סיכום קורס עקרונות שפות תכנות נכתב ע"י ולדיס מרקין.

Python

אלמנטים של שפות תכנות:

- נתונים (Data) – דברים שאנו מעוניינים לתפעל.
- תהליכים (Procedure) – חוקים לתפעול נתונים.
- הצהרות וביטויים פרימיטיביים (בסיסיים) (Primitive Expressions and statements).
- ביטויים מורכבים / שילובים (Compound expressions/combinations).
- ביטויים מופשטים (Abstracting expressions).

הצהרה / ביטוי

בשפה יש הבדל בין הצהרה לביטוי.
ביטוי – עובר הערכה, המפרש מעריך את הערך של הביטוי לפני שהוא עובר לפרמטר החדש.
הצהרה – להצהרה אין ערך, היא רק מתבצעת. המפרש לא מתעלם ממנה למרות שאנו לא רואים תוצאה מיידית.
***ההצהרה משפיעה על הסביבה (למשל השמה – תפיסת זיכרון).**

```
>> 1+1 – ביטוי
>> x = 1+2 – הצהרה
```

הערך המוחזר של ביטוי הוא תוצאת הביטוי, הערך המוחזר של הצהרה הוא None.
ביטויים קבועים (*Constant* expressions) - הם הערכה עצמית. הערך של קבוע הוא עצמו.

```
>> 56
56
```

ביטויים מורכבים (Compound expressions) – הם ביטויים שעשויים מביטויים פשוטים יותר.

```
>> 1/2 + 1/4 + 1/8 + 1/16 + 1/32 + 1/64 + 1/128
0.9921875
```

ביטויי קריאה (Call Expressions) - הסוג החשוב ביותר של ביטוי מורכב. מפעיל פונקציה על כמה ארגומנטים:

```
>> max(7.5, 9.5)
9.5
>> pow(100, 2)
10000
>> pow(2, 100)
1267650600228229401496703205376
```

פונקציות מסוגלות לקחת מספר שרירותי של ארגומנטים:

```
>>> max(1, -2, 3, -4)
3
```

הרחבה בדרך פשוטה ביטויים מקוננים, כאשר האלמנטים עצמם ביטויים מורכבים:

```
>>> max(min(1, -2), min(pow(3, 5), -4))
-2
```

כללי הערכה של הרכבות:

1. הערכת אופרטור והפעלת האופרטור על הארגומנטים.
2. הערכת ארגומנטים במידת הצורך.

Applicative and Normal order

שתי אלגוריתמים עיקריים שך הערכה ששפות תוכנה משתמשות. ההבדל העיקרי בין השתיים טמון בעת הערכת הארגומנטים. ב Applicative order הארגומנטים מוערכים ישירות, בניגוד ל normal order אשר מעקבת את ההערכה של הארגומנטים עד שיש בהם צורך.
פייטון עובדת על עקרון Normal order. ההערכה מתבצעת באופן הבא:

- בהערכת הרכבות קודם כל מעריכים את האופרנדים ולאחר מכן את האופרטור.
- בהערכת המשתנה – סורקים את הסביבה על מנת למצוא איפה בסביבה המשתנה נמצא.
- בהערכת אופרטור – מחפשים בסביבה משתנה בשם add לדוגמה, ומפעילים את הפונקציה על הארגומנטים.
- בהערכת קריאה לפונקציה (call expration):
  1. הערכת אופרנדים (ארגומנטים)
  2. הערכת אופרטור (פונקציה)
  3. הפעלת האופרטור על האופרנדים.

  ולאחר מכן, הצבת הערכים המופיעים בקריאה לפונקציה במקום המשתנים בגוף הפונקציה, והערכת הביטוי או ביצוע ההצהרה המתקבלת ביחס לסביבה הנוכחית.

```
>> add( div (10,0), 5)
>>def func(x,y)
        Return y
```

ב Applicative order התוכנית תיפול בהערכת המשתנה div(10,0) כי הפעולה לא תקינה (לא ניתן לחלק ב-0).
ב Normal order הפונקציה תחזיר y = 5. לא תתבצע הערכה ל-x מכיוון שהוא לא שמיש בתוכנית, כלומר אין בו שום צורך.

