וכל שורה בקובץ נגמרת עם תו מיוחד שנקרא EOL (End Of Line) כמו ירידת שורה או פסיק, והתו מסמן למפרש מתי נגמרת השורה ומתחילה שורה חדשה בקובץ.   
בשביל לפתוח קובץ משתמשים בפונקציה בנויה מראש open() שמקבלת כפרמטר את שם המסמך (ה-path שלו), ותו הרשאה: r- לקריאה(reading); w- לכתיבה מחדש של המסמך, כלומר שכתוב שלו(writing); a- הוספה למשך הקיים (appending); r+ -גם לקרוא וגם לכתוב.  
הפונקציה מחזירה אובייקט מטיפוס קובץ:

>>> file = open("stam.txt","r")  
>>> type(file)  
<class '\_io.TextIOWrapper'>

התו הרשאה לא מחייב וכברירת מחדל הוא נחשב 'r' .  
  
***קריאה-***  
יש כמה דרכים לקרוא ממקובץ, הדרך האינטואיטיבית היא לקרוא שורה אחר שורה מהקובץ בלולאה, אך ניתן גם לקבל את כל תוכן הקובץ כמחרוזת עם הפונקציה read() של קבצים שמקבלת כפרמטר int עם כמות התווים שנרצה לקרוא מהקובץ, אך אם לא נתנו לה ארגומנט היא תקרא את כל המסמך:

>>> file = open("stam.txt","r")  
>>> print(file.read())

הקריאה של הקובץ היא בצורה של חוצץ בזיכרון ומצביע לראש השורה, כך שכל שורה שנקראת המצביע מצביע לראש השורה ,עובר לשורה אחריה בסוף הקריאה של השורה ומוחק מהחוצץ בזיכרון את השורה שהוא קרא הרגע. בסוף קריאת המסמך המצביע עומד על סוף המסמך.  
אי אפשר לקרוא את המסמך שוב, אלא אם נפתח אותו מחדש.   
  
 ***כתיבה-***  
בדומה לפונקציה read() יש גם פונקציה write() שמאפשרת לכתוב למסמך.  
כתיבה עם הפונקציה write דורסת את מה שהיה כתוב לפני בטקסט (אם היה) ומחזירה את מספר התווים שהכנסנו.  
לאחר השימוש בפונקציה write() יש לסגור את השימוש האובייקט "קובץ" כלומר להראות לתוכנה שאין לנו כוונה שנות אותו יותר ויש לסגור אותו.  
אפשר גם לכתוב מסמך חדש במידה והוא לא קיים והוא ישמר בשם ומיקום שהכנסנו לפונקציה, כברירת מחדל באותה תיקייה של התוכנית:

>>> file = open('new\_file.txt','w')

>>> file.write("This is the first line I write in this file")

43

>>> file.write("\nThis is the second line I write un this file")

45

>>> file.close()

>>> file = open('new\_file.txt','r')

>>> print(file.read())

This is the first line I write in this file

This is the second line I write un this file

>>>

בדומה להרשאת כתיבה, ניתן גם להוסיף לקובץ הקיים עם ההרשאה 'a' והוא יוסיף לסוף הקובץ טקסט חדש:

file = open('new\_file.txt','a')

file.write("\nThis will add this line")

file.close()

This is the first line I write in this file

This is the second line I write un this file  
This will add this line

אם נבחר להשתמש ב-r+ זה יאפשר לנו גם לקרוא וגם לכתוב לקובץ, אך הכתיבה תעשה למקום בו המצביע עומד בדיוק, מה שעלול למחוק שורות מתוך הטקסט.   
  
***המילה השמורה with-***  
אם עד עכשיו לא הרגשתם בהוואי חכו שתשמעו על הפונקציה הבאה:  
השימוש בopen() ו-close() יכול מאוד לבלבל, לכן המפתחים של פייתון סיפקו דרך שבה לא נצטרך להיכנס בכלל לתסבוכת הזאת.   
במקום להגדיר פתיחה וסגירה של קובץ, יש אפשרות להגדיר בלוק שבסופו יתבצע סגירה אוטומטית של הקובץ, עם המילה שמורה with, פתיחת קובץ והגדרת שם הקובץ עם המילה שמורה as:

>>> with open("hawaii.txt") as file:

... print(file.read())

...

