



תכנות מונחה עצמים

נושא 8 JUNIT

אליזבת איצקוביץ, elizabeti@ariel.ac.il

מחלקה למדעי המחשב

אוניברסיטת אריאל בשומרון

נושא 8 JUNIT

JUnit הוא כלי פתוח וחופשי לבדיקות יחידה בשפת התכנה java.

הכלי פותח במשותף על ידי קנט בר ואריך גמא כחלק מעבודתם בנושא פיתוח מונחה בדיקות

ופיתוח תכנה זריז. JUnit הוא פרויקט קוד פתוח המתארח ב- Github

טסט JUnit הוא שיטה הנכללת במחלקה המשמשת רק לבדיקה. זה נקרא מחלקת טסט.

כדי להגדיר ששיטה מסוימת היא שיטת בדיקה, הערה עליה בהערת @Test.

בגדול – ה- JUnit נותן לנו את היכולת להריץ את הבדיקות (הקוד) שלנו בצורה

אוטומטית (כן, למעשה זה האוטומציה של האוטומציה).

להלן ה- Dependencies אותם יש לצרף לפרויקט בשביל לעבוד עם JUnit5.

junit-jupiter-api חושף בפנינו את ה-API של JUnit.

נושא 8 JUNIT

אנוטאציות:

פונקציית ביאור	הסבר
@Test	מסמל שהפונקציה היא פונקציית בדיקה
@BeforeEach	פונקציה שתרוץ לפני כל בדיקה. פונקציה זו יכולה להכין את סביבת הבדיקה (למשל: לקרוא את נתוני הקלט, לאתחל את המחלקה)
@AfterEach	פונקציה שתרוץ לאחר כל בדיקה. פונקציה זו יכולה לנקות את סביבת הבדיקה (למשל: למחוק נתונים זמניים, לשחזר ברירת מחדל). בנוסף יכולה לחסוך בזיכרון על ידי ניכוי מבני נתונים יקרים.
@BeforeAll	פונקציה זו מתבצעת פעם אחת, לפני תחילת כל הבדיקות. יכול לשמש לביצוע פעולות שדורשות זמן כגון חיבור למסדי נתונים. צריכה להיות מוגדרת כסטטית.
@AfterAll	פונקציה זו מתבצעת פעם אחת, לאחר שכל הבדיקות הסתיימו. יכולה לשמש לביצוע פעולות ניקוי, למשל התנתקות מבסיס נתונים. צריכה להיות מוגדרת כסטטית.
@Disabled	מתעלם מפונקציית הבדיקה. שימושי כאשר הקוד הבסיסי השתנה ומקרי הבדיקה עדיין לא אומצו או שזמן בדיקה זה יותר מידי גדול מלהיכלל

נושא 8 JUNIT

בדיקה אם הפונקציה עברה את הבדיקה

```
assertNotNull(Object object, String message);
assertNotSame(Object expected, Object actual);
assertNotSame(Object expected, Object actual, String message);
assertNull(Object object);
assertNull(Object object, String message);
assertSame(Object expected, Object actual);
assertSame(Object expected, Object actual, String message);
assertTrue(boolean condition);
fail(String message);
```

נושא 8 JUNIT

בדיקה אם הפונקציה עברה את הבדיקה

```
assertEquals(Type expected, Type actual);
assertEquals(Type expected, Type actual, Type delta);
assertEquals(Type expected, Type actual, String message);
assertEquals(Type expected, Type actual, Type delta, String message);
assertFalse(boolean condition);
assertFalse(boolean condition, String message);
assertNotNull(Object object);
assertTrue(boolean condition, String message);
fail();
```



את השאלות יש לשאול בפורום הקורס
(מיצוץ אישי - מודול)



PYTHON

שעור 1

אליזבת איצקוביץ, elizabeti@ariel.ac.il

מחלקה למדעי המחשב

אוניברסיטת אריאל בשומרון

Python בסיס השפה

Python היא שפת תכנות פופולרית. היא נוצרה על ידי גווידו ואן רוסום (Guido van Rossum), ומשוחררת בשנת 1991.

- ❖ Python היא שפת תכנות שיכולה לפעול במספר פלטפורמות כמו Windows, macOS, Linux.
- ❖ ל-Python יש תחביר פשוט הדומה לשפה האנגלית.
- ❖ התחביר של Python מאפשר למפתחים לכתוב תוכניות עם פחות שורות מאשר בשפות תכנות אחרות.

מה Python יכול לעשות?

- ❖ Python יכולה להתחבר למערכות מסדי נתונים. היא יכולה גם לקרוא ולשנות קבצים.
- ❖ ניתן להשתמש ב-Python לטיפול Big Data ולביצוע מתמטיקה מורכבת.
- ❖ ניתן להשתמש ב-Python בשרת כדי ליצור יישומי אינטרנט.

Python היא חנימית עם קוד פתוח.

Python בסיס השפה

טוב לדעת:

- ❖ גרסה העיקרית האחרונה של Python היא 3, בה נשתמש. עם זאת, Python 2 , למרות שאינה מעודכנת בשום דבר מלבד עדכוני האבטחה, עדיין פופולרי למדי.
- ❖ אפשר לכתוב Python בסביבת פיתוח משולבת, כגון Thonny, Pycharm, Netbeans או Eclipse, אשר שימושיים במיוחד בעת ניהול אוספים גדולים יותר של קבצי Python.

אנו נשתמש ב- Pycharm

Python בסיס השפה

תחביר Python בהשוואה לשפות תכנות אחרות:

- ❖ Python תוכנן לקריאות, ויש לה קווי דמיון לשפה האנגלית עם השפעה של מתמטיקה.
- ❖ Python משתמש בשורות חדשות להשלמת פקודה, בניגוד לשפות תכנות אחרות המשתמשות לרוב בנקודות-פס או בסוגריים.
- ❖ Python מסתמך על כניסה, תוך שימוש רווחים, כדי להגדיר היקף; כגון היקף לולאות, פונקציות ומחלקות. שפות תכנות אחרות משתמשות לעתים קרובות בסוגריים מסולסלים למטרה זו.

Python בסיס השפה

Python היא שפת תכנות שיכולה לפעול במספר פלטפורמות כמו Windows, macOS, Linux ואף היא חינמית עם קוד פתוח. אנו עובדים בסביבת **pycharm**.

Python Keywords מילות מפתח של פייתון.