מודול סביבות

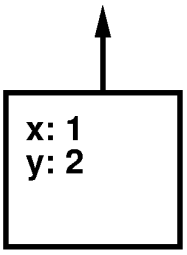
לכל משתנה יש מיקום מוגדר בסביבה. הערך של המשתנה תלוי במיקום שלו. כל מפרש מנהל סביבה לפי מודול הסביבה.

לביטוי יש משמעות אך ורק ביחס לסביבה מסויימת שמכונה הסביבה הנוכחית.

- מתייחסים למשתנים לפי מיקום.
  - יוצר שיוך (קשירה של ערך לשם המשתנה)
- כל משתנה נשמר בצמד.

קשירה – צמד ששמור שם משתנה יחד עם ערכו.

<u>מסגרות</u>

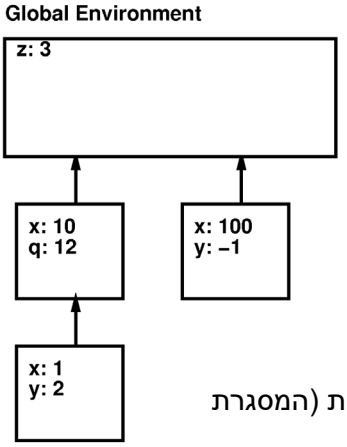


- כל מסגרת (חוץ ממסגרת גלובאלית) מצביעה לסביבה הגלובאלית.
  כל מסגרת היא נקודת התחלה של סביבה מסויימת.
- מסגרת אחת יכולה להכיל מספר קשירות.
- רצף של מסגרות = סביבה.

<u>סביבה – Environment</u>

סביבה – רצף של מסגרות מקושרות שמתחילות ממסגרת ספציפית ומסתיימות בסביבה הגלובאלית. הסביבה שומרת את כל המשתנים (כולל פונקציות) שהוגדרו במהלך ריצה, עם ערכים מעודכנים.

מטרת הסביבה: לאפשר למפרש "להגיע" לערכים מעודכנים של משתנים שהוגדרו (כשמדובר על הערכה) ולאפשר לשמור שינויים כשמדובר על ביצוע הצהרות.



דיאגראמות אלה מכונות "דיאגראמות סביבה", הן יכולות להכיל מספר סביבות (4 בדוגמה שלנו)

- כל מסגרת היא נקודת התחלה של סביבה מסוימת.
- **אך ורק הפעלה של פונקציה יכולה להרחיב סביבה קיימת.**

<u>מבנה סביבה:</u>

- רצף של מסגרות שמתחיל בסביבה הנוכחית והגמר בסביבה הגלובאלית.

<u>חיפוש בסביבה:</u>

חיפוש בסביבה הוא תהליך רקורסיבי שמתחיל במסגרת גלובאלית הנוכחית (המסגרת התחתונה של הסביבה הנוכחית) ונשמר באחד מהשניים:

1. מצאנו קשר עם שם המשתמש אותו אנו מחפשים
2. לא מצאנו את הקשירה (גם לא במסגרת הגלובאלית), נקבל הודעת שגיאה "המשתנה לא מוגדר".

- אם שם המשתנה נמצא במסגרת הנוכחית, אז ערכו נלקח מהשיוך במסגרת זו.
- אם השיוך של ערך לשם המשתנה לא נמצא במסגרת הנוכחית,
- נחפש אותו במסגרת אחת למעלה (parent frame).

דיאגרמות סביבה:

- הן עצים הבנויים מרשימות מקושרות
- מנקודת מבט של כל מסגרת אפשר לראות את סביבה כרשימה מקושרת (*linked list*)

מודל סביבות להערכה:

לפי מודל סביבות, הערכה של כל ביטוי נעשית ביחס לסביבה מסוימת המכונה "סביבה נוכחית" לביטוי זה.

מודל סביבות: השמה (assignment):

- לפי ברירת מחדל ב Python, ההשמה היא תמיד לוקאלית!
- אין הבדל תחבירי בין הכרזת משתמש חדש לבין פעולת השמה.
- השמה **לא** יכולה לשנות את הקשירה בסביבה הכוללת.

