יחידה 5

פונקציות

ארבעה פרקים, 31 דקות. def, טווח הכרה, return, ברירות מחדל, תיעוד ו-main.

מפת היחידה

23 חלקים ▲

פרק	דקה	מה קורה שם
5.1	0:00	למה פונקציות
5.1	**1:05**	**`def` , וקריאה לפונקציה**
5.1	1:39	פונקציה עם פרמטר
5.1	2:42	שם בלי סוגריים: פונקציה היא אובייקט
5.1	3:45	פרמטר מול ארגומנט
5.1	4:16	ארגומנט מהטיפוס הלא נכון
5.2	0:32	משתנה בתוך פונקציה ומחוצה לה
5.2	**1:05**	**טווח הכרה: גלובלי ולוקאלי**
5.2	2:11	`NameError` מחוץ לפונקציה
5.2	3:14	אותו שם בפנים ובחוץ
5.2	5:21	קריאת משתנה גלובלי, ולמה לא
5.2	5:53	`global`
5.3	**0:35**	**`return`**
5.3	2:10	פונקציה בלי `return` מחזירה `None`
5.3	3:16	החזרת כמה ערכים
5.3	4:20	ערכי ברירת מחדל
5.3	6:23	ברירת מחדל רק בסוף
5.3	7:25	ארגומנטים לפי שם
5.4	**1:05**	**תיעוד (docstring)**
5.4	3:21	`help` ו- `__doc__`
5.4	**4:24**	**הפונקציה `main`**
5.4	6:35	קוד לא קריא מול קוד קריא

פרק	דקה	מה קורה שם
5.4	8:17	ארבעה כללים לפונקציות

5.1 · לכתוב פונקציה

פונקציה היא רצף פקודות באותו בלוק, שמבצע משימה מוגדרת. כותבים אותה פעם אחת, נותנים לה שם, וקוראים לה כמה פעמים שצריך. קל יותר לשנות, לבדוק ולדבג. ביחידה 3 השתמשנו בפונקציות מוכנות. מכאן כותבים לבד.

```
>>> def my_first_function():
...     print("This is my first function in Python")
...
>>> my_first_function()
This is my first function in Python
```

1. `def` (מ-define), שם הפונקציה, סוגריים ונקודתיים
2. גוף הפונקציה מוזח, כמו כל בלוק
3. **הגדרה אינה הרצה.** הפונקציה רצה רק כשקוראים לה, עם סוגריים

פרמטר

```
>>> def square(number):
...     print(number ** 2)
...
>>> square(3)
9
```

פונקציה עם פרמטר מצפה לקבל אותו בכל קריאה. בלעדיו:

```
>>> square()
Traceback (most recent call last):
  ...
TypeError: square() missing 1 required positional argument: 'number'
```

פרמטר מול ארגומנט

	מה זה	בדוגמה
פרמטר	השם בהגדרה	number
ארגומנט	הערך שנשלח בקריאה	3

שם בלי סוגריים

```
>>> square
<function square at 0x...>
```

אין שגיאה, ושום דבר לא רץ. שם של פונקציה בלי סוגריים הוא כמו שם של משתנה: הוא מצביע על אובייקט, והאובייקט הוא הפונקציה עצמה. (המספר אחרי `at` הוא המיקום בזיכרון, ומשתנה בכל הרצה.)

ולכן אפשר להשים פונקציה למשתנה:

```
>>> func = square
>>> func(6)
36
```

הטיפוס של הארגומנט

```
>>> square("3")
Traceback (most recent call last):
  ...
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

פייתון לא בודקת טיפוסים מראש. השגיאה תופיע רק כשהשורה תרוץ, ויכולה להפיל את התוכנית באמצע.
שליחת הטיפוס הנכון היא באחריות מי שקורא לפונקציה, ואת הטיפוס המצופה מוצאים בתיעוד שלה (פרק 5.4).

5.2 · טווח הכרה

טווח הכרה (scope) הוא האזור בתוכנית שבו משתנה קיים.

	איפה הוגדר	איפה מוכר
גלובלי	בתוכנית, מחוץ לכל פונקציה	בכל התוכנית
לוקאלי	בתוך פונקציה	רק בתוך הפונקציה

```
>>> fish_num = 9
>>> def fish_food():
...     food_for_fish = 5
...     print(food_for_fish * fish_num)
...
>>> fish_food()
45
```

`fish_num` גלובלי, ו- `food_for_fish` לוקאלי. כמו דגים: הם קיימים רק באקווריום.

לוקאלי מחוץ לפונקציה

```
>>> def ocean():
...     fish_view = "coral reef"
...     print(fish_view)
...
>>> ocean()
coral reef
>>> print(fish_view)
Traceback (most recent call last):
  ...
NameError: name 'fish_view' is not defined
```

אותו שם בפנים ובחוץ

```
>>> animal = "rabbit"
>>> def water():
...     animal = "goldfish"
...     print(animal)
...
>>> water()
goldfish
>>> print(animal)
rabbit
```

ההשמה בתוך הפונקציה **יצרה משתנה לוקאלי חדש**, ולא שינתה את הגלובלי. כשיש שני משתנים באותו שם, **הלוקאלי קודם**: פייתון מחפשת קודם בתוך הפונקציה, ורק אם לא מצאה עוברת לגלובלי.

הפתרון: שמות שונים. `land_animal` ו- `sea_animal` , וכבר לא צריך לנחש.

לקרוא משתנה גלובלי מתוך פונקציה

זה עובד (ראינו את `fish_num` למעלה), אבל **לא מומלץ**, אלא אם זה קבוע. הדרך המסודרת להכניס ערך לפונקציה היא **כארגומנט**.

`global`

```
>>> animal = "rabbit"
>>> def water():
...     global animal
...     animal = "goldfish"
...     print(animal)
...
>>> water()
goldfish
>>> print(animal)
goldfish
```

`global` אומר לפייתון שהשם הוא של המשתנה הגלובלי, ולכן ההשמה משנה אותו. **אפשרי, וממש לא מומלץ:** פונקציה שמשנה משתנים מבחוץ קשה לבדוק ולהבין.

5.3 · קלט ופלט של פונקציה

`return`

`print` מציג ערך ושוכח אותו. `return` **מחזיר** אותו, ואפשר להמשיך לעבוד איתו.

```
>>> def num_of_oranges(small_cups, large_cups):
...     return small_cups + large_cups * 3
...
>>> oranges = num_of_oranges(8, 10)
>>> oranges
38
```

כוס קטנה היא תפוז אחד, וגדולה היא שלושה. `oranges` מצביע על **הערך שהוחזר**, `38` , ולא על הפונקציה.

בלי `return` : `None`

```
>>> def greet_customers(small_cups, large_cups):
...     total = small_cups + large_cups
...     print("Thanks for ordering", total, "cups!")
...
>>> result = greet_customers(2, 1)
Thanks for ordering 3 cups!
>>> print(result)
None
```

כל פונקציה מחזירה משהו. בלי `return` , היא מחזירה את `None` , אובייקט שמשמעותו "כלום". `return` ו- `None` לבד מחזירים את אותו דבר.

כמה ערכים

```
>>> def amount_of_oranges(small_cups, large_cups):
...     oranges = small_cups + large_cups * 3
...     kilograms = oranges / 5
...     return oranges, kilograms
...
>>> oranges_number, kilograms_to_buy = amount_of_oranges(15, 5)
>>> oranges_number
30
>>> kilograms_to_buy
6.0
```

מפרידים בפסיקים ב- `return` , ומקבלים לתוך כמה משתנים, כמו בהחלפת ערכים ביחידה 2. מאחורי הקלעים חוזר **tuple**, שמגיע ביחידה 8.

ערכי ברירת מחדל

```
>>> def amount_of_oranges(small_cups=20, large_cups=10):
...     oranges = small_cups + large_cups * 3
...     return oranges, oranges / 5
...
>>> amount_of_oranges()
(50, 10.0)
>>> amount_of_oranges(3, 4)
(15, 3.0)
```

ארגומנט עם ברירת מחדל נקרא גם **אופציונלי**: אפשר לשלוח אותו, ולא חובה.

ברירת מחדל רק בסוף:

```
>>> def amount_of_oranges(small_cups=20, large_cups):
...     pass
Traceback (most recent call last):
  ...
SyntaxError: parameter without a default follows parameter with a default
```

אם היה מותר, `amount_of_oranges(5)` היה דו-משמעי: האם `5` הולך לראשון או לשני? לכן פרמטרים עם ברירת מחדל באים **אחרי** כל אלה שבלי.

ארגומנטים לפי שם

```
>>> def profit(small_cups, large_cups,
...            small_price, large_price):
...     print(small_cups * small_price
...           + large_cups * large_price)
...
>>> profit(large_price=10, small_price=4,
...        large_cups=3, small_cups=5)
50
```

כשכותבים את שם הפרמטר, **הסדר לא משנה**, והקריאה מסבירה את עצמה. נוח במיוחד כשיש הרבה פרמטרים. אבל כל פרמטר בלי ברירת מחדל חייב לקבל ערך:

```
>>> profit(large_cups=3, small_price=4, large_price=10)
Traceback (most recent call last):
  ...
TypeError: profit() missing 1 required positional argument: 'small_cups'
```

5.4 · תיעוד, main וקונבנציות

docstring

תיעוד אומר מה הפונקציה עושה ואיך משתמשים בה. גם קוד שכתבתם בעצמכם נהיה לא מובן אחרי כמה שבועות. כותבים אותו **מיד אחרי שורת ההגדרה**, במרכאות משולשות:

```
>>> import math
>>> def num_of_bottles(total_liters):
...     """Calculates how many 1.5 L bottles to take.
...     :param total_liters: total water needed
```

```
...     :type total_liters: float
...     :return: the number of bottles
...     :rtype: int
...     """
...     return math.ceil(total_liters / 1.5)
...
>>> num_of_bottles(4)
3
```

שורה	מה כותבים
ראשונה	מה הפונקציה עושה, במשפט
:שם param:	מה הפרמטר. שורה לכל פרמטר
:שם type:	הטיפוס שלו
:return:	מה חוזר. בלי `return` : `None`
:rtype:	הטיפוס של מה שחוזר

(`math.ceil` מעגלת כלפי מעלה, כי שני בקבוקים וחצי הם שלושה בקבוקים. היא מאותה ספרייה של `sqrt` .)

לקרוא את התיעוד

```
>>> print(num_of_bottles.__doc__.splitlines()[0])
Calculates how many 1.5 L bottles to take.
```

`help(num_of_bottles)` מציג את כל התיעוד, ו- `num_of_bottles.__doc__` מחזיר אותו כמחרוזת. שני קווים תחתונים מכל צד.

מעכשיו, תיעוד לכל פונקציה שאתם כותבים, כולל בפתרונות התרגילים.

main

תוכנית שרק מגדירה פונקציות לא מדפיסה כלום: פייתון מגדירה אותן, ומסיימת. צריך לקרוא להן. בקורס עושים את זה מתוך פונקציה ראשית בשם `main` , במבנה הזה:

```
def welcome_hikers():
    print("Welcome to the trail!")


def remind_water():
```

```
    print("Don't forget to drink water.")


def main():
    welcome_hikers()
    remind_water()


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

- `main` היא הראשונה שנקראת כשמריצים את הקובץ, והיא קוראת לכל השאר
- שורת ה- `if` המוזרה נראית מסתורית עכשיו. היא תהיה הגיונית כשתכתבו ספריות משלכם. בינתיים: **הקורס מבקש לכתוב לפי המבנה הזה מעכשיו**
- השם `main` הוא קונבנציה, לא חוק. ובכל זאת, `main`
- **שתי שורות ריקות** בין פונקציות. כך ממליץ PEP 8, ולכן גם בדוגמה

קוד קריא

הסרטון מראה תוכנית שבה המשתנים נקראים `c` , `t` , `ts` , ואי אפשר להבין מה היא עושה. אותה תוכנית בדיוק, עם שמות ברורים ותיעוד, מתגלה כתחזית מזג אוויר לעיר שהמשתמש מקליד. **ההבדל הוא רק בשמות.**

ארבעה כללים

1. **מטרה אחת לכל פונקציה.** פונקציה ממוקדת אפשר להשתמש בה בהרבה מקומות
2. **שם שאומר מה היא עושה**, ותיעוד שתואם לו
3. **קלט ופלט ברורים**
4. **שורה ריקה בין פונקציות** (שתיים, לפי PEP 8)

בדיקה עצמית

1. מה ההבדל בין `print(x)` לבין `return x` בתוך פונקציה?
2. מה יודפס? `def f(): x = 1` ואז `f()` ואז `print(x)` .
3. מה מחזירה פונקציה שאין בה `return` ?
4. האם מותר `def g(a, b=2, c):` ?
5. במה `square` שונה מ- `square()` ?

תשובות. 1: `print` מציג ושוכח. `return` מחזיר ערך שאפשר לשמור ולהמשיך איתו. 2: `NameError` . `x` לוקאלי, וקיים רק בתוך `f` . 3: `None` . 4: לא. אחרי פרמטר עם ברירת מחדל, גם `c` חייב לקבל ברירת מחדל. 5: `square` הוא הפונקציה עצמה, כאובייקט. `square()` מריץ אותה.

סיכומים אישיים מתוך סרטוני הקורס Self.py בקמפוס IL, ולא חומר רשמי של הקורס.