Python, כמו Java מבחינה בין אותיות גדולות לקטנות, (In Python, keywords are case sensitive)

כל מילות המפתח למעט True, False, None כתובות באותיות קטנות.

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

רשימת כל מילות המפתח מוצגת להלן:

True ו- False ב Python

זהה ל 1 ו- 0:

Python בסיס השפה

רווחים ב-Python : רווחים מאוד חשובים ב-Python

דוגמה תקינה

```
if 5 > 2:  
    print("Five is greater than two!")
```

דוגמה שגויה:

```
if 5 > 2:  
print("Five is greater than two!")
```

משתנים ב-Python

משתנים הם מיכלים לאחסון ערכי נתונים.

ל-Python אין פקודה להכריז על משתנה. משתנה נוצר ברגע בו תחילה הוקצו לו ערך.

דוגמה

```
x = 5  
y = "John"  
print(x)  
print(y)
```

Python בסיס השפה

comments – הערות

הערות מתחילות ב- #, ו- Python תתעלם מהן. אפשר להוסיף הערה מרובת שורות: """

דוגמה

```
#This is a comment
print("Hello, World!")

print("Hello, World!") #This is a comment
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

Python בסיס השפה

ניתן להכריז על משתני מחרוזות באמצעות מרכאות בודדות או כפולות:

```
x = "John" # is the same as  
x = 'John'
```

שמות של משתנים:

משתנה יכול להיות שם קצר (כמו x ו-y) או שם תיאורי יותר (age, firstName, last_name).

כללים למשתני Python:

- ❖ שם משתנה חייב להתחיל באות או במקף תחתון.
- ❖ שם משתנה אינו יכול להתחיל במספר.
- ❖ שם משתנה יכול להכיל רק תווים אלפא-נומריים ומקף תחתון (0-9, A-Z, _).
- ❖ שמות משתנים הם case sensitive (age, Age, AGE הם שלושה משתנים שונים).

Python בסיס השפה

אין צורך להכריז על משתנים עם סוג מסוים ואף יכולים לשנות סוג לאחר הגדרתם:

```
x = 4          # x is of type int
x = "Sally"    # x is now of type str
print(x)
```

Casting - המרה

אם ברצוננו לציין את סוג הנתונים של משתנה, ניתן לעשות זאת באמצעות ההמרה

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

אפשר לקבל את סוג הנתונים של משתנה עם הפונקציה `type()`:

```
x = 5
y = "John"
print(type(x))
print(type(y))
```


Python בסיס השפה

Python מאפשר להקצות ערכים למספר משתנים בשורה אחת

```
x, y, z = "Orange", "Banana", "Cherry"  
print("x = ", x, "y = ", y, "z = ", z)
```

```
x = y = z = "Orange"  
print("x = ", x, "y = ", y, "z = ", z)
```

Output - פלט, כדי לשלב גם טקסט וגם משתנה, Python משתמש בתו +:

```
x , y = "I love ", "Python and "  
print(x + y + "Shokolad ")
```

כאשר מנסים לשלב מחרוזת ומספר, Python תיתן שגיאה:

```
x = 5 , y = "I love Java and "Python "  
print(x + y)
```

TypeError: unsupported operand type(s) for +: 'int' and 'str'

תיקון:

```
print(str(x) + y)
```

Python בסיס השפה

Global Variables משתנים גלובליים

משתנים שנוצרים מחוץ לפונקציה (כמו בכל הדוגמאות לעיל) ידועים כמשתנים גלובליים. במשתנים גלובליים ניתן להשתמש הן בתוך הפונקציות והן מבחוץ.

```
x = "world"
def myfunc():
    print("hello " + x)
myfunc()
Output: hello world
```

כאשר נוצר משתנה עם אותו שם בתוך פונקציה, משתנה זה יהיה מקומי, וניתן להשתמש בו רק בתוך הפונקציה. המשתנה הגלובלי עם אותו שם יישאר כמו שהיה, גלובלי ועם הערך המקורי.

```
x = "world "
def myfunc():
    x = "fantastic"
    print("Python is " + x)
myfunc()
print("Python is " + x)
Output: Python is fantastic
```

Python בסיס השפה

Python - Basic Operators – פעולות בסיסיות

פעולות אריתמטיות

פעולה	תיאור	דוגמה
+	חיבור	$3+2=5$
-	חיסור	$3-2=1$
*	כפל	$2*3=6$
/	חילוק	$3/2=1.5$
%	מודולו	$3\%2=1$
**	חזקה	$2**3=8$
//	חילוק, התוצאה מעוגלת כלפי מטה	$9//2=4$ $-11//3=-4$

פעולות השמה

פעולה	דוגמה
$+=$	$a+=b = a+b$
$-=$	$a-=b = a-b$
$*=$	$a*=b = a*b$
$/=$	$a/=b = a/b$
$\% =$	$a\%=b = a\%b$
$**=$	$a**=b = a**b$
$//=$	$a//=b = a//b$

פעולות השוואה : $==$, $!=$ אותו דבר כמו $<$, $<=$, $>$, $>=$, $<$.

Python בסיס השפה

Python Logical Operators – פעולות לוגיות

דוגמה: x=2, y=3	תיאור	פעולה
x and y = True	שני משתנים True, התשובה True	x and y
x or y = True	לפחות משתנה אחד True, התשובה True	x or y
Not (x and y) = False	הפיכת ערך לוגי	not (x and y)

Python Membership Operators Example

in , not in:

```
a, b = 10, 2
arr = [1, 2, 3, 4, 5]
if a in arr:
    print("a is in the given list")
else:
    print("a is not in the given list")
```

output:

```
a is not in the given list
b is in the given list
```

```
if b in arr:
    print("b is in the given list")
else:
    print("b is not in the given list")
```

Python בסיס השפה

loops – לולאות, while loop

```
count = 0
while count < 3:
    print("count = ", count)
    count = count + 1
print "good bye!"
```

Output:

```
count = 0
count = 1
count = 2
good bye!
```

Using else statement with while Loop :

If the **else** statement is used with
a **while** loop, the **else** statement is
executed when the condition becomes
false.

