מג'אווה לפייתון - 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 בפונצה, למשל ()append של רשימות, אז השינוי היה ניכר גם בפונקציה וגם מחוץ לה:

```
>>> lst =[0]
>>> def null_lst(lst):
...  lst.clear()
...
>>> 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 זה 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 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
```

אוקיי כל זה טוב ויפה אם נשתמש במשתנים מטיפוס פרימיטיבי, אבל מה יקרה אם נרצה להשתמש באובייקט מטיפוס קצת יותר מתקדם, למשל רשימה שיכולה לקבל לעצמה סוגים של כמה אובייקטים, או פונקציה, או אובייקט ממחלקה שאנחנו הגדרנו?
לכל דבר שהוא מתקדם מטיפוס פרימיטיבי נשתמש בספרייה typing.
הספרייה מספקת את טיפוסים של אובייקטים מתקדמים עבור annotations , למשל עבור מילון נוכל להשתמש בDict של הספרייה או Tuple,List ו- Set עבור רשימה tuple וסט בהתאמה, ואפשר אפילו להגדיר מה סוג האוסף שאמור להתקבל או להישלח:

```
>>> from typing import List,Dict
>>> def print_names(names: List[str]) -> None:
...    for student in names:
...        print(student)
>>> def print_name_and_grade(grades: Dict[str, float]) -> None:
...    for student, grade in grades.items():
...        print(student, grade)
```

אפשר גם להגדיר annotation מוגדר מראש שבנוי מannotations מוכרים:

```
>>> from typing import List, Tuple
>>> # Declare a point type annotation using a tuple of ints of [x, y]
>>> Point = Tuple[int, int]
>>> # Create a function designed to take in a list of Points
>>> def print_points(points: List[Point]):
...    for point in points:
...        print("X:", point[0], "  Y:", point[1])
```

השתמשנו ב-annotation של tuple ו-int כדי להגדיר אובייקט חדש שקוראים לו point.
נוכל להשתמש ב-union כדי לתאר אפשרות של קבלה של כמה משתנים מטיפוסים שונים, למשל אם לפונקציה שלנו יש פרמטר שיכול להיות int או מחרוזת:

```
from typing import Union
def print_grade(grade: Union[int, str]):
    if isinstance(grade, str):
        print(grade + ' percent')
    else:
        print(str(grade) + '%')
```

לפעמים נרצה להחזיר פונקציה, במקרים כאלה נשתמש ב-Callable של typing, שבדומה לאוספים הביאור אמור להיות מוגדר עם רשימה שמכילה רשימה של טיפוסי הארגומנטים לפונקציה, וטיפוס ערך החזרה של הפונקציה:

```
>>> from typing import Callable
```

```
>>> def decoration_function(func: Callable[[int],None])->Callable[[None],None]:
>>>    def wrapper()->None:
...        print("An example of a decoration function")
...        func()
...    return wrapper
```

וביצירת אובייקט חדש נוכל להשתמש בפונקציה Type() כדי להוסיפו לביאורים:

```
>>> from typing import Type
>>> class MyClass:
...    pass
>>> myClass=Type[MyClass]
>>> def my_class_function()-> myClass:
    return MyClass()
```

החל מגירסה 3.7 של פייתון ניתן להשתמש במודל __future__.annotations כדי להגדיר annotations מתואמים מראש, ואין צורך להשתמש ב- typing.Type:

```
from __future__ import annotations
class MyNewClass:
    def return_self(self)->MyNewClass:
        return self
print(type(MyNewClass().return_self()))
```

```
<class '__main__.MyNewClass'>
```

יש לשים לב שהמודול __future__ צריך להיות המודול הראשון שמיובא

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 ה`unicode`, 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, ואלה 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), ו-reverse 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 אובייקט בפייתון:

```
>>> 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'
```