פונקציות:

ישנן 2 סוגים של פונקציות.

Pure – מחזירה ערך. (יש return בסוף הפונקציה).

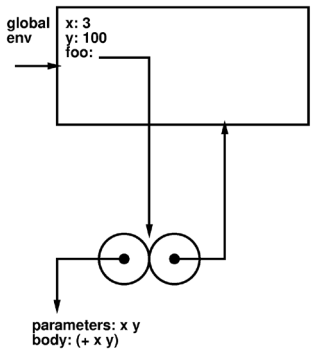
Non-pure – לא מחזירה ערך. לדוגמה: print(1).

- פונקציות הן הפשטות המתארות פעולות מורכבות ללא תלות בערכים של הארגומנטים שלהם.
- פונקציה היא צמד שמכיל את הטקסט של הפונקציה ומצביע לסביבה בה היא הוגדה.
- Def יוצר קשירה (שיוך) חדשה בסביבה הנוכחית בין שם הפונקציה לבין הפונקציה עצמה.
- טקסט של פונקציה מחולק לשני חלקים. פרמטרים (חתימת הפונקציה) וגוף (תוכן הפונקציה).
- הפעלת פונקציה מרחיבה את הסביבה, נוצרת מסגרת חדשה.
- הפעלת פונקציה על האופרנדים שלה:
  - יוצרת מסגרת חדשה
  - מבצעת קשירה של פרמטרים פורמאליים לארגומנטים המופיעים בקריאה לפונקציה במסגרת החדשה.
  - הערכת הגוף של הפונקציה ביחס לסביבה שמתחילה מהמסגרת החדשה.
- **חשוב מאוד!!! המסגרת החדשה מצביעה לסביבה שבה הוגדרה פונקציה המופעלת ולא פונקציה שמפעילה אותה!!! המיקום של הפעלת הפונקציה לא רלוונטי.**
- **שם לא נחשב כחלק מפונקציה!**
- רצף ההצהרות מתבצע באופן רקורסיבי, מתחילים בפקודה הראשונה ומיד לשניה.
- המילה השמורה – global מתייחסת אך ורק למסגרת גלובלית. מומלץ לא לעבוד עם משתמשים כאלו אלה אם כן אין ברירה.

def f(x, y): return add(x, y)

Parameters: x, y

Body: return add(x, y)



פונקציות מסדר גבוה

פונקציות מסדר גבוה פונקציות ושיטות הם אובייקטים ממדרגה ראשונה בפייתון, כך שאם אתה רוצה להעביר פונקציה לפונקציה אחרת, אתה יכול פשוט להתייחס אליה כמו אל כל אובייקט אחר. פונקציות שמסוגלות לקבל פונקציות מסדר גבוה. –פונקציות אחרות כארגומנט נקראות.

Lexical scoping vs Dynamic scoping

Lexical scoping פירושו שהסביבה של פונקציה תוגדר ע"י המרחב שבו היא כתובה. אם פונקציה מקוננת בתוך שתי פונקציות, היא תחפש את המזהה בגבולות ההגדרה שלה עצמה, ולאחר מכן תחפש בהגדרה של הפונקציה המכילה אותה, לאחר מכן תעלה לפונקציה המכילה את שתיהן ולבסוף תחפש במרחב הגלובלי (Global scope)

Dynamic scoping פירושו שהסביבה של פונקציה תוגדר ע"י היררכיית הקריאות במחסנית. כדי למצוא מזהה מסוים, פונקציה תחפש אותו בגבולות ההגדרה שלה, ואז תנסה לטפס במעלה המחסנית ולחפש בסביבה של הפונקציה שקראה לה, וזו שקראה לה וכדומה.

Lambda

פונקציה חד פעמית, אנונית ללא שם. פונקציות אלה נקראות אנונימיות משום שהן אינן מוכרות באופן סטנדאטי באמצעות מילת המפתח def. פונקציות למדה מסוגלות לקחת מספר של ארגומנטים אך להחזיר רק ארגומנט אחד. הן אינן יכולות להכיל פקודות או ביטויים מרובים.

פונקציות למדה לא יכולות להוות קריאה ישירה להדפסה מכיוון שהן דורשות ביטוי.