```
count = 0
while count < 3:
    print 'The count is:', count
    count = count + 1
else
    print("count is bigger than 2")
print "good bye!"
```

Output:

```
count = 0
count = 1
count = 2
count is bigger than 2
good bye!
```

Python בסיס השפה

loops – לולאות, for loop

```
for i in "java"  
    print(i)
```

Output:

j
a
v
a

Print in the same row:

```
for i in "java"  
    print(i, end = "")
```

Output:

java

```
fruits = ['banana', 'apple', 'mango']  
for fruit in fruits:  
    print ('Current fruit :', fruit)
```

Iterating by Sequence Index

```
fruits = ['banana', 'apple', 'mango']  
for index in range(len(fruits)):  
    print ('Current fruit :', fruits[index])
```

Output:

Current fruit : banana
Current fruit : apple
Current fruit : mango

Python בסיס השפה

loops – לולאות, for loop

Using else statement with for Loop

```
for num in range(3,10):
    for i in range(2, num):
        if num % i == 0:
            j = num / i
            print (num, ' equals', i, "*", j)
            break
    else:
        print (num, ' is', " prime")
```

Output:

```
2 is prime
3 is prime
4 equals 2*2.0
5 is prime
6 equals 2*3.0
7 is prime
8 equals 2*4.0
9 equals 3*3.0
```


Python בסיס השפה

loops – לולאות, for loop

Using step with for loop: for(start, stop, step)

```
for i in range (0, 10, 2):  
    print("i = ", i, ", ", end="")
```

Output:

i = 0, i = 2, i = 4, i = 8,

Using step with for loop: for(start:stop:step)

```
arr = [1,2,3,4,5,6,7,8]  
for i in arr[0:8:2]  
    print("i = ", i, ", ", end="")
```

Output: 1357

```
for i in arr[0::3]  
    print("i = ", i, ", ", end="")
```

Output: 147

```
for i in arr[::]  
    print("i = ", i, ", ", end="")
```

Output: 12345678

Python בסיס השפה

Built-in Data Types - סוגי נתונים מובנים:

Text Type:	str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	dict
Set Types:	set, frozenset
Boolean Type:	bool
Binary Types:	bytes, bytearray, memoryview

ניתן לקבל את סוג הנתונים של כל אובייקט באמצעות הפונקציה `:type()`

```
x = 5  
print(type(x))
```

Output: <class 'int'>

Python בסיס השפה

Python Collections (Arrays)

There are four collection data types in the Python programming language:

List is a collection which is ordered and changeable.
Allows duplicate members.

Tuple is a collection which is ordered and unchangeable.
Allows duplicate members.

Set is a collection which is unordered and unindexed.
No duplicate members.

Dictionary is a collection which is unordered and changeable.
No duplicate members.

Python בסיס השפה

Python Lists

דבר חשוב ברשימה הוא שפריטים ברשימה אינם צריכים להיות מאותו סוג.

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

אינדקס של איבר ראשון ברשימה הוא 0. ניתן לגשת לאברי הרשימה לפי אינדקס:

```
list2[0] → 1, list3[3] → d, list2[1:3] → 2,3
```

איך לעדכן את הרשימה:

```
list1[0] = "c++" → list1=['c++', 'chemistry', 1997, 2000]
```

הוספת איבר לסוף הרשימה:

```
list3.append("xyz") → list3 = ["a", "b", "c", "d", "xyz"]
```

מחיקת איבר מרשימה:

```
del list3[1] → list3 = ["a", "c", "d", "xyz"]  
del list3[0:2] → list3 = ["d", "xyz"]  
del list3  
del list3 → Error: list3 is not defined
```

Python בסיס השפה

Basic List Operations

Expression	Result	Description
<code>arr=[1,2,3,4], len(arr)</code>	4	Length
<code>[6,5,4] + [3,2,1]</code>	<code>[6,5,4,3,2,1]</code>	Concatenation
<code>s="hello", s2=s*2</code>	hellohello	Repetition
<code>4 in arr</code>	true	Membership
<code>print(max(arr))</code>	4	maximum

Built-in List Functions & Methods

`arr.count(obj)` - Returns count of how many times `obj` occurs in `arr`

`arr.index(obj)` - Returns the lowest index in list that `obj` appears in `arr`

`arr.insert(index, obj)` - Inserts object `obj` into list at offset `index`

`arr.remove(obj)` - Removes the first occurrence of `obj` in `arr`

`arr.reverse()` - Reverses objects of list in place

`arr.sort()` - Sorts objects of list, use compare func if given

Python בסיס השפה

Python – Dictionary

Dictionaries are used to store data values in key:value pairs.

```
thisdict = {"brand": "Ford", "model": "Mustang", "year": 1964}  
print(thisdict["brand"]) → Ford
```

Dictionary items are unordered, changeable, and does not allow duplicates.

Unordered - When we say that dictionaries are unordered, it means that the items does not have a defined order, you cannot refer to an item by using an index.

Duplicate values will overwrite existing values:

```
thisdict = {"brand": "Ford", "model": "Mustang", "year": 1964, "year": 2000}  
print(thisdict) → {"brand": "Ford", "model": "Mustang", "year": 2000}
```


Python בסיס השפה

Python – Dictionary

Dictionary Length - To determine how many items a dictionary has, use the len() function:

```
print(len(thisdict)) → 3
```

Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created:

```
del(thisdict("year"))
```

```
print(thisdict) → {"brand": "Ford", "model": "Mustang"}
```


Python בסיס השפה

Python – Dictionary

The values in dictionary items can be of any data type:

String, int, boolean, and list data types:

```
thisdict = {  
    "brand": "Ford",  
    "electric": False,  
    "year": 1964,  
    "colors": ["red", "white", "blue"]  
}  
print(thisdict) → {"brand": "Ford", "electric": False, "year": 1964, "colors":  
["red", "white", "blue"]}
```

Python Functions

In Python a function is defined using the def keyword:

```
def myFunction(food):  
    for x in food:  
        print(x, end="")
```

```
fruits = ["apple", "banana", "cherry"]  
myFunction(fruits) → apple banana cherry
```

Arbitrary Keyword Arguments, **kwargs

If the number of keyword arguments is unknown, add a double ** before the parameter name:

```
def myFunction(**kid):  
    print("His last name is " + kid["lname"])
```

```
myFunction(fname = "Moshe", lname = "Levi")
```

Functions Python

Default Parameter Value

If we call the function without argument, it uses the default value:

```
def myFunction(country = "Israel"):  
    print("I am from " + country)
```

myFunction("USA") → USA

myFunction() → Israel

Return Values

```
def maxim(arr)  
    m = arr[0]  
    for i in range(len(arr)):  
        if m < arr[i]  
            m = arr[i]  
    return m
```

arr = [4, 6, 7, 1, 0]

maxim(arr) → 7

Functions Python

The pass Statement

Function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the **pass** statement to avoid getting an error.

