# תחביר בסיסי

## הגדרה

פייתון היא שפת תכנות כללית High-Level. נוצרה על ידי גווידו ואן רוסום בשנים 1985 – 1990. נקראת על שם סדרת הילדים "הקרקס המעופף של מונטי-פייתון".

### יתרונות

* קוד פתוח.
* מבוססת על Interpreter – ניתן לקמפל ולהריץ את התוכנית שורה אחר שורה בלי לקמפל את כל התוכנית לפני ההרצה. היתרון הוא שלא צריך לקמפל את התוכנית ומגיעים לשלב ההרצה יותר מהר. החיסרון הוא שביצוע כל שורה לוקח יותר זמן מאשר תוכנית שקומפלה לפני הרצה. יש אפשרות גם לקמפל בשביל הרצה מהירה.
* אינטראקטיבית - ניתן לכתוב ולהריץ כל שורה בנפרד באמצעות ה-Interpreter.
* תמציתית וקריאה - בדרך כלל דורשת הרבה פחות שורות קוד משפות אחרות.
* דינאמית - אין צורך להכריז על משתנים.
* מונחית עצמים.
* נפוצה מאוד בתחומים רבים, כמו: פיתוח רשת, ניתוח מסדי נתונים, למידת מכונה ועוד.

## Interpreter לעומת Compiler

ישנם שני סוגים של מימושי קוד: מימוש קוד באמצעות מהדר (compiler) ומימוש באמצעות מפרש (interpreter). מימוש מהדר הוא מימוש הקוד ע"י "הידורו" לשפת מכונה, כלומר הקוד המתקבל עובר תהליך של קומפילציה שמחזיר קוד הכתוב בשפת מכונה שמתבצע.

מימוש מפרש הוא מימוש שעובר שורה אחר שורה בקוד, מריץ אותה, ושומר את הנתונים בזיכרון לשימוש מאוחר יותר. הקוד מופעל ישירות ולא עובר איזשהו תהליך סריקה מקדים מה שמאפשר ביצוע של קטע קוד ספציפי מתוך התוכנית, מבלי לעבור על כל הקוד קודם לכן. תכונה זו מאפשרת לנו לכתוב קוד בצורה אינטראקטיבית, כלומר לאחר כתיבת כל שורה ניתן ישר לקבל תוצאה. אין צורך בפונקציית main בשפות מסוג זה.

## גרסאות

החל מ-2020 פייתון הסירה את התמיכה שלה מ-python2 ומספקת תמיכה אך ורק ב-python3. ניתן לבדוק את הגרסה של פייתון שאנו עובדים בה על ידי הרצה של הפקודה python--version בטרמינל.

## בלוקים, הזחות והצהרות

בפייתון, המבנה של בלוקים בתוכנית נקבע על ידי ההזחה שלהם. עצם הזחת הבלוק מגדירה אותו כבלוק תכנותי שונה. המטרה היא לשפר את קריאות התוכנית, בכך שהיא כופה על המתכנת כתיבה בסטנדרט אחיד (פחות או יותר), וחוסכת בקוד. כל הזחה גדולה או שווה ל-2 היא חוקית כל עוד אנו עקביים לכל השורות בבלוק. מומלץ להגדיר את tab להיות 4 רווחים כי אורך ה-tab יכול להיות שונה בין מערכות הפעלה.

פקודות מופרדות לרוב על ידי מעבר שורה. שימוש בתו ';' (נקודה פסיק) להפרדה בין פקודות הוא אופציונלי וכמעט שלא בשימוש בפועל. נקודה פסיק ';' מאפשרת מספר הצהרות בשורה הבודדת בהינתן שאף משפט לא מתחיל בלוק קוד חדש.

x = 'foo'; sys.stdout.write(x + '\n')

לבלוקים של קוד בלולאות, תנאים, פונקציות, מחלקות וכדומה, מקדימים תמיד את התו ':' (נקודתיים).

if True:

print "True"

else:

print "False"

כדי להפריד הצהרת קוד אחת למספר שורות ניתן להשתמש בתו '\'. אמנם הצהרה בתוך סוגריים מסוג [], {}, () אינם צריכים תו זה וניתן להסתפק בירידת שורה.

total = item\_one + \

item\_two

## הערות

הערות בפייתון מתחילים עם סימון #. ניתן להוסיף הערות בשורה נפרדת או בסוף שורת קוד.

# This is a comment.

עבור הערה עם מספר שורות, יש להקיף אותה בשלושה גרשים.

'''

This is a multiline

comment.

'''

נהוג להתחיל קובץ פייתון בכמה שורות של הערות המציינות ידע 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)

#------------------------------------------------------------------------------------

סוג הערות נוסף הוא docstring. אלו הערות שמוסיפים מתחת לכותרת של פונקציה, מחלקה, שיטה של מחלקה או module.

ההערה תיכנס לתוך השדה \_\_doc\_\_ של הפונקציה או המחלקה. ניתן לקבל את הערת ה-docstring גם באמצעות פונקציית help שלתוכה נכניס את שם הפונקציה או המחלקה עליה אנו רוצים לקבל מידע.

def add(value1, value2):

"""Calculate the sum of value1 and value2."""

return value1 + value2

print (add.\_\_doc\_\_)

help(add)

## משתנים

בפייתון אין צורך להגדיר טיפוס למשתנה, אלא אפשר לרשום ישירות את שם המשתנה ואת הערך ההתחלתי שאנו נותנים לו. הצהרה מסוג זה נקראת dynamic typing. ה-interpreter מבין לבדו לפי ההשמה על המשתנה מהו הטיפוס שלו. היתרון של שפות דינמיות הוא שהן יותר תמציתיות, פשוטות וקריאות.