,---,

,--.' | ,--, ,--,

| | : .---. ,--.'| ,--.'|

: : : /. ./| | |, | |,

: | |,--. ,--.--. .-'-. ' | ,--.--. `--'\_ `--'\_

| : ' | / \ /\_\_\_/ \: | / \ ,' ,'| ,' ,'|

| | /' :.--. .-. | .-'.. ' ' ..--. .-. | ' | | ' | |

' : | | | \\_\_\/: . ./\_\_\_/ \: ' \\_\_\/: . . | | : | | :

| | ' | : ," .--.; |. \ ' .\ ," .--.; | ' : |\_\_' : |\_\_

| : :\_:,'/ / ,. | \ \ ' \ |/ / ,. | | | '.'| | '.'|

| | ,' ; : .' \ \ \ |--"; : .' \; : ; : ;

`--'' | , .-./ \ \ | | , .-./| , /| , /

`--`---' '---" `--`---' ---`-' ---`-'

טיפול בשגיאות-

תכניות נגמרות ברגע שהן נתקלות בשגיאה בקוד.  
כמתכנתים נרצה להימנע כמה שיותר משגיאות, אך לפעמים אפילו לא נוכל להימנע לחלוטין מכניסה למצבים שעלולים להוביל לשגיאות, ובמקרים כאלה ללא נרצה שהתוכנית תפסיק לחלוטין, במיוחד כשהשגיאה לא קשורה להמשך התוכנית.  
בפייתון יש שני סוגים של שגיאות- שגיאות סינטקס, וחריגות.   
שגיאות סינטקס קורות כאשר המפרש מבחין בביטוי או הכרזה שאינם נכונים מבחינת צורת הכתיבה שלהם:

>>> print( 0 / 0 ))  
File "<stdin>", line 1

print( 0 / 0 ))

^

SyntaxError: invalid syntax

השלב של שגיאות סינטקס מגיע עוד לפני השלב שהמפרש מנסה להעריך את הביטוי, למשל במקרה לעיל המפרש מסמן על שגיאה תחבירית בחלק של הסוגרים הכפולים, ולא על החילוק באפס.  
אם היינו מתקנים את השגיאה היינו נתקלים בשגיאה מסוג אחר:

>>> print( 0 / 0)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ZeroDivisionError: integer division or modulo by zero

הפעם המפרש "זורק" שגיאת חריגה.   
סוג זה של שגיאה כאשר הקוד כתוב נכון מבחינה תחבירית, אך עדיין המפרש נתקל בכשל במהלך הערכה של הביטוי.

זריקת שגיאות-   
ניתן לזרוק שגיאה בעצמנו עם המילה השמורה raise ואחריה ליצור אובייקט מסוג חריגה עם הגדרת החריגה:

>>> def error\_fun(x):  
... raise Exception("Invalid input")  
...  
>>> error\_fun(0)  
Traceback (most recent call last):  
 File "<stdin>", line 1, in <module>  
 File "<stdin>", line 2, in error\_fun  
Exception: Invalid input

אפשר גם להשתמש בפונקציה assert() כדי לוודא שהמכשיר שמריץ את התוכנית יכול לעמוד בקריטריונים של התוכנית, או שנרצה לבחון את הקוד שלנו לפני תחילת התוכנית כדי לוודא שהתוכנית לא תעצר באמצע או תפגע במכשיר שמריץ.  
בפונקציה אנחנו בודקים שתנאי מתקיים, במידה והוא לא הפונקציה תזרוק שגיאה מכוונת עוד בהתחלה ותמנע מהתוכנית ליפול בהמשך:

import sys

assert ('linux' in sys.platform), "This code runs on Linux only."

אם מערכת ההפעלה של המכשיר שהריץ את הקוד היא לינוקס הפונקציה לא תזרוק שגיאה, אחרת היא תזרוק שגיאת assert שהסיבה לה היא המחרוזת שמופרדת בפסיק מהפונקציה assert   
  
try ו- except -  
 בג'אווה מוכר כ-try ו- catch , הוא מנגנון שמאפשר לנו לנסות לבצע רצף של פקודות ולהגדיר מה יקרה במקרה שנזרקה שגיאה במהלך ריצת הפקודות.   
עם המילה השמורה try אנו מורים לתוכנית להריץ את הבלוק הבא, ועם המילה except מורים לבצע רצף את רצף הפקודות שמופיע מתחת במקרה שנתקלנו בשגיאה שהגיע מהבלוק של ה-try.

try:

10/x

except:

pass

בפייתון לא ניתן לכתוב בלוק ריק כמו בג'אווה, לכן משתמשים במילה השמורה pass כשאין פקודות לביצוע.  
אם הערך של x היה 0 הייתה אמורה להתקבל שגיאת חלוקה באפס, השגיאה אכן הגיעה אך היא נתפסה ב-except שביצע את הפקודה שבאה בבלוק שלו במקום :