```
def myFunction():  
    pass
```

Recursion

```
def fact(n):  
    if n == 0:  
        return 1  
    return n*fact(n-1)
```

fact(5) → 120

Functions Python

Python - Numbers

Python supports four different numerical types

int (signed integers) - They are often called just integers or ints, are positive or negative whole numbers with no decimal point.

long (long integers) - Also called longs, they are integers of unlimited size, written like integers and followed by an uppercase or lowercase L.

float (floating point real values) - Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 10**2 = 250).

complex (complex numbers) - are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

Functions Python

Mathematical Functions

```
import math
```

```
math.sqrt(x) - The square root of x for  $x > 0$ 
```

```
math.exp(x) - The exponential of x:  $e^x$ 
```

```
math.ceil(x) - The ceiling of x: the smallest integer not less than x
```

```
math.floor(x) - The floor of x: the largest integer not greater than x
```

```
math.log(x) - The natural logarithm of x, for  $x > 0$ 
```

```
math.log10(x) - The base 10 logarithm of x for  $x > 0$ .
```

```
math.log2(x) - The base 2 logarithm of x for  $x > 0$ .
```

```
max(x1, x2, . . . xn)
```

```
min(x1, x2, . . . xn)
```

```
abs(z)
```


Python Strings

Strings

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable.

```
s1 = 'Hello World!'
```

Accessing Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

```
s1[0] → H
```

```
s1[1:7] → ello W
```

```
s1[:3] → Hel
```

```
s1[:6] + "Python" → Hello Python
```


Python Strings

In python, the string data types are immutable.

Which means a string value cannot be updated. We can verify this by trying to update a part of the string which will lead us to an error.

```
S = "abcd"  
s[0] = "x" → TypeError: str object does not support item assignment
```

Accessing Strings - continue

```
s2 = "Computer Science"  
"m" in s2 → true  
"z" not in s2 → true  
"x" in s2 → false
```

Python's triple quotes come to the rescue by allowing strings to span multiple lines, including verbatim NEWLINES, TABs, and any other special characters.

```
longString = """this is a long string that is made up of  
several lines and non-printable characters such as  
TAB ( \t ) """
```

Python Strings

Built-in String Methods

❑ `s.count(string, beg, end)`

Counts how many times string occurs in s or in a substring of s if starting index beg and ending index end are given.

```
s = "this is string example"  
n = s.count("i", 0, len(s)) → 3
```

❑ `s.endswith(suffix)`

Determines if string or a substring of s (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.

```
s = "string"  
s.endswith("ing") → true
```

❑ `s.find(string, beg, end)`

Python string method find() determines if string string occurs in s, or in a substring of s if starting index beg and ending index end are given. Returns Index if found and -1 otherwise.

```
s = "string"  
s.find("tring") → 1  
s.find("tring", 2) → -1
```

Python Strings

❑ `s = "543543"`

`s.isdigit()`

Python string method `isdigit()` checks whether the string `s` consists of digits only. This method returns `true` if all characters in the string are digits and there is at least one character, `false` otherwise.

`s.isdigit() → true`

`s = "5x43543"`

`s.isdigit() → false`

❑ `s = "to be or not to be"`

`s.replace(old, new[, max])`

Python string method `replace()` returns a copy of the string in which the occurrences of *old* have been replaced with *new*, optionally restricting the number of replacements to *max*.

`s = s.replace("be", "eat", 1) → s = "to eat or not to be"`

`s = "to be or not to be"`

`s = s.replace("be", "eat") → s = "to eat or not to eat"`

Python Tuples

A tuple is a collection of objects which ordered and immutable. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Create a tuple

```
tup1 = (12, 23, 43, "a")  
tup2 = 1, 3, 6, "tuple"
```

The empty tuple is written as two parentheses containing nothing:

```
tup3 = ()
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Accessing Values in Tuples

```
tup1[0] → 12  
tup1[1:3] → (23, 43)
```

Python Tuples

Updating Tuples

```
tup1 = (12, "a")
```

```
tup2 = 1, 3, 6, "tuple"
```

```
tup3 = tup1 + tup2 → (12, "a", 1, 3, 6, "tuple")
```

```
tup4 = tup1*2 → (12, "a", 12, "a")
```

Membership

```
12 in tup1 → true
```

```
12 not in tup1 → false
```

Iteration

```
for x in tup2:  
    print (x, end=" ")
```

Output: 1 3 6 "tuple"



את השאלות יש לשאול בפורום הקורס
(מיצוץ אישי – מודול)



PYTHON

שעור 2

אליזבת איצקוביץ, elizabeti@ariel.ac.il

מחלקה למדעי המחשב

אוניברסיטת אריאל בשומרון

Python - מחלקות ואובייקטים

Python היא שפה מונחית עצמים.

יצירת מחלקות:

מילה שמורה class מגדירה מחלקה חדשה. שם המחלקה כותבים מיד אחרי מילה class:

```
class ClassName:
```

```
    'Optional class documentation string'
```

```
    class suite
```

מחלקה מכילה שורה אופציונאלית המאפיינת את המחלקה. ה-class suite מורכב ממאפייני המחלקה:

משתני עצם, משתני המחלקה ומתודות. לדוגמה:

Python מחלקות ואובייקטים

```
class Point:
```

```
    """ this class represent a 2d point in the plane """
    num = 0
    # initilizes the point according to its coordinates: (x, y)
    # the default values: x=0, y=0
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
        Point.num = Point.num + 1

    # returns a String contains the Point data
    def __str__(self):
        return "[" + str(self.x) + "," + str(self.y) + "]"

    # returns the distance from the point to the origin
    def dist(self):
        return math.sqrt(self.x**2 + self.y**2)
```

Python - מחלקות ואובייקטים

- ❖ הפרמטר `self` הוא מצביע למופע הנוכחי של המחלקה, ומשמש לגישה למשתנים השייכים למחלקה. `self` היא לא מילה שמורה, ניתן להשתמש במילים אחרות, אבל המילה הזו **חייבת להיות פרמטר ראשון** של כל מתודה המחלקה. מקובל להשתמש במילה `self`. אין צורך בהוספת `self` בקריאה למתודה.
- ❖ משתנה `num` הוא משתנה המחלקה שערכו משותף לכל המופעים של מחלקה זו. ניתן לגשת אליו כ- `Point.num` מתוך המחלקה או מחוץ למחלקה.
- ❖ השיטה הראשונה `() __init__` היא שיטה מיוחדת, הנקראת בנאי (constructor) המחלקה או שיטת אתחול ש-Python קראה כשיוצרים מופע חדש של מחלקה זו.
- ❖ השיטה `() __str__` מחזירה מחרוזת המכילה את ערכי משתני העצם.