לאחר יצירת משתנה ניתן לדרוס את הטיפוס שלו ולתת לו השמה מטיפוס אחר. ניתן לגלות את הטיפוס הנוכחי של משתנה באמצעות פונקציית type.

x = 3 #‘int’

x = 3.0 #‘float’

ניתן להצהיר על כמה משתנים באותה שורה בדרך הבאה:

t, f = True, False

## שמות מזהים

מזהים ב-python הם שמות המשמשים לזיהוי משתנה, פונקציה, מחלקה, מודול או אובייקט אחר. מזהה מתחיל באותיות A-Z או a-z או קו תחתון. לא ניתן להשתמש בסימני פיסוק, כגון @, $ ו-%, בתוך מזהים. Python היא שפת case sensitive. להלן מספר כללי עיצוב קוד לשמות מזהים:

* שמות משתנים, פונקציות ומודולים יכתבו באותיות קטנות ועם קו תחתון בין מילים - variables\_and\_functions.
* שמות ערכים קבועים יכתבו באותיות גדולות בלבד ועם קו תחתון בין המילים - UPPERCASE\_CONSTANTS.
* שמות מחלקות וחריגות יחוברו עם אות גדולה בין מילים - ClassName.
* קו תחתון לפני שם מזהה מציין שזהו מזהה פרטי של מחלקה. שני קווים תחתונים לפני שם מזהה מציין מזהה פרטי חזק. אם בנוסף גם מסיימים את שם המזהה עם שני קווים תחתונים, המזהה הוא מילה שמורה המוגדרת בשפה.
* משתנים שלא משתמשים בהם, כמו לדוגמא בלולאה, נסמן בקו תחתון '\_'. לדוגמה, for \_ in range(10).

## טיפוסים בסיסיים

### Numbers

* int - מאחסן מספר שלם. בגרסה 3.0 אוחד טיפוס זה אל תוך long ואין לו הגבלת גודל (מעבר למה שמאפשר הזיכרון הווירטואלי של המחשב), בדומה ל־BigInteger בשפת Java.
* float - נקודה צפה. מאפשר אחסון ערך מקורב של מספר ממשי באמצעות ייצוג נקודה צפה. טיפוס זה זהה לטיפוס double בשפת C (ולמעשה ברוב המקרים ממומש על ידיו). בנוסף, ניתן לייבא את המודול decimal המאפשר ייצוג של שברים עשרוניים, ונוח יותר לעבודה במקרים בהם הייצוג הזה מדויק יותר (כגון חשבונאות).
* complex - מספר מרוכב, נכתב בצורה 4+3j. טיפוס זה נוסף לשפה בשל השימוש הרב שעושים בו מדענים ומהנדסים, שהם אחת מהקהילות העיקריות המשתמשות בשפה, ועל מנת למנוע מהם "להמציא את הגלגל" כל פעם מחדש.

### string

מחרוזות בפייתון מזוהות כקבוצה רציפה של תווים המיוצגים במרכאות. פייתון מקבל מרכאות בודדות ('), כפולות (") ומשולשות (''' או """) לציון מחרוזת, כל עוד אותו סוג של מרכאות מתחיל ומסיים את המחרוזת. החל מגרסה 3, המחרוזות הן Unicode, אם כי ישנו טיפוס bytes עבור רצפים של בתים.

אובייקטים מטיפוסי מספרים ומחרוזות הם בלתי ניתנים לשינוי (Immutable), כלומר שינוי ערכו של משתנה מטיפוס כזה הוא בעצם הסרת ההתייחסות אליו והפנייתה לערך חדש.

### bool

טיפוס בוליאני שיכול להחזיק True או False (עם אות גדולה בהתחלה בניגוד ל-Java ו-C++).

### רצפים

* List - רשימות הן המגוונות ביותר מבין סוגי הנתונים המורכבים של Python. רשימה מכילה פריטים מופרדים בפסיקים ומוקפים בתוך סוגריים מרובעים []. במידה מסוימת, רשימות דומות למערכים ב-C. הבדל אחד ביניהן הוא שכל הפריטים השייכים לרשימה יכולים להיות מסוג נתונים שונה.
* Tuple – בדומה לרשימה, מורכב ממספר ערכים המופרדים בפסיקים. ההבדלים העיקריים בין רשימות ל-tuples הם: רשימות מוקפות בסוגריים מרובעות [ ] וניתן לשנות את האיברים והגודל שלהן, בעוד ש-tuples מוקפים בסוגריים עגולות ( ) ואינם ניתנים לעדכון. ניתן לחשוב על tuples כרשימות לקריאה בלבד.
* Dictionary - מילונים בפייתון הם סוג של hash table המורכבים מצמדי מפתח-ערך. מפתח במילון יכול להיות כמעט כל סוג טפוס בפייתון אך בדרך כלל מספרים או מחרוזות. ערכים, לעומת זאת, יכולים להיות כל אובייקט. מגדירים מילון באמצעות סוגריים מסולסלות {}.

## אופרטורים

קיימים כל האופרטורים הרגילים שיש בשאר השפות מלבד האופרטורים '++', '--' שאינם חוקיים בפייתון. במקומם אפשר להשתמש ב-‘+=1’. אופרטורים לוגיים בפייתון הם המילים and, or, not. אופרטור xor מפעילים באמצעות !=. ישנם גם מספר אופרטורים חדשים:

\*\* - מייצג חזקה כאשר האופרנד משמאל הוא הבסיס והאופרנד מימין הוא המעריך. אפשר להשתמש גם באופרטור \*\*= שמבצע השמה למשתנה כאשר הוא גם הבסיס בחזקה.

7\*\*2 #49

// - חלוקה ללא שארית. אפשר להשתמש גם באופרטור //= שמבצע השמה למשתנה כאשר הוא גם המונה בחלוקה.

9//2 #4

in - אופרטור השתייכות. בודק האם איבר כלשהו נמצא במשפט, רשימה או מפתחות מילון. יש גם אופרטור הפוך ‘not in’.

"ell" in "Hi and hello" #True

is - אופרטור זהות. בודק אם המשתנים משני צידי האופרטור מצביעים על אותו אובייקט. קיים גם אופרטור ‘is not’.

## תנאים

אותו עקרון של if, if else, else שדומה לשאר השפות, אלא שבפייתון תנאים לא חייבים להיות בתוך סוגריים עגולות, לאחר הצבת התנאי רושמים נקודותיים, והתחום של התנאי הוא עד שמפסיקים את הרווח (טאב) בכל שורה. משתנה שהוגדר בתוך התנאי חי גם מחוץ לתנאי.