במידה ונרצה להחזיר פונקציה מפונקציה מורכבת, נוכל להיעזר בלמדה.

לפונקציות למבדה יש מרחב שמות משלהן ואינן יכולות לגשת למשתנים אחרים מאלה ברשימת הפרמטרים שלהן ולאלו במרחב השמות הגלובלי.

```
>> f = lambda x: x+2

>> f(2)

>> 4
```

הפשטת נתונים

ניתן לתפעל פונקציות כמו נתונים. למשל – להעביר כפרמטר לפונקציה ולהחזיר פונקציה מפונקציה.

אובייקט בפייטון:

```
Someday = date(2018, 11, 23)
```

כאשר: sumday הוא שם האובייקט, date בנאי ו2018, 11, 23 הם השדות של האובייקט.

כל אובייקט בפייטון צריך לתמוך בפונקציות str שהיא פונקציה המקבלת אובייקט ומחזירה מחרוזת נתונים על האובייקט שהועבר. ניתן לדעת מה טיפוס האובייקט באמצעות הפונקציה ()type.

עקרון הפשטת נתונים:

הרעיון הבסיסי של הפשטת נתונים הוא הפרדה בין המימוש לשימוש של הטיפוס. כלומר, אין צורך להכיר את המימוש של הטיפוס כדי להשתמש בטיפוס, יש להכיר רק את הממשק של הטיפוס המאפשר שימוש בו.
המשמעות – אם הבצע עדכון גרסה והמימוש של הטיפוס השתנה – הקוד שנכתוב לא ישתנה כתוצאה מכך.

API – ממשק. אוסף התכונות המוגדרות על הטיפוס.

https://composingprograms.com/pages/22-data-abstraction.html - דוגמה מההרצאה לגבי הפשטת נתונים.

כל רמה עליונה משתמשת ברמות התחתונות מבלי להכיר את המימוש שלהם. השימוש ברמה התחתונה זה שימוש בטיפוסים מובנים של פייטון שאנו לא מכירים את מימושם.

מבני נתונים ב Python:

מבנה הנתונים הבסיסי ביותר בפייטון הוא רצף. לכל אלמנט ברצף מוקצה מספר – מיקומו ברצף (האינדקס שלו)

לפייטון יש שישה סוגים מובנים של רצפים אך המוכרים ביותר הם Lists ו Tuples.

ישנן פעולות מסויימות שאנחנו יכולים לעשות על רצף. פעולות אלה כוללות:

1. **אורך (length)** – לכל רצף יש אורך סופי.
2. **בחירת אלמנט (element selection)** – ניתן לגשת לכל אלמנט ברצף.
3. **פעולות אריטמטיות (+ ו *)** – שרשור רצפים שמחזיר רצף חדש.
4. **מיפוי (mapping)** – ניתן להפעיל כל פונקציה על אלמנט של רצף ולקבל רצף חדש שמורכב מאלמנטים חדשים. קיימת פונקציה map שמקבלת פונקציה ויוצרת map object שניתן להמיר לרצף.
5. **סינון (filering)** – ניתן להפעיל פונקצית סינון על אלמנטים של רצ, ולקבל filter object שניתן להמיר לרצף המורכב אך ורק מאלמנטים "התקינים" (פונקצית סינון מחזירה עבורם True).
6. **חברות (membership)** – ניתן לבדוק שייכות של ערך לרצף.
7. **חיתוך (slicing).**

תכונות 1-2 מאפיינות כל רצף. תכונות 3-8 מאפיינות רצף מלא.

Tuple

רצף מלא של שלא ניתן לשינוי (**immutable**) של עצמים. ההבדלים בין Tuple ו Lists הם ש Tuples לא ניתנים לשינוי והם ממומשים ע"י סוגריים עגולות בניגוד ל Lists שניתנים לשינוי וממומשים ע"י סוגריים מרובעות.

בשביל ליצור Tuple עלינו להפריד את הערכים בעזרת סימן פסיק. ניתן גם ליישם Tuple ע"י שימוש בגרשיים:

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

מימוש Tuple ריק: tup1 = () .