Python - מחלקות ואובייקטים

Underscore in Python

- ❖ `__foo__`: This is just a convention, a way for the Python system to use names that won't conflict with usernames.
- ❖ `_foo`: This is just a convention, a way for a programmer to indicate that a variable is private (whatever that means in Python).
- ❖ `__foo`: this makes a real difference: the interpreter replaces this name with `_classname__foo` to ensure that the name does not overlap with a similar name in another class.

No other form of underscore matters in the Python world

Python מחלקות ואובייקטים

יצירת אובייקטים:

האובייקט נוצר על ידי השמה של שם משתנה חדש והשוואתו לשם מחלקה:

```
# create the point - origin (0,0)
p1 = Point()
# create the point (3,4)
p2 = Point(3, 4)
print("p2 = ", p2.__str__())
```

חישוב והדפסת מרחק מ-p2 לראשית הצירים:

```
d = p2.dist()
print("distance from p2 to the origin = ", d)
# output:
p2 = [3,4]
distance from p2 to the origin = 5
```

Python מחלקות ואובייקטים

אופרטור is

אופרטור is של Python. משתמשים באופרטור is כדי להשוות את האובייקטים, לא אם הם שווים, אלא אם הם למעשה אותו אובייקט, עם אותו מקום בזיכרון.

is Returns true if both variables are the same object:

is not Returns true if both variables are not the same object

```
arr1 = [1, 3, 7]
arr2 = arr1
arr3 = [1, 3, 7]
print(arr1 is arr2) → true
print(arr1 is arr3) → false
```

פונקציה id():

הפונקציה id () מחזירה id ייחודי עבור האובייקט ספציפי. לכל האובייקטים ב-Python יש מזהה ייחודי משלו. המזהה מוקצה לאובייקט בעת יצירתו.

```
print("id(arr1) = ", id(arr1) → 1231232132165
print("id(arr2) = ", id(arr2) → 1231232132165
print("id(arr3) = ", id(arr3) → 7987798798777
```

Python מחלקות ואובייקטים

העתקה בעזרת =, העתקה עמוקה והעתקה רדודה ב-Python

ב-Python כאשר משתמשים operator = כדי ליצור עותק של אובייקט, לא נוצר אובייקט חדש;

זה נוצר רק משתנה חדש שמשתף את מצביע של האובייקט המקורי.

לכן, אם ברצוננו לשנות ערכים באובייקט החדש או באובייקט הישן, השינוי גלוי בשניהם.

```
old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 'a']]
new_list = old_list
new_list[2][2] = 99
print('Old List:', old_list)
print('ID of Old List:', id(old_list))
print('New List:', new_list)
print('ID of New List:', id(new_list))
```

Output:

```
Old List: [[1, 2, 3], [4, 5, 6], [7, 8, 99]]
ID of Old List: 140673303268168
```

```
New List: [[1, 2, 3], [4, 5, 6], [7, 8, 99]]
ID of New List: 140673303268168
```


Python מחלקות ואובייקטים

אנו משתמשים ב-`copy module` של Python לצורך פעולות העתקה רדודות ועמוקות.

נניח, שיש להעתיק את רשימת מורכבת בשם `x`:

```
import copy
scopy = copy.copy(x)
dcopy = copy.deepcopy(x)
```

העתקה רדודה יוצרת אובייקט חדש המאחסן את המצביעים האלמנטים המקוריים.

```
old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.copy(old_list)
old_list.append([4, 4, 4])
print("Old list:", old_list)
print("New list:", new_list)
```

Output:

```
Old list: [[1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

Python מחלקות ואובייקטים

עם זאת, כאשר משנים אובייקטים מקוננים ב- `old_list` השינויים מופיעים ב- `new_list`:

```
old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.copy(old_list)
old_list[1][1] = 'XX'
print("Old list:", old_list)
print("New list:", new_list)
```

Output:

```
Old list: [[1, 1, 1], [2, 'XX', 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 'XX', 2], [3, 3, 3]]
```

כאן עשינו שינוי ברשימה ישנה, ומשתנה מקונן, והשינוי עבר גם לרשימה המקורית. הסיבה לכך היא ששתי הרשימות חולקות את המצביעים של אותם אובייקטים מקוננים.

Python מחלקות ואובייקטים

העתקה עמוקה יוצרת אובייקט חדש ומוסיפה באופן רקורסיבי את העותקים של האובייקטים המקוננים הנמצאים באלמנטים המקוריים.

```
old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.deepcopy(old_list)
old_list[1][1] = 'XX'
print("Old list:", old_list)
print("New list:", new_list)

Output:
Old list: [[1, 1, 1], [2, 'XX', 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

עם זאת, כאשר מבצעים שינויים באובייקטים מקוננים באובייקט המקורי, ה- `new_list` לא משתנה.

Python מחלקות ואובייקטים

תכונות מובנות

כל מחלקה של Python מכילה מאפיינים מובנים הבאים וניתן לגשת אליהם באמצעות אופרטור נקודה כמו כל מאפיין אחר:

- **__dict__** – Dictionary containing the class's namespace.
- **__doc__** – Class documentation string or none, if undefined.
- **__name__** – Class name.
- **__module__** – Module name in which the class is defined. This attribute is **"__main__"** in interactive mode.
- **__bases__** – A possibly empty tuple containing the base classes, in the order of their occurrence in the base class list.

Python מחלקות ואובייקטים

```
print("dict: ", Point.__dict__)
print("doc: ", Point.__doc__)
print("name: ", Point.__name__)
print("module : ", Point.__module__)
print("bases: ", Point.__bases__)
```

Output:

```
dict: {'__module__': 'Point', '__doc__': 'this class represent a 2d
point in the plane ', 'num': 2, '__init__', . . . . }
```

```
doc: this class represent a 2d point in the plane
```

```
name: Point
```

```
module: Point
```

```
bases: (<class 'object'>,,)
```

Python מחלקות ואובייקטים

Destroying Objects (Garbage Collection) מחיקת אובייקטים

Python מוחק אובייקטים לא נחוצים (סוגים מובנים או מופעי מחלקה) באופן אוטומטי כדי לפנות את מקום הזיכרון. התהליך שבו Python מחזיר מעת לעת בלוקים של זיכרון שכבר אינם בשימוש נקרא Garbage Collection.