>>> a = 7

>>> if a < 5:

... b = 10

... else:

... b = 3

>>> b

3

## לולאות for

לולאת for משמשת כדי לעבור על כל האיברים ברשימה מסוימת. למעשה, כל הלולאות בפייתון הם לולאות foreach.

>>> list = [2, 3, "Hello",]

>>> for x in list:

... print(x)

2

3

Hello

כאשר רק רוצים לעבור בלולאה מספר כלשהו של פעמים, אפשר ליצור רשימה של מספרים טבעיים בסדר עולה באמצעות פונקציית range(num).

>>> for i in range(10):

… print(i)

ניתן לקבל בכל איטרציה איבר ברשימה וכן את האינדקס שלו על ידי שנפעיל את הפונקציה enumerate(<list>) על הרשימה, שמחזירה מילון שבו האינדקס הוא המפתח והערך הוא האיבר המתאים ברשימה.

>>> for (i, obj) in enumerate(list):

... print('%d element in list is %s' %(i, str(obj)))

0 element in list is 2

1 element in list is 3

2 element in list is Hello

שימוש נוסף ללולאות for:

>>> a = [x\*x for x in range (1, 10)]

>>> a

[1, 4, 9, 16, 25, 36, 49, 64, 81]

אפשר גם לשלב עם תנאים:

>>> even\_squares = [x\*x for x in range (1, 10) if x % 2 == 0]

>>> print(even\_squares)

[4, 16, 36, 64]

## ספריות

הכוח האמיתי של פייתון נמצא במגוון העצום של ספריות שנכתבו לה והקלות בה ניתן לייבא ולהשתמש בהן. לפייתון יש מעל ל-130,000 ספריות רשומות שלחלקן לא ניתן למצוא מקבילה בשום שפת תכנות אחרת.

כדי לייבא ספרייה או module יחיד משתמשים במילה השמורה import. כדי לגשת לפונקציות בתוך הספרייה יש לרשום אופרטור נקודה לאחר שם הספרייה ולאחר מכן את שם הפונקציה. ניתן לתת שם נוח יותר לספרייה.

Import numpy as np

np.sqrt(4)

אם לא רוצים לייבא את כל הספרייה ניתן לייבא רק פונקציות ספציפיות. במצב זה ניתן לקרוא לפונקציה ישירות.

From numpy import sqrt

Sqrt(4)

### התקנת חבילות - pip

pip היא מערכת ניהול חבילות שנכתבה בפייתון ומשמשת להתקנה וניהול של חבילות תוכנה ב-IDE. היא מחוברת לספרייה מקוונת של חבילות ציבוריות ופרטיות (בתשלום) הנקראות Python package index. היתרון המשמעותי של pip הוא הפשטות בממשק שורת הפקודה שמאפשר להתקין ולמחוק חבילה של פייתון בשורת קוד אחת בלבד:

pip install some-package-name

pip uninstall some-package-name

כברירת מחדל pip יתקין את החבילה בגרסה החדשה ביותר של פייתון שמותקן במחשב, אך ניתן גם להגדיר באיזו גרסה להשתמש:

pip${python-version-number} install some-package-name

ניתן למצוא חבילות באתר הרשמי של Python package index - <https://pypi.org/>.

### import

כדי לייבא ספריות יש להשתמש במילה השמורה import. לשם הנוחות, ניתן לתת לספרייה שם קצר יותר.

import numpy as np

ניתן לייבא פונקציה או מחלקה ספציפיים מתוך ספריה בדרך הבאה:

from module import f1, f2

כאשר רוצים להגדיר תיקייה כ-module היא חייבת להכיל קובץ ריק ששמו \_\_init\_\_.py.

## מונחים

Script – קובץ פייתון יחיד (סיומת .py) שמכיל קוד פייתון שנועד להרצה ישירות על ידי המשתמש.

Module – קובץ פייתון יחיד (סיומת .py) שמכיל קוד פייתון שמטרתו להיות מיובא בקבצים אחרים.

Package –

Library –

Framework -

## מילים שמורות

### \_\_name\_\_

כאשר מריצים תוכנית כלשהי, אזי המשתנה \_\_name\_\_ בקובץ הראשי שאותו הרצנו (כ-script) יקבל את הערך \_\_main\_\_. בשאר הקבצים שאנו מייבאים את הקוד בהם לתוכנית הראשית (כ-module) \_\_name\_\_ יקבל את שם הקובץ המיובא.

# מחרוזות ומכלים

## מחרוזות

מחרוזת היא אינה מיכל בפייתון אלא טיפוס בסיסי, כיוון שאין בשפה טיפוס מסוג תו, וההתייחסות לתו בודד היא כאל מחרוזת באורך 1. עם זאת, ניתן לגשת אל כל תו במחרוזת באמצעות גישה לפי אינדקס, כמו בטיפוסים הסדרתיים, וניתן להמיר ביניהם בקלות. ניתן לכתוב מחרוזות הן בעזרת גרשיים כפולים: "hello" והן בעזרת גרש יחיד: 'hello'. אין כל הבדל בין צורות אלו, אך מקובל להשתמש בגרש יחיד. מחרוזות הנכתבות בין שלושה גרשיים רצופים - """hello""" או '''hello''' - ניתן לפרוש על פני מספר שורות ברצף.