>>> def devi(x):  
... try:  
... 10/x  
... except:  
... print(f'EXCEPTION')  
...  
>>> devi(10)  
>>> devi(0)  
EXCEPTION

מה שאנחנו לא זוכים לראות זה את סוג השגיאה שנתפסה, בשביל לראות את סוג השגיאה נצרך להגדיר לexcept מה סוג השגיאה שעלולה לקרות:

>>> def devi(x):  
... try:  
... 10/x  
... except ZeroDivisionError as error:  
... print(error)  
...  
>>> devi(0)  
division by zero

לא מומלץ לתפוס שגיאות מבלי להגדיר אותן מראש מעוד סיבה- ההגדרה של except מבלי להגדיר את סוג החריגה תתפוס את **כל** סוגי החריגות, ולרוב לא נרצה את, הדוגמא הבאה תמחיש את זה:

while True:   
 try:  
 print("I am in a looppppppp ☺" )  
 except:  
 print("Hooo I'm still in the loop ☹")

גם אם ננסה לעצור את הלולאה האינסופית הזאת עם ctrl+c בטרמינל, נראה שהיא לא תעצר מהסיבה הפשוטה שהיא תופסת גם interruptions כלומר היא תופסת את כל סוגי החריגות גם אלה אינן שגיאות.   
אבל אם במקום נגדיר את סוג החריגה העצירה העצירה תתבצע כמתוכנן:

while True:   
 try:  
 print("I am in a looppppppp ☺" )  
 except Exception:  
 print("finally I stopped!!! ")

בקשר לחריגה מטיפוס Exception: האובייקט הוא טיפוס אב לרוב סוגי החריגות הקיימות בשפה למעת חריגות BaseException, systemExit, KeyboardInterrupt ו- GeneratorExit שהן חריגות מיוחדות.   
במקרים בהם לא ידוע לנו סוג החריגה שעלולה לקרות הוא יכול לשמש כגלגל הצלה, אך תמיד כדאי להשתמש בסוג החריגה המקורי כדי א. לידע את שאר המתכנתים על שגאיות שעלולות לקרות, ב. כדי להגדיר את התנהגות ספציפית במקרה של סוג מסוים של חריגה:

>>> def file\_reader(file\_path):  
... try:  
... with open(file\_path) as file:  
... file\_str = file.read()  
... 10/len(file\_str)  
... except FileNotFoundError as fnt:  
... print("there is no such file, try again…")  
... except ZeroDivisionError:  
... print("Are you crazy? Did you just tried to divide by 0?!")  
>>> file\_reader("not\_such\_file.txt")  
there is no such file, try again…  
>>> file\_reader("empty.txt")  
Are you crazy? Did you just tried to divied by 0?!

כשחריגות מופיעות על המסך ועוצרות את ריצת התוכנית הן מציגות לנו את מקור השגיאה, כלומר מאיזו קריאה בתוכנית (מאיזו שורה בקוד) היא נזרקה.  
כשאנחנו תופסים שחריגות בexcept נעדיף שזה יציג לנו את מקור החריגה ולא רק את שמה, כדי לעשות את זה נוכל להשתמש בmodule "taceback" שמאפשר לנו בקלות לקבל מידע על החריגה:

>>> import traceback  
>>> try:  
... raise Exception("I am an Exception >:-) ")  
... except Exception as e:  
... print("just printing the exception")  
... print(e,'\n')  
... print("printing the traceback")  
... traceback.print\_exc()  
... print("\nprinting traceback.format\_exc():")  
... print(traceback.format\_exc())  
...

just printing the exception  
I am an Exception >:-)

printing the traceback  
Traceback (most recent call last):  
 File "<stdin>", line 2, in <module>  
Exception: I am an Exception >:-)

printing traceback.format\_exc():  
Traceback (most recent call last):  
 File "<stdin>", line 2, in <module>  
Exception: I am an Exception >:-)

וכמו בלולאות גם לtry ו- except יש else רק שבמקרה זה הבלוק שלו מתבצע רק אם לא נתפסו חריגות ב-except.   
ולפעמים נרצה שתתבצע פעולת "ניקוי" גם ארע חריגה וגם אם לא, למשל פתחנו מסמך ונרצה לסגור אותו בין אם הייתה חריגה ובין אם לאו, לא נוכל לסגור את המסמך בבלוק של ה-try כי יכול להיות שלא נגיע לשורת קוד הזו וישר נכנס לבלוק של ה-except , במקרה זה נוכל להשתמש ב-finally שמגדיר בלוק שיקרה גם בין אם נתפסה חריגה ובין אם לא:

>>> def file\_writer(file\_path):  
... try:  
... file = open(file\_path)  
... file\_str = file.read()  
... 10/len(file\_str)  
... except FileNotFoundError as fnt:  
... print("there is no such file, try again.")  
... except ZeroDivisionError:  
... print("Are you crazy? Did you just tried to divide by 0?!")  
... else:  
... file.write("No exceptions were cought")  
... finally:  
... try:  
... file.close()  
... except NameError:  
... pass  
...  
>>> file\_writer("not\_such\_file.txt")  
there is no such file, try again.  
>>> file\_writer("empty.txt")  
Are you crazy? Did you just try to divide by 0?!

תכנות מונחה עצמים-

תכנות מונחה עצמים הוא מתודולוגיה תכנותית שמפרידה בין חלקי קוד לאובייקטים שמתקשרים בניהם.   
הרעיון של תכנות מונחה עצמים הוא פירוק התוכנית לחלקים עצמיים שמתקשרים בניהם כך שלא נצטרך למחזר קוד, כלומר כל שינוי שמשותף לכמה חלקים בקוד יתבצע במקום אחד במקום בכל חלק בנפרד.  
הגישה מנסה לחקות את העולם האמתי בכך שהיא מציגה תכונות והתנהגויות משותפות במבנה אחד.  
כאשר הקוד מאורגן לפי אובייקטים המייצגים עצמים, הדבר מאפשר שימוש חוזר בתוכנה, עם אפשרות טובה להרחיב את הקוד גם בתכוניות אחרות, כמו כן מעבר טבעי יותר בין התכנון של הקוד ומימוש שלו.   
כבר בשפת c נראים ניצנים לשיטה עם פיתוח משתנה מטיפוס struct שניתן להגדיר לו "מהות" (מה הוא בא לייצג) ותכונות, למשל בן אדם- יש לו שם, גובה, מוצא, טמפרמנט וכו'   
בשפות שממשיכות את c ניתן גם להגדיר למבנה שיטות, כלומר איך האובייקט מתנהג, למשל אדם יש לו את השיטה "להגיד שלום", "ללכת", "להשתעל", "להוציא את הילד מהגן" וכו'.  
בג'אווה ובפייתון תכנות מונחה עצמים מיוצג במחלקות שהן התבנית של העצם, דהיינו התכונות והתנהגויות המשותפות לכל האובייקטים מאותו המבנה.  
למחלקות יכולות להיות: שדות(fields) מאפייני המחלקה כלפי פנים, תכונות(properties) המגדירים מאפיינים של המחלקה כלפי חוץ, ושיטות (methods) שהן פונקציות הפועלות של משתני המחלקה או מגדירות את התנהגות העצם.  
אובייקט הוא בעצם דוגמא(אינסטנס) לעצם מטיפוס המחלקה, ניקח למשל את הדוגמא שהצגנו מקודם- אדם, לאדם יש תכונות כמו שם, גובה, מוצא שלרוב אפשר לראות כלפי חוץ.   
יש לו גם שדות כמו מצב נפשי, מצב בריאותי וכו' שעלולים לשנות את מצבו החיצוני, אבל אינם גלויים כלפי חוץ.  
ויש לו גם שיטות שמתארות את ההתנהגות שלו או פועלות על המשתנים אחרים, למשל בנאדם שמצבו הבריאותי לא תקין יכול להפעיל איזו שיטה שתשנה את הגוון עור שלו וכו'.  
אובייקט הוא דוגמא לטיפוס מהמחלקה, אם המחלקה מייצגת "בן אדם" , אז אובייקט הוא למשל "אורנית" או "בני" וכו'.  
  
*מאפייני השיטה-*המאפיינים המרכזיים של השיטה הם : כימוס(encapsulation), הפשטת נתונים(data abstraction), ירושה (inheritance) ורב צורתיות (polymorphism) .  
  
כימוס- מהמילה כמוס כלומר מוסתר(כמו סוד כמוס לפרה ולסוס),

טקסט  
לחלק של המונחה עצמים: פונציות מיוחדות של מחלקות כמו \_\_str\_\_ ומימוש אופרטורים   
בטיפול בשגיאות לא לשכוח להוסיף על with ]