בדרך כלל לא ניתן להבחין מתי Garbage Collection מוחק אובייקט שכבר לא בשימוש ומחזיר לעצמו את הזיכרון. אך מחלקה יכולה ליישם את השיטה המיוחדת `__del__ (self)` הנקראת destructor , שמופעלת כאשר המופע עומד להיהרס. ניתן להשתמש בשיטה זו לניקוי משאבי זיכרון שאינם בשימוש:

Python מחלקות ואובייקטים

```
class Point:
```

```
    . . . . .
```

```
    def __del__(self):  
        class_name = self.__class__.__name__  
        print(class_name, "destroyed")
```

```
class TestPoint
```

```
    p1 = Point()  
    p2 = p1  
    del p1
```

Output:

```
Point destroyed  
Point destroyed
```

מקבלים שתי הודעות בגלל ש- p1 ו-p2 הם שני מצביעים על אותו אובייקט.



את השאלות יש לשאול בפורום הקורס
(מיצוץ אישי - מודול)



PYTHON

שעור 2 המשך

אליזבת איצקוביץ, elizabeti@ariel.ac.il

מחלקה למדעי המחשב

אוניברסיטת אריאל בשומרון

Python - הורשה inheritance

במקום להתחיל מאפס, ניתן ליצור מחלקה על ידי מחלקה שכבר קיימת, על ידי רישום שם מחלקת האב (**מחלקה מורשה**) בסוגריים אחרי שם המחלקה החדש (**מחלקה יורשת**, מחלקת הבן).

מחלקת הבן יורשת את התכונות של מחלקת האב שלה, וניתן להשתמש בתכונות אלה כאילו הוגדרו מחלקת הבן. מחלקת הבן יכולה גם **לדרוס (override)** את משתני עצם והשיטות ממחלקת האב.

ב-Python, כמו ב-C++ יש **הוקשה מרובה**, כלומר ניתן לרשת ממספר מחלקות, שהמספר הוא גדול או שווה 1.

משתנים "פרטיים" שלא ניתן לגשת אליהם אלא מתוך המחלקה עצמה אינם קיימים ב-Python. עם זאת, ישנה הסכמה שמשתמשים בה ברוב קודים של Python : יש להתייחס אל שם קידומת קו תחתון _ (למשל _spam כחלק שאינו ציבורי ב-API) בין אם זה פונקציה, שיטה או משתנה.

Python - הורשה inheritance

```
class JustCounter:
    __secretCounter = 0

    def __init__(self):
        self.__secretCounter = self.__secretCounter + 1

    def count(self):
        self.__secretCounter = self.__secretCounter + 1

    def getAttr(self):
        return self.__secretCounter

if __name__ == '__main__':
    print_hi('PyCharm')
    counter = JustCounter()
    counter.count()
    counter.count()
    # print(counter.__secretCounter) unresolved attribute
    # reference __secretCounter for class JustCounter
    print(counter.getAttr())
```

Output:

3

הסתרת נתונים (private)

מאפייני אובייקט עשויים להיות נראים או לא נראים מחוץ להגדרת המחלקה. כאשר מסמנים תכונות עם קידומת קו תחתון כפול, התכונות האלה אינן גלויות ישירות לגורמים חיצוניים.

inheritance הורשה - Python

file A.py

```
class A:
    def __init__(self, x=0):
        self._x = x
        print("A init")

    def get_attr(self):
        return self._x
```

file B.py

```
from A import A
class B(A):
    def __init__(self):
        super().__init__()
        print("B init")

    def method(self):
        print("child_method x = ", self._x)
```

הסתרת נתונים (protected)

מסמנים תכונות עם קידומת קו תחתון בודד, התכונות האלה גלויות ישירות ממחלקות היורשות בלבד ואינן גלויות ישירות לגורמים חיצוניים שלא יורשים ממחלקה הנוכחית.

file Test.py

```
from A import A
from B import B
a = A(3)
print(b.method())
#print(a._x) Access to a protected member _x
of a class
```

inheritance הורשה - Python

file A.py

```
class A:
    def __init__(self, x=0):
        self.x = x
        print("A init")

    def method(self):
        print("parent_method")

    def get_attr(self):
        print("x = ", self.x)

    def set_attr(self, x):
        self.x = x
```

file B.py

```
from A import A
class B(A):
    def __init__(self, x):
        super().__init__(x)
        print("B init")

    def method(self):
        print("child_method x = ", self.x)
```

file Test.py

```
from A import A
from B import B

a = A(3)
b = B(99)
b.get_attr()
b.set_attr(55)
b.get_attr()
a.method()
b.method()
```

Output:

```
A init
A init
B init
x = 99
x = 55
parent_method
child_method x = 55
```

inheritance הורשה - Python

Following table lists some generic functionality that you can override in your own classes:

1	<code>__init__ (self [,args...])</code> Constructor (with any optional arguments) Sample Call : <code>obj = className(args)</code>
2	<code>__del__(self)</code> Destructor, deletes an object, Sample Call : <code>del obj</code>
3	<code>__repr__(self)</code> Evaluable string representation, Sample Call : <code>repr(obj)</code>
4	<code>__str__(self)</code> Printable string representation, Sample Call : <code>str(obj)</code>
5	<code>__cmp__ (self, x)</code> Object comparison Sample Call : <code>cmp(obj, x)</code>



את השאלות יש לשאול בפורום הקורס
(מיצוץ אישי - מודול)



PYTHON

שעור 3 Interface

אליזבת איצקוביץ, elizabeti@ariel.ac.il

מחלקה למדעי המחשב

אוניברסיטת אריאל בשומרון

Interface - Python

ממשק משמש כתכנית לעיצוב מחלקות.

כמו מחלקה, ממשקים מגדירים שיטות. בניגוד למחלקות, שיטות אלה אבסטרקטיות. שיטה אבסטרקטית היא זו שהממשק פשוט מגדיר אותה ללא מימוש. מימוש נעשה על ידי מחלקות, אשר לאחר מכן מיישמים את הממשק ונותנים משמעות קונקרטית לשיטות האבסטרקטיות של הממשק.