### גישה לתווים

כדי לגשת לחלק ממחרוזת רושמים את שם המחרוזת ואז בסוגריים מרובעות את אינדקס התו הרצוי. גישה לאינדקס לא מוגדר יוביל לשגיאה. כדי לגשת לכל התווים מאינדקס i עד אינדקס j (לא כולל) יש להוסיף בתוך הסוגריים [i:j]. כאשר i חסר הכוונה זה מאינדקס 0, וכאשר j חסר הכוונה עד האינדקס האחרון (כולל). גישה לאינדקס שלילי חוזר אחורה מהסוף להתחלה במחרוזת. שימוש בתוך הסוגריים ב-"::" ולאחר מספר טבעי יחזיר את המחרוזת המתקבלת עם קפיצות של n בין תו לתו.

str = 'Hello World!'

print (str[0]) # Prints first character of the string - “H”

print (str[2:5]) # Prints characters starting from 3rd to 5th - “llo”

print (str[2:]) # Prints string starting from 3rd character - “'llo World!”

print (str[-2]) # Prints 2nd character from the end - “d”

### Formatting

דרך מאוד נוחה להדפיס מחרוזות עם ערכים משתנים היא באמצעות formatting, שבה ניתן להגדיר מחרוזות עם דגלים {} שאותם נמלא בסוף המחרוזות בתוך השיטה format שתופעל על המחרוזת.

print('{} {} {}'.format(hello, world, 12))

hello world 12

דרך נוספת היא להדפיס מחרוזת עם דגלים מסוג %, לאחר סיום המחרוזת נרשום % ואז בסוגריים את כל ערכי הדגלים לפי סדר. את סוגי הדגלים ניתן למצוא בקישור [כאן](https://www.tutorialspoint.com/python/python_strings.htm).

print "My name is %s and weight is %d kg!" % ('Zara', 21)

My name is Zara and weight is 21 kg!

### אופרטורים

הפעלת אופרטור + בין מחרוזות מחזירה מחרוזת שהיא שרשור המחרוזת.

הפעלת אופרטור \* בין מחרוזת s למספר n מחזירה מחרוזת שהיא שרשור s בדיוק n פעמים.

s = ‘hello’

print(s + "world!") # Prints concatenated string – “helloworld!”

print(s \* 2) # Prints string two times – “hellohello“

### שיטות מרכזיות

s = ‘hello’

print(len(s)) # Returns the length of the string - 5

print(s.capitalize()) # Capitalize a string - “Hello”

print(s.upper()) # Convert a string to uppercase - "HELLO"

print(s.rjust(7)) # Right-justify a string, so that the total size of the string will be 7. padding with spaces. - “ hello”

print(s.center(7)) # Center a string, padding with spaces. - “ hello ”

print(s.replace('l', '(ell)')) # Replace all instances of one substring with another - “he(ell)(ell)o”

print(' world '.strip()) # Strip leading and trailing whitespace - “world”

print(s.count(‘l)) # Counts how many times substring occurs in string – 2

print(s.find(‘world’)) # Determine if str occurs in string. returns index if found and -1 otherwise.

## רשימות - Lists

רשימה בפייתון זהו רצף של עצמים, לא חייב שיהיו מאותו סוג. מצהירים על רשימה באמצעות סוגריים מרובעות שבתוכן האיברים מופרדים בפסיקים.

>>> a = [4, 3.0, 'Hello']

אפשר גם להגדיר רשימה שהיא איבר פנימי בתוך רשימה אחרת.

>>>a = [4, [5, 4, [5.2, 9.3]]]

>>>a[1][2][0]

5.2

### גישה לאיברים

הגישה לאיבר ברשימה היא בדיוק כמו במערך ב-Java, כלומר רושמים את שם הרשימה ואז בסוגריים מרובעות את אינדקס האיבר הרצוי. גישה לאינדקס לא מוגדר יוביל לשגיאה. כדי לגשת לכל האיברים מאינדקס i עד אינדקס j (לא כולל) יש להוסיף בתוך הסוגריים [i:j]. כאשר i חסר הכוונה זה מאינדקס 0, וכאשר j חסר הכוונה עד האינדקס האחרון (כולל). גישה לאינדקס שלילי חוזר אחורה מהסוף להתחלה ברשימה. שימוש בתוך הסוגריים ב-"::" ולאחריו מספר טבעי n, יחזיר את הרשימה המתקבלת עם קפיצות של n בין איבר לאיבר.

>>> a[1]

3.0

>>> a[1:2]

3.0

>>> a[-1]

'Hello'

>>> a[::2]

[4, 'Hello']

### אופרטורים

הפעלת אופרטור + בין רשימות מחזירה רשימה שהיא שרשור של הרשימות.

הפעלת אופרטור \* בין רשימה l למספר n מחזירה רשימה שהיא שרשור l בדיוק n פעמים.

print([123, 'john'] + [1,2]) # Prints concatenated lists – [123, 'john', 1, 2]

print([123, 'john'] \* 2) # Prints list two times - [123, 'john', 123, 'john']

### שיטות מרכזיות

יש המון שיטות שימושיות שניתן להפעיל על רשימות.

ls.append('bar')  # Add a new element to the end of the list

ls.pop() # Remove and return the last element of the list

## Tuples

Tuples הם אוסף מסודר של אובייקטים. הם זהים בדיוק לרשימות אלא ש-tuple הוא מוגן, כלומר לאחר שיוצרים אותו לא ניתן לשנות את האיברים. ניסיון לשנות איבר בתוך tuple יוביל לשגיאת TypeError. הבדל נוסף הוא שעל tuple מצהירים באמצעות סוגריים עגולות ולא מרובעות. הגישה לאיבר ב-tuple זהה בדיוק לגישה לאיבר במערך. כדי להגדיר tuple עם איבר אחד יש לרשום בתוך הסוגריים את הערך ולאחריו פסיק.

>>> a = (4, 3.0, "Hello")

>>> a[1:2]

3.0

>>>t = (1,2,3)

>>>(a, b, c) = t

>>>a

1

Tuple, להבדיל מרשימה גם יכול להיות מפתח במילון וכן איבר ב-set.

d = {(x, x + 1): x for x in range(10)}  # Create a dictionary with tuple keys

## מילונים - Dictionaries

מילון הוא אוסף של איברים מסודרים בצורה של מפתח-ערך. מגדירים מילון באמצעות סוגריים מסולסלות {}. כל איבר מופרד בפסיק, ובין כל מפתח ערך יש נקודותיים ':'. המפתחות במילון חייבים להיות ייחודיים בעוד שהערכים אינם חייבים להיות ייחודיים. הערכים של מילון יכולים להיות מכל סוג שהוא, אך המפתחות חייבים להיות מסוג נתונים בלתי משתנה (immutable) כגון מחרוזות, מספרים או Tuples.

### גישה לאיברים

כדי לגשת לערך במילון ניתן לרשום בסוגריים מרובעות את המפתח שלו.

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

dict[‘Name’] #Zara

באמצעות אופרטור in ניתן לבדוק האם קיים מפתח כלשהו במילון.

‘Class’ in dict #True

אם ניגשים למפתח שלא קיים במילון תתקבל שגיאה “not a key”. ניתן להשתמש בשיטה get על המילון המקבלת מפתח וערך ברירת מחדל, כך שבמידה והערך לא נמצא יוחזר ערך ברירת מחדל.

print(dict.get('monkey', 'N/A')) #N/A

### שיטות מרכזיות

dict.keys() # Prints all the keys

dict.values() # Prints all the values

ניתן לעדכן ערך על ידי גישה למפתח שלו ולאחר מכן השמה. בשיטה זו ניתן גם להכניס רשומה חדשה למילון. מוחקים ערך באמצעות אופרטור del.

dict['Age'] = 8; # update existing entry

dict['School'] = "DPS School"; # Add new entry

del dict[‘Age’] # delete existing entry

### איטרציה

ניתן לעבור על כל איברי המילון באמצעות שיטת items המחזירה רשימה של זוגות:

>>> d = {'person': 2, 'cat': 4, 'spider': 8}

>>> for animal, legs in d.items():

…   print('A {} has {} legs'.format(animal, legs))

A person has 2 legs

A cat has 4 legs

A spider has 8 legs

## Set

זהו אוסף לא מסודר של איברים **שונים**. מגדירים set באמצעות סוגריים מסולסלות {}. כל איבר מופרד בפסיק. האיברים ב-set חייבים להיות ייחודיים. הערכים יכולים להיות מכל סוג שהוא. איטרציה על איברים ה-set היא בדיוק כמו ברשימה, אך בגלל שהאיברים לא מסודרים אי אפשר לדעת מה יהיה סדר האיברים באיטרציה.

animals = {'cat', 'dog'}

### שיטות מרכזיות

animals.add('fish')     # Add an element to a set

animals.add('cat') # Adding an element that is already in the set does nothing

animals.remove('cat') # Remove an element from a set

## פעולות על מחרוזות ומכלים

* כדי לקבל את אורך המיכל או המחרוזת משתמשים בפונקציה len.
* הפעלת אופרטור + תשרשר בין המכלים או המחרוזות.
* ניתן להשתמש באופרטור in, המחזיר ערך bool, כדי לבדוק אם איבר נמצא במיכל.

# פונקציות

## הצהרה

פונקציות מתחילים עם מילת המפתח def ואחריהם שם הפונקציה וסוגריים. בתוך הסוגריים יש למקם את כל פרמטרים לפונקציה. הבלוק של הפונקציה מתחיל בנקודותיים שבאים לאחר הסוגריים. הבלוק של הפונקציה הם כל השורות המוזחות שבאות אחרי ההצהרה. המשפט הראשון של פונקציה יכול להיות משפט אופציונלי - מחרוזת תיעוד לפונקציה או docstring. באמצעות return אפשר להחזיר ערך כלשהו, או מספר ערכים (tuple), לפונקציה הקוראת. פונקציות בפייתון תמיד מחזירות ערך, במידה וה-return ריק יוחזר None.

def func\_name(parameters):

"function\_docstring"

function\_suite

return [expression]

## פרמטרים וארגומנטים

פרמטרים אלו הערכים שהפונקציה מקבלת, ארגומנטים הם הערכים שנשלחים לפונקציה. כל הארגומנטים בפייתון מועברים By Reference, כלומר כל שינוי בארגומנט בתוך הפונקציה ישפיע גם בחזרה על הפונקציה הקוראת.

כששולחים ארגומנטים לפונקציה, הפונקציה תקבל אותם לפי סדר הפרמטרים שכתובים בחתימתה, כלומר הארגומנט הראשון יכנס לפרמטר הראשון, וכן הלאה. אך ניתן לשלוח את הארגומנטים לא לפי סדר הופעתן בחתימת הפונקציה, אם נגדיר לאיזה פרמטר אנחנו שולחים איזה ארגומנט בקריאה לפונקציה.

def func(A ,B, C):

print(f 'A is {A}, B is {B}, C is {C}')

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

### פרמטר ברירת מחדל

ניתן להגדיר בפונקציה ערכי ברירת מחדל לפרמטר כך שיהיה ניתן לקרוא לפונקציה גם בלי להכניס את הפרמטרים שהם ברירת מחדל. לאחר שמגדירים פרמטר עם ברירת מחדל לא ניתן להכניס פרמטרים ללא ברירת מחדל לאחריו.

def printinfo(name, age = 35 ):

print "Name: ", name

print "Age ", age

כאשר ה-interpreter יגיע לשורת הצהרת פונקציה עם ערך ברירת מחדל, הוא מגדיר מצביע למשתנה בטיפוס הברירת מחדל. לכן, במידה וטיפוס זה הוא מסוג mutable והפונקציה תשנה ערך זה, בקריאה הבאה לפונקציה משתנה ברירת המחדל ישמר עם הערך מהקריאה הקודמת מה שעלול להוביל להתנהגות לא רצויה. המסקנה מכך היא שנשתדל שטיפוס ערך הברירת מחדל יהיה מסוג immutable. אם בתוך הפונקציה נרצה לשנות ערך זה, נעשה זאת באמצעות תנאי האם המשתנה מסוג immutable, אם כן, נשנה אותו ל-mutable.

def stam\_list (my\_list=None):

if my\_list==None:

my\_list=[]

my\_list.append('###')

return my\_list

### הוספת כוכבית

כאשר מצהירים על פרמטר שלפניו יש \*, אזי הארגומנט הראשון שיכנס לפרמטר זה, וכל הארגומנטים שיבואו אחריו שאין להם פרמטר אחר להיכנס אליהם, יכנסו לתוך tuple ששמו הוא שם הפרמטר. פרמטר מסוג זה הוא מאוד שימושי כאשר לא רוצים להגביל את מספר הפרמטרים.

def avg(\*nums):

sum = 0

for num in nums:

sum += num

return sum/len(nums)

שליחה של ארגומנט עם כוכבית תבצע את הפעולה ההפוכה. כלומר, תפרק את הארגומנט לרכיבים ותשלח כל אחד מהם בנפרד כפרמטר לפונקציה. שימושי רק כאשר הארגומנט הוא אוסף כלשהו.

def sum\_of\_three(num1,num2,num3):

return (num1 + num2 + num3)

tup\_int = (1,2,3)

sum\_of\_three(\*tup\_int) #6

### הוספת שתי כוכביות

פרמטר שלפניו יש שתי כוכביות עוטף את הארגומנטים שמתקבלים והופך אותם למילון. בשליחת הארגומנטים לפרמטר עם שני כוכביות צריך לציין לכל ארגומנט מהו המפתח שלו.

def f(\*\*kwargs):

for key, val in kwargs.items():

print(key, '->', val)

f(e=1, r=2,j=3)

# e -> 1

# r -> 2

# j -> 3

שליחת ארגומנט עם שתי כוכביות לפונקציה שימושית רק כאשר הארגומנט הוא מילון. במקרה זה כל הערכים במילון יפורקו וישלחו כפרמטרים נפרדים. שליחת מילון כארגומנט לפונקציה עם כוכבית אחת לפני, תפרק את המפתחות ותשלח אותם כפרמטרים נפרדים.

## פונקציות אנונימיות - Lambda Expressions

פונקציות אלה נקראות אנונימיות מכיוון שהן אינן מוכרזות באופן הסטנדרטי באמצעות המילה השמורה def. אלו פונקציות המוגדרות באמצעות שורה אחת. ההצהרה עליהן מתחילה במילה השמורה lambda לאחר מכן רושמים את הפרמטרים שהפונקציה מקבלת עם פסיק בין כל פרמטר. ניתן להכניס איזה מספר של פרמטרים שנרצה. לאחר הפרמטרים רושמים נקודותיים ':', ואז את הביטוי שהוא גם ערך החזרה מהפונקציה.

lambda [arg1 ,arg2,.....argn]: expression

דוגמאות לפונקציות אנונימיות:

>>> f = lambda x: x\*x

>>> f(5)

25

>>> def o(f,g): # composite functions

... return lambda x: f(g(x))

>>> b = o(lambda x: x\*x, lambda x: x+1)

>>> b(3)

16

# מונחה עצמים

פייתון הייתה שפה מונחה עצמים מאז שהיא קיימת. בגלל זה, יצירה ושימוש במחלקות ואובייקטים הם פשוטים לחלוטין.

## מחלקה

כדי ליצור מחלקה משתמשים במילה השמורה class, לאחריה שם המחלקה ואז בתוך בלוק המחלקה נגדיר את משתני המחלקה, בנאים ושיטות. לדוגמה:

class Employee:

'Common base class for all employees'

empCount = 0

def \_\_init\_\_(self, name, salary):

self.name = name

self.salary = salary

Employee.empCount += 1

def displayCount(self):

print "Total Employee %d" % Employee.empCount

def displayEmployee(self):

print "Name : ", self.name, ", Salary: ", self.salary

## משתני מחלקה

זהו משתנה המשותף לכל המופעים של מחלקה. משתני מחלקה מוגדרים בתוך המחלקה אך מחוץ לכל אחת מהשיטות של המחלקה. בדוגמה לעיל, empCount הוא משתנה מחלקה. ניתן לגשת אליהם דרך אופרטור נקודה לאחר שם המחלקה.

Employee.empCount

## בנאים \_\_init\_\_

השיטה הראשונה בכל מחלקה היא שיטה מיוחדת המגדירה כיצד לבנות אובייקט חדש מסוג המחלקה. הפרמטרים לשיטה זו אלו הם השדות של האובייקט. בתוך השיטה \_\_iniy\_\_ יש לאתחל שדות אלו.

def \_\_init\_\_(self, name, salary):

self.name = name

self.salary = salary

Employee.empCount += 1

כדי ליצור מופעים של מחלקה, קוראים למחלקה באמצעות שם המחלקה ומעבירים את כל הארגומנטים ששיטת \_\_init\_\_ שלה מקבלת.

emp = Employee("Zara", 2000)

## שיטות

מכריזים על שיטות מחלקה כמו פונקציות רגילות, למעט זה שהארגומנט הראשון לכל שיטה הוא self, המייצג מצביע לאובייקט שעליו הפעילו את השיטה (כמו this ב-Java). כאשר קוראים לשיטה דרך שם האובייקט, Python מוסיף את הארגומנט self כארגומנט ואין צורך לציין זאת כשקוראים לשיטות.

emp1.displayEmployee() # return “Name : Zara ,Salary: 2000”

### שיטות מובנות

כל מחלקה של Python מכילה שיטות מובנות וניתן לגשת אליהן באמצעות אופרטור נקודה כמו לכל שיטה ותכונה אחרת:

\_\_dict\_\_ − מילון המכיל את מרחב השמות של המחלקה.

\_\_doc\_\_ − מחרוזת תיעוד המחלקה או none, אם לא מוגדר.

\_\_name\_\_ − שם המחלקה.

\_\_module\_\_ − שם המודול שבו מוגדרת המחלקה. "\_\_main\_\_" במצב אינטראקטיבי.

\_\_bases\_\_ − tuple אולי ריק המכיל את מחלקות הבסיס, לפי סדר הופעתם ברשימת מחלקות הבסיס.

## גישה לשדות אובייקט

ניתן לגשת לתכונות האובייקט באמצעות אופרטור נקודה כמו בשיטות, אולם מומלץ להשתמש בשיטות מובנות הבאות:

The getattr(obj, name[, default]) − to access the attribute of object.

The hasattr(obj,name) − to check if an attribute exists or not.

The setattr(obj,name,value) − to set an attribute. If attribute does not exist, then it would be created.

The delattr(obj, name) − to delete an attribute.

# קבצים

## open

כדי להחזיר אובייקט המייצג קובץ משתמשים בפונקציה open. הפונקציה מקבלת:

* path אל הקובץ אותו רוצים לפתוח.
* הרשאת גישה: "r" לקריאה. "w" לכתיבה. "a" להוספה. "r+" הן לקריאה והן לכתיבה.

file = open('file\_path', 'w')

### with

נהוג בפתיחת קובץ להשתמש בהצהרת with כדי לטפל בחריגות. חריגה במהלך פעולה על קובץ עלולה למנוע מהקובץ להיסגר כראוי, מה שעלול להכניס מספר באגים בקוד, כלומר שינויים רבים בקבצים לא נכנסים לתוקף עד שהקובץ נסגר כראוי. כשמשתמשים ב-with אין צורך לקרוא ל-file.close() אלא הצהרת with עצמה מבטיחה הקצאה ושחרור של משאבים כאשר הקוד המשתמש במשאב מבוצע לחלוטין. הצהרה זו בעצם שקולה לשימוש ב-try לפני פקודה על קובץ ו-finally כדי לסגור את הקובץ, אלא שהצהרת with עוזרת להפוך את הקוד ליותר נקי וקריא.

with open('file\_path', 'w') as file:

file.write('hello world !')

## פעולות על קובץ

### close

מפסיקה את כל המשאבים הנמצאים בשימוש ומשחררת את המערכת של תוכנית מסוימת זו.

# NumPy

## הגדרה

NumPy היא ספרייה בפייתון המוסיפה תמיכה במערכים ומטריצות רב-ממדיות, יחד עם אוסף גדול של פונקציות מתמטיות ברמה גבוהה להפעלת מערכים אלה. NumPy היא תוכנת קוד פתוח ויש לה תורמים רבים.

לאחר התקנת NumPy, בכל תוכנית צריך לייבא אותו על ידי import. NumPy מיובא בדרך כלל תחת הכינוי np.

import numpy as np

## מערך ב-NumPy

בפייתון יש לנו רשימות המשרתות את מטרת המערכים, אך הן איטיות לעיבוד. NumPy שואפת לספק אובייקט מערך המהיר פי 50 מהרשימה המובנית של פייתון. מערכי NumPy מאוחסנים במקום אחד רציף בזיכרון בניגוד לרשימות, כך שתהליכים יכולים לגשת אליהם ולנהל אותם בצורה יעילה מאוד. כמו כן, הוא מותאם לעבודה עם ארכיטקטורות המעבד העדכניות ביותר. מערכים משמשים לעתים קרובות במדעי הנתונים, שם המהירות והמשאבים חשובים מאוד.

אובייקט המערך ב-NumPy נקרא ndarray, מפני שניתן להגדיר לכל מערך n ממדים (מערכים מקוננים). מערך ב-NumPy מספק שיטות תומכות רבות שהופכות את העבודה עם ndarray לקלה מאוד. הגישה לאיברים במערך היא כמו שלמדנו בגישה לאיבר ברשימה.

>>> arr0d = np.array(42)

>>> arr1d = np.array([1, 2, 3, 4, 5])

>>> arr2d = np.array([[1, 2, 3], [4, 5, 6]])

>>> arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

>>> arr2d.shape()

(2, 3)

>>> arr3d.ndim()

2

>>> arr3d.dtype()

dtype('int32')

>>> arr2d.size()

6

## פעולות על ndarray

ברשימות רגילות, פעולות אריתמטיות, כמו חיבור וכפל, רק מגדילות את הרשימה. לעומת זאת פעולות אריתמטיות על ndarray מפעיל את הפעולה לכל איבר בנפרד בהתאמה.

>>> np.array([2, 4, 5]) + np.array([7, 3, 2])

array([9, 7, 7])

>>> np.array([2, 4, 5]) \* np.array([7, 3, 2])

array([14, 12, 10])

* np.dot(<np.array>, <np.array>) - מקבלת שני ndarray ומחזירה את המכפלה הפנימית שלהם. עבור שני מערכים חד ממדיים פעולה זו שקולה למכפלת וקטורים, ועבור שני מערכים דו-ממדיים פעולה זו שקולה להכפלת מטריצות.
* np.mean(<np.array>) – מקבלת וקטור ומחזירה את הממוצע של כל הרכיבים בו.
* np.linalg.inv(<np.array>) - מקבלת מערך דו-ממדי (מטריצה) ומחזירה את המטריצה ההופכית שלה, כך שאם יכפילו בין שניהם תתקבל מטריצת היחידה.

# בדיקות

כל מערכת תוכנה רצינית כוללת אוסף מקיף של בדיקות אוטומטיות. בתהליך הפיתוח, בכל פעם שמבצעים שינוי במערכת, גם אם זה שינוי קטן, מריצים את כל הבדיקות מחדש, כדי לוודא שלא קלקלנו שום דבר. יש שלוש דרכים נפוצות לבצע בדיקות אוטומטיות בפייתון: Doctest, Unittes, Pytest.

## Doctest

אלו הן בדיקות שבדרך כלל נמצאות בתוך התיעוד של פונקציה או קובץ. יתרון מרכזי של כתיבת הבדיקות בתיעוד הוא שמאפשר להציג למתכנתים אחרים את השימוש הפונקציה.

שימוש ב-Doctest מאוד דומה להרצת פקודות בטרמינל של פייתון. כדי לבצע בדיקה יש לרשום ">>>" ולאחר מכן את הפקודה שאנו רוצים לבדוק. בשורה לאחר מכן נרשום את התוצאה שצריכה להיות לפקודה. הבדיקה נעשית באמצעות השוואת מחרוזות בין התוצאה של הפקודה לתוצאה שרשמנו. בדיקות אלו צריכות להיות פשוטות ומדויקות, ולכן עבור בדיקות מסובכות נרצה להשתמש בסוג אחר של בדיקות.

בקובץ של הבדיקות יש להוסיף פקודה, שכאשר מריצים קובץ זה כ-main, אזי נדפיס את התוצאה של הפונקציה testmode מהמודול doctest. הדפסה זו תציין איזה בדיקות עברו ואיזה לא. עבור בדיקות שלא עברו יצוין מה הבדיקה ציפתה לקבל ומה קיבלה בפועל. אם גם עבור בדיקות שעברו נרצה לראות מה הפונקציה ניסתה לבחון, אפשר להוסיף- '-v' לשורת ה'קימפול' של הקובץ והוא יציג את התוצאה.

### דוגמה

def factorial(n):

"""Return the factorial of n, an exact integer >= 0.

>>> [factorial(n) for n in range(6)]

[1, 1, 2, 6, 24, 120]

>>> factorial(30)

265252859812191058636308480000000

>>> factorial(-1)

Traceback (most recent call last):

...

ValueError: n must be >= 0

Factorials of floats are OK, but the float must be an exact integer:

>>> factorial(30.1)

Traceback (most recent call last):

...

ValueError: n must be exact integer

>>> factorial(30.0)

265252859812191058636308480000000

It must also not be ridiculously large:

>>> factorial(1e100)

Traceback (most recent call last):

...

OverflowError: n too large

"""

import math

if not n >= 0:

raise ValueError("n must be >= 0")

if math.floor(n) != n:

raise ValueError("n must be exact integer")

if n+1 == n: # catch a value like 1e300

raise OverflowError("n too large")

result = 1

factor = 2

while factor <= n:

result \*= factor

factor += 1

return result

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

import doctest

print(doctest.testmod())

## Unittest

אלו הן בדיקות שמטרתן היא בדרך כלל לבדוק מחלקה או כל הפונקציות בקובץ פייתון כלשהו. כדי להריץ בדיקות unittest יש ליצור קובץ פייתון נפרד, שכמוסכמה השם שלו הוא כשם הקובץ אותו הוא בא לבחון בתוספת המילה test\_ לפני, למשל test\_gymatria.py.

כל קובץ unittest מתחיל ביצירת מחלקה חדשה שיורשת מהמחלקה TestCase. כל מתודה שבה אנחנו מבצעים את הבדיקות חייבת להתחיל במילה test\_ בתחילת שמה (אחרת היא לא תוכר כמתודת טסט), וחייבת להכיל self כפרמטר. הבדיקות במתודה אמורים להיות בצורה של טסט assert כלשהו. יש הרבה סוגים של asserts שאפשר להשתמש בהם, ואפשר למצוא את רובם [בקישור כאן](https://docs.python.org/3/library/unittest.html).

יש שני דרכים להריץ את הבדיקות:

1. ליצור פונקציית main שתפעיל את הפונקציה main של המודול unitest ואז לקמפל את קובץ הבדיקה משורת הפקודה.
2. לקמפל ולהריץ את הקובץ באמצעות הפקודה "python –m unittest test\_filename.py".

אפשרות מעניינת היא שאם נרצה לשמור משתנים שיאותחלו בתחילת כל טסט, נוכל להשתמש במתודה setUp() שמתבצעת לפני כל טסט והנתונים שלה נשמרים גם בטסט. מתודה שימושית בהקשר זה היא גם teardown() שמתבצעת אחרי כל טסט. שימושית בעיקר כשמשתמשים בקבצים שצריך לסגור לאחר כל טסט.

### דוגמה

import unittest

from gymatria import Gymatria

class TestGymatria(unittest.TestCase):

def setUp(self):

        self.aba = Gymatria('אבא')

        self.ab = Gymatria('אב')

        self.efes = Gymatria('')

        self.aima = Gymatria('אמא')

        self.df = Gymatria('dfא')

    def tearDown(self):

        pass

def test\_add(self):

self.assertEqual(self.aba + self.aima , 46)

        self.assertEqual(self.aba + Gymatria('fi') , 4)

        self.assertEqual(self.aba + self.aba , 8)

        self.assertEqual(self.aba + self.efes , 4)

        self.assertEqual( self.efes + self.efes , 0)

        self.assertEqual( self.efes + 10 , 10)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

## Pytest

זהו מודול חיצוני שנוצר כתחליף ל unittest כדי לאפשר תחביר קצר ופשוט יותר, וכן להוסיף עוד אפשרויות. כיום, pytest היא מערכת הבדיקות הנפוצה ביותר. כיוון שזה מודול חיצוני, יש להתקין אותו כמו כל מודול:

pip install pytest

בדומה ל-unittest יש ליצור קובץ נפרד שמתחיל ב-test\_, וכן כל פונקציה שמריצה בדיקות חייבת להתחיל ב-test\_. אמנם בשונה מ-unittest הקובץ לא חייב להכיל מחלקות וכן לא צריך להשתמש בהמון סוגים של assert אלא רק סוג אחד שמקבל ביטוי בוליאני.

pytest יודע לזהות בדיקות של doctest ושל unittest ולהריץ גם אותן, כך שאפשר לראות בו מערכת אחת שכוללת את כל סוגי הבדיקות. כדי להריץ את כל הבדיקות בכל קבצי ה-pytest ו-unittest שנמצאים בתיקייה נוכחית (קבצים אלו מתחילים ב-test\_), יש להריץ את הפקודה pytest בשורת הפקודה. כדי להריץ בנוסף את בדיקות ה-doctest יש להריץ:

pytest --doctest-modules

### דוגמה

aba = Gymatria('אבא')

aima = Gymatria('אמא')

def test\_add(): # test functions must start with 'test\_'

assert aba + aima == 46

assert aba + Gymatria('fi') == 4

def test\_raise():

with pytest.raises(ValueError):

Gymatria.get\_value()

## שילוב בדיקות אוטומטיות בתהליך הפיתוח – GitHub Actions

ביצוע בדיקות אוטומטיות אחרי כל שינוי הוא מאד חשוב, ולכן ישנם כלים המאפשרים לנו לבצע זאת באופן אוטומטי.