כדי לרשום Tuple שמכיל אך ורק ערך אחד, עלינו להפריד ערך זה בעזרת פסיק – tup2=(50,)

גישה לערכים:

ניתן לגשת לערכים ב Tuple ע"י סוגריים מרובעות ובתוכן האינדקס של המיקום במבוקש, כמו כן ניתן גם לעשות Slicing.

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );
tup1[0]:  physics
tup2[1:5]:  [2, 3, 4, 5]
```

עדכון:

Tupple לא ניתנים לשינוי (immutable) מה שאומר שאנו לא יכולים לעדכן אותם או לשנות ערכים של האלמנים. אנו יכולים ליצור העתק חדש של Tupple קיים ע"י מספר פעולות בסיסיות:

```
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');
tup3 = tup1 + tup2;
print tup3;

(12, 34.56, 'abc', 'xyz')
tup4 = 2*tup2;
print tup4;
(12, 34.56, 'abc', 'xyz', 12, 34.56, 'abc', 'xyz')
```

מחיקת איברים:

בעוד שלא ניתן למחוק איברים מה Tupple, אין שום דבר רע בלשים את האיברים שאנו **לא** רוצים למחוק ב Tupple חדש. בכדי למחוק רשימה שלמה, נשתמש במילה השמורה del.

```
tup = ('physics', 'chemistry', 1997, 2000);
del tup;
```

פעולות בסיסיות על Tuples:

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes:

בגלל ש Tuples הם רצף, indexing ו slicing יעבדו באותה צורה כמו על strings.

```
L = ('spam', 'Spam', 'SPAM!')
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

פונקציות מובנות עבור Tuples:

מס'	פונקציות עם הסבר:
1	**cmp(tuple1, tuple2)** - ביצוע השווה בין 2 Tuples.
2	**len(tuple)** - נותן את אורך ה Tuple.
3	**max(tuple)** - מחזיר איבר מקסימלי.
4	**min(tuple)** - מחזיר איבר מינימלי.
5	**tuple(seq)** - ממיר רשימה ל Tuple.

Tuple Methods:

Count – מחזיר את מספר המופעים של אלמנט

Index – מחזיר את מאינדקס של האלמנט

Lists

רשימה היא רצף מלא של שניתן לשינוי (**mutable**) של עצמים. היא סוג הנתונים הרב – תכליתי ביותר בשפה. רשימה מופרדת בפסיקים בין סוגריים מרובעים. הדבר החשוב ביותר ברשימה הוא שהאלמנטים לא חייבים ליהיות מאותו הסוג.

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"]
```

באופן דומה למחרוזות, רשימות מתחילות באינדקס 0 והן יכולות ליהיות פרוסות (slicing) ומשורשרות וכך הלאה.

גישה לערכים:

ניתן לגשת לערכים ברשימה ע"י סוגריים מרובעות ובתוכן האינדקס של המיקום במבוקש, כמו כן ניתן גם לעשות Slicing.

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
print "list1[0]: ", list1[0]
print "list2[1:5]: ", list2[1:5]
```

עדכון:

ניתן לעדכן אלמנטים בודדים או מרובים של רשימות ע"י השמה של ערך עם מיקום האינדקס הרצוי או בעזרת מטודה ()append.

```
list = ['physics', 'chemistry', 1997, 2000];
print "Value available at index 2 : "
print list[2]
list[2] = 2001;
print "New value available at index 2 : "
print list[2]
```

מחיקת איברים:

על מנת למחוק איבר ברשימה אפשר להשתמש במילה השמורה del אם אתה יודע בוודאות איזה אלמנט מהרשימה אתה מוחק או במתודת ()remove אם אתה לא יודע איזה איבר אתה מוחק.

```
list1 = ['physics', 'chemistry', 1997, 2000];
print list1
del list1[2];
print "After deleting value at index 2 : "
print list1
```

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

פעולות בסיסיות על רשימות:

רשימות מגיבות לאופרטורים + ו * כמו מחרוזות; התוצאה רשימה חדשה ולא מחרוזת.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

מאחר והרשימות הן רצפים, ניתן לבצע אינדקסינג ופריסה כמו במחרוזות:

```
L = ['spam', 'Spam', 'SPAM!']
```

Python Expression	Results	Description
L[2]	SPAM!	Offsets start at zero
L[-2]	Spam	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

פונקציות ומטודות מובנות עבור רשימות:

מס'	פונקציות עם הסבר:
1	**cmp(list1, list2)** - ביצוע השווה בין 2 רשימות.
2	**len(list)** - נותן את אורך הרשימה
3	**max(list)** - מחזיר איבר מקסימלי.
4	**min(list)** - מחזיר איבר מינימלי.
5	**list(seq)** - ממיר Tuple לרשימה.

מס'	מתודות עם הסבר:
1	**list.append(obj)** – הוספת איבר לרשימה.
2	**list.count(obj)** – החזרת מספר המופעים של אלמנט ברשימה.
3	**list.extend(seq)** – הוספה לרשימה קיימת איברים של רשימה אחרת.
4	**list.index(obj)** – מחזיר את האינדקס הנמוך ביותר של האיבר ברשימה.
5	**list.insert(index, obj)** – הוספת אובייקט לרשימה ובחירת האינדקס בו הוא יהיה.
6	**list.pop(obj=list[-1])** – מוחק ומחזיר את האובייקט האחרון ברשימה.
7	**list.remove(obj)** – מוחק אובייקט מהרשימה.
8	**list.reverse()** – הופך את הרשימה.
9	**list.sort([func])** – מסדר את המערך בעזרת פונקצית השוואה (במידה והפונקציה ניתנת).

דוגמאות:

```
>>> digits = [1, 8, 2, 8]
>>> len(digits)
4
>>> digits[3]
8
```

```
>>> [2, 7] + digits * 2
[2, 7, 1, 8, 2, 8, 1, 8, 2, 8]
```

```
>>> pairs = [[10, 20], [30, 40]]
>>> pairs[1]
[30, 40]
>>> pairs[1][0]
30
```

```
>>> pairs = [[1, 2], [2, 2], [2, 3], [4, 4]]
>>> same_count = 0

>>> for x, y in pairs:
        if x == y:
            same_count = same_count + 1
>>> same_count
2
```

```
>>> list(range(5, 8))
[5, 6, 7]


>>> list(range(4))
[0, 1, 2, 3]


>>> odds = [1, 3, 5, 7, 9]
>>> [x+1 for x in odds]
[2, 4, 6, 8, 10]


>>> [x for x in odds if 25 % x == 0]
[1, 5]


>>> def divisors(n):
        return [1] + [x for x in range(2, n) if n % x == 0]
>>> divisors(4)
[1, 2]
>>> divisors(12)
[1, 2, 3, 4, 6]


>>> [n for n in range(1, 1000) if sum(divisors(n)) == n]
[6, 28, 496]
```

Global Variables

משתנה המוצהר מחוץ לפונקציה או בסביבה הגלובלית נקרא משתנה גלובלי. ניתן לגשת אליו בתוך או מחוץ לפונקציה.

```
x = "global"

def foo():
    print("x inside :", x)
foo()
print("x outside:", x)
```

```
x inside : global
x outside: global
```

מה אם נרצה לשנות את הערך של x בתוך הפונקציה?

```
x = "global"

def foo():
    x = x * 2
    print(x)
foo()
```

נקבל הודעת שגיאה. הפלט מראה שגיאה משום ש- Python מתייחס ל- x כמשתנה מקומי ו- x אינו מוגדר בתוך foo (). בפעולה ← x = x * 2 הקוד נפל. לא ניתן לשנות את ערכו של x.

```
UnboundLocalError: local variable 'x' referenced before assignment
```

בשביל שכן נוכל לעשות זאת, ניתן להשתמש ב:

Python Global Keyword

בפייתון, מילת המפתח Global מאפשרת לך לשנות את המשתנה מחוץ לסביבה הנוכחית. היא משמשת ליצירת משתנה גלובלי ולשינויים במשתנה בהקשר מקומי.

הבסיסיים של מילת מפתח גלובלית בפייתון הם

1. כאשר אנו יוצרים משתנה בתוך פונקציה, הוא מקומי כברירת מחדל .
2. כאשר אנו מגדירים משתנה מחוץ לפונקציה, הוא גלובלי כברירת מחדל. אינך צריך להשתמש במילת מפתח גלובלית.
3. אנו משתמשים במילת מפתח גלובלית כדי לקרוא ולשנות ערך של משתנה גלובלי בתוך פונקציה.
4. לשימוש במילת מפתח גלובלית מחוץ לפונקציה אין כל השפעה.

```
c = 1 # global variable

def add():
    print(c)

add()
```

```
1
```

עם זאת, ייתכן שיהיה לנו כמה תרחישים שבהם אנחנו צריכים לשנות את המשתנה הגלובלי בתוך פונקציה.

```
c = 1 # global variable

def add():
    c = c + 2 # increment c by 2
    print(c)

add()
```

נריץ ונקבל:

```
UnboundLocalError: local variable 'c' referenced before assignment
```

הסיבה לכך היא שאנחנו יכולים רק לגשת למשתנה הגלובלי אבל לא ניתן לשנות אותו מתוך הפונקציה. הפתרון לכך הוא להשתמש במילת המפתח global.

```
c = 0 # global variable

def add():
    global c
    c = c + 2 # increment by 2
    print("Inside add():", c)

add()
print("In main:", c)
```

```
Inside add(): 2
In main: 2
```

כפי שניתן לראות, השינוי התרחש גם במשתנה הגלובלי מחוץ לתפקוד, c = 2.

Local Variables

משתנה המוצהר בתוך פונקציה או בסביבה לוקאלית מוגדר כמשתנה לוקאלי.

```
def foo():
    y = "local"

foo()
print(y)
```

```
NameError: name 'y' is not defined
```

הפלט מראה שגיאה, מכיוון שאנו מנסים לגשת למשתנה מקומי y במסגרת הגלובלית, בעוד שהמשתנה המקומי פועל רק בתוך foo () או במסגרת מקומית.

```
def foo():
    y = "local"
    print(y)

foo()
```

```
local
```

None Local

מילת המפתח nonlocal משמשת לעבודה עם משתנים בתוך פונקציות מקוננות, כאשר המשתנה לא שייך לפונקציה הפנימיה.

```
def outer():
    x = "local"
    def inner():
        nonlocal x
        x = "nonlocal"
        print("inner:", x)
    inner()
    print("outer:", x)
outer()
```

```
inner: nonlocal
outer: nonlocal
```

במידה ונשנה את ערך המשתנה nonlocal, השינויים ישפיעו על המשתנה המקומי.

Global and local variables

כאן נראה כיצד להשתמש במשתנים גלובליים ובמשתנים מקומיים באותו קוד.

```
x = "global"

def foo():
   global x
   y = "local"
   x = x * 2
   print(x)
   print(y)
foo()
```

```
global global
local
```

בקוד לעיל, אנו מצהירים על x כגלובלי ו- y כמשתנה מקומי ב- foo (). לאחר מכן, אנו משתמשים באופרטור הכפל כדי לשנות את המשתנה הגלובלי x ואנו מדפיסים את x ו- y.

לאחר קריאת foo (), הערך של x הופך לglobal global, כי השתמשנו ב 2 * x להדפיס פעמיים global. לאחר מכן, אנו מדפיסים את הערך של המשתנה המקומי y כלומר local.

```
x = 5

def foo():
   x = 10
   print("local x:", x)

foo()
print("global x:", x)
```

```
local x: 10
global x: 5
```

בקוד לעיל, השתמשנו באותו שם x עבור המשתנה הגלובלי והמשתנה המקומי. אנו מקבלים תוצאה שונה כאשר אנו מדפיסים אותו המשתנה מכיוון שהמשתנה הוכרז בשני מסגרות, כלומר, מסגרת מקומית בתוך foo () ומסגרת גלובלי מחוץ foo ().

כאשר אנו מדפיסים את המשתנה בתוך foo () הפלט הוא x: 10, זה נקרא מסגרת מקומית של משתנה.

באופן דומה, כאשר אנו מדפיסים את המשתנה מחוץ ל- foo (), ערכו של x הוא x: 5, זה נקרא מסגרת גלובלי של משתנה.