מג'אווה לפייתון-טיפוס נתונים

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 לא נתמקד על מתודות של מבני נתונים, אבל ניתן למצוא מידע עליהן בתיקייה 5.0 ← code.טיפוסי נתונים.)
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 immutable , 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(message)
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(message)
False,False,False,False,False,False,False
```