גישה של פיתון לעיצוב ממשקים שונה במקצת בהשוואה לשפות כמו Java או C++. לשפות אלה יש מילת מפתח **Interface**, ואילו לפיתון אין. ממשק בפייטון זה בעצם מחלקה אבסטרקטית שמכילה רק פונקציות אבסטרקטיות. פיתון לא דורש מהמחלקה שמיישמת את הממשק להגדיר את כל השיטות אבסטרקטיות של הממשק.

Interface - Python

Informal Interfaces ממשקים לא פורמליים

בנסיבות מסוימות, ייתכן שלא נזדקק לכללים המחמירים של ממשק פייתון רשמי. האופי הדינמי של פייתון מאפשר לנו ליישם ממשק לא פורמלי. ממשק לא רשמי הוא מחלקה המגדירה שיטות שניתן לדרוס אותן, אך אין אכיפה קפדנית.

לצורך הדוגמא ניקח מחלקה אבסטרקטית שמייצגת צורה ומכילה שתי פונקציות אבסטרקטיות לחישוב שטח והיקף של צורה:

```
class Shape:
    def area(self) -> float:
        pass

    def perimeter(self) -> float:
        pass
```

Interface - Python

כדי להשתמש בממשק זה, עלינו ליצור מחלקה קונקרטית שמספקת יישום של שיטות הממשק. למטרה זו נגדיר מחלקת

Point המייצגת את נקודה במישור מיישמת (יורשת) את מחלקה אבסטרקטית **Shape**:

```
import copy
class Point(Shape):
    """this class represents point on a plane"""
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def deepcopy(self):
        p = copy.deepcopy(self)
        return p

    def str(self) -> str:
        return "[" + str(self.x) + "," + str(self.y) + "]"
```

Interface - Python

למרות ש- Point לא מיישמת את השיטות האבסטרקטיות של Shape אנחנו לא מקבלים שום שגיאה והפונקציות הבאות מחזירות True:

```
print("is Point subclass of Shape? ", subclass(Point, Shape))
print("is p instance of Shape? ", isinstance(Point(), Shape))
```

is Point subclass of Shape? True

is p instance of Shape? True

נבדוק גם את **Method Resolution Order (MRO)** של המחלקות שלנו:

```
Shape.mro(): [<class 'packInterface.Shape.Shape'>, <class 'object'>]
```

```
Point.mro(): [<class '__main__.Point'>, <class 'packInterface.Shape.Shape'>, <class 'object'>]
```

Interface - Python

Using Metaclasses **Metaprogramming**

Metaprogramming – הוא סוג של תכנות המשויך ליצירת תוכניות המייצרות תוכניות אחרות כתוצאה מעבודתם (במיוחד בזמן קומפילציה של הקוד המקורי שלהן). או שזה סוג של תוכניות שמשנות את עצמן במהלך הביצוע (קוד לשינוי עצמי). הראשון מאפשר לקבל תוכניות בזמן קצר יותר והשני מאפשר לשפר את מאפייני הקוד כמו גודל הקוד וזמן ביצוע הקוד.

Using Metaclasses

נגדיר מחלקת **Circle** המייצגת מעגל במישור ומיישמת את הממשק **Shape**:

```
class Circle:
    """this class represents circle on a plane"""
    def __init__(self, radius=1):
        self.radius = radius

    def area(self):
        return math.pi*self.radius**2

    def perimeter(self):
        return 2*math.pi*self.radius
```

אופן אידיאלי, היינו רוצים שפונקציה `issubclass (Circle, Shape)` תחזיר `False` כאשר המחלקה המיישמת לא מגדירה את כל השיטות האבסטרקטיות של הממשק.

לשם כך תיצור מטא-מחלקה בשם `ShapeMeta`. ואנחנו נצטרך לממש שתי פונקציות הבאות:

1. `__instancecheck__()`
2. `__subclasscheck__()`

Interface - Python

בבלוק הקוד שלמטה, ניצור מחלקה הנקראת `UpdatedInformalShapeInterface`
הבונה ממטא-קלאס `:ShapeMeta`

```
class ShapeMeta(type):
    def __instancecheck__(cls, instance):
        return cls.__subclasscheck__(type(instance))
    def __subclasscheck__(cls, subclass):
        return (hasattr(subclass, 'area') and callable(subclass.area) and
                hasattr(subclass, 'perimeter') and callable(subclass.perimeter))

class UpdateInformalShapeInterface(metaclass=ShapeMeta):
    pass
```

Interface - Python

נריץ את `issubclass` על המחלקות שלנו:

```
b = issubclass(Circle, UpdateInformalShapeInterface)
print("is Circle a subclass of UpdateInformalShapeInterface class? ", b)
```

ונקבל:

```
is Circle a subclass of UpdateInformalShapeInterface class? True
```

עכשיו, בואו נסתכל על ה- **MRO (Method Resolution Order)**

```
print("Shape.mro(): ", Shape.mro())
print("Circle.mro(): ", Circle.mro())
```

Output:

```
Circle.mro(): [<class 'Circle.Circle'>, <class 'object'>]
Shape.mro():  [<class 'Shape.Shape'>, <class 'object'>]
```

כפי שאנחנו יכולים לראות, `UpdatedInformalShapeInterface` הוא superclass של `Circle`, אך הוא לא מופיע ב-MRO. התנהגות החריגה זו נגרמת על ידי העובדה ש- `UpdatedInformalShapeInterface` היא מחלקת בסיס וירטואלית של `Circle`.

Interface - Python

אם נמחוק מימוש של שיטת `perimeter` במחלקת `Circle` נקבל ש-`Circle` היא לא `subclass` של מחלקת `UpdatedInformalShapeInterface`

```
b = issubclass(Circle, UpdateInformalShapeInterface)
print("is Circle a subclass of UpdateInformalShapeInterface class? ", b)
```

Output:

```
is Circle a subclass of UpdateInformalShapeInterface class? False
```



את השאלות יש לשאול בפורום הקורס
(מיצוץ אישי - מודול)



PYTHON

שעור 3 JSON

אליזבת איצקוביץ, elizabeti@ariel.ac.il

מחלקה למדעי המחשב

אוניברסיטת אריאל בשומרון

JSON - Python

אופרטור with

אופרטור with ב - Python משמש בטיפול בחריגים כדי להפוך את הקוד לנקי וקריא הרבה יותר. זה מפשט את ניהול המשאבים הנפוצים כמו זרמי קבצים. שימו לב לדוגמא הקוד הבאה כיצד השימוש ב- with משפט הופך את הקוד לנקי יותר.

```
# without using with statement
file = open('file_path', 'w')
try:
    file.write('hello world')
finally:
    file.close()

# using with statement
with open('file_path', 'w') as file:
    file.write('hello world !')
```

שימו לב שבניגוד לישום הראשון, אין צורך לקרוא לפונקציה `file.close ()` בעת שימוש אופרטור with. שימוש באופרטור with מבטיח שחרור נכון של המשאבים.

JSON - Python

כדי לעבוד עם JSON (מחרוזת או קובץ המכיל אובייקט JSON) יש להשתמש במודול json של Python:

```
import json
```

כדי להפוך מילון ל-JSON יש להשתמש בפקודת `json.dump()`:

```
person_dict = {"name": "Tom", "age": 34, "children": ["Bob", "Ron", "Bar"], "married": True}
person_json = json.dumps(person_dict)
print(person_json)
```

Output:

```
{"name": "Tom", "age": 14, "children": ["Bob", "Ron", "Bar"], "married": true}
```

JSON - Python

כדי לכתוב JSON לקובץ ב-Python אפשר להשתמש בשיטת `json.dump()`:

```
person_dict = {"name": "Tom", "age": 34, "children": ["Bob", "Ron", "Bar"],  
               "married": True}
```

```
with open("person.txt", 'w') as json_file:  
    json.dump(person_dict, json_file)
```

מקבלים קובץ טקסט ששמו `person.txt` בתוך תיקיית הפרויקט.

כדי לפענח מחרוזת של JSON אפשר להשתמש בשיטת `json.loads()` השיטה מחזירה מילון:

```
with open("person.txt", 'r') as f:  
    data = json.load(f)  
    print(data)
```

Output:

```
{'name': 'Tom', 'age': 14, 'children': ['Bob', 'Ron', 'Bar'], 'married': True}
```

JSON - Python

הנה טבלה המציגה אובייקטים של Python וההמרה המקבילה שלהם ל-JSON:

Python	JSON Equivalent
<code>dict</code>	<code>object</code>
<code>list</code> , <code>tuple</code>	<code>array</code>
<code>str</code>	<code>string</code>
<code>int</code> , <code>float</code> , <code>int</code>	<code>number</code>
<code>True</code>	<code>true</code>
<code>False</code>	<code>false</code>
<code>None</code>	<code>null</code>

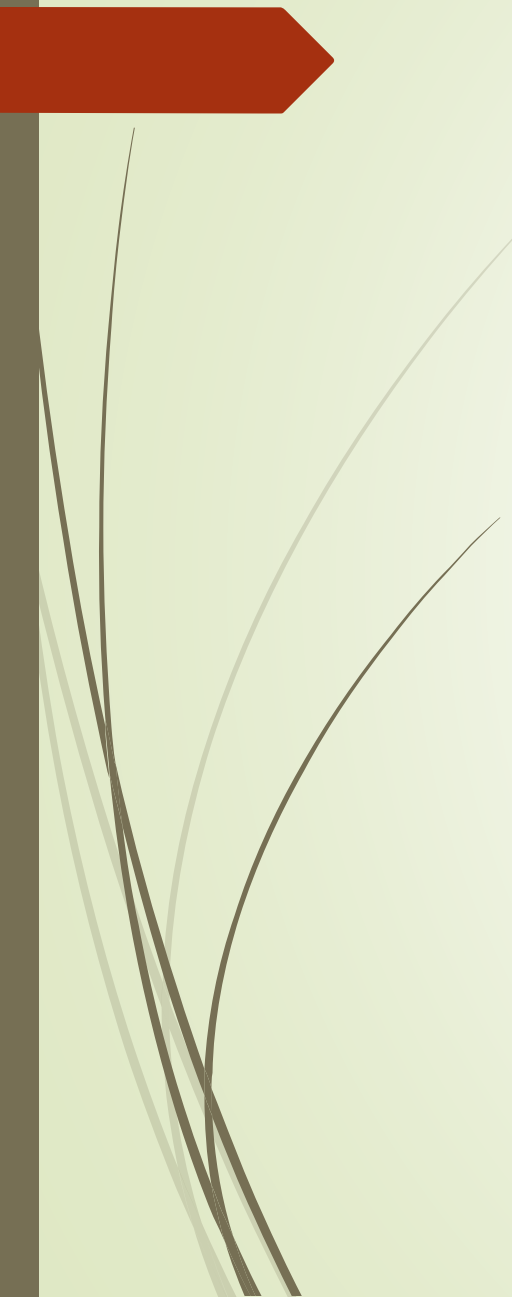
JSON - Python

Python pretty print JSON

כדי לנתח ודבג את נתוני JSON 'ייתכן שנצטרך להדפיס אותם בפורמט קריא יותר. ניתן לעשות זאת על ידי העברת פרמטרים נוספים indent and sort_keys לשיטת `json.dumps ()` ו- `json.dump`.
(.)

```
person_dict = {"name": "Tom", "age": 14, "children": ["Bob", "Ron",  
"Bar"], "married": True}  
data = json.dumps(person_dict, indent=4, sort_keys=True)  
print(data)
```

JSON - Python



```
{  
    "age": 44,  
    "children": [  
        "Bob",  
        "Ron",  
        "Bar"  
    ],  
    "married": true,  
    "name": "Tom"  
}
```

בתוכנית שלעיל השתמשנו ב-4 רווחים לצורך כניסה. והמפתחות ממוינים בסדר עולה.
אגב, ערך ברירת המחדל של כניסה הוא ללא. ערך הערך המוגדר כברירת מחדל של `sort_keys` הוא `False`



את השאלות יש לשאול בפורום הקורס
(מיצוץ אישי - מודול)



PYTHON

שעור 3 Unittest

אליזבת איצקוביץ, elizabeti@ariel.ac.il

מחלקה למדעי המחשב

אוניברסיטת אריאל בשומרון

Unittest - Python

יש בעצם שני סוגים של טסטים::

1. טסט אינטגרציה (integration test) בודק כי רכיבים ביישום פועלים זה עם זה.

2. טסט יחידה (unit test) בודק רכיבים קטנים בודדים ביישום שלנו.

בפייתון ניתן לכתוב את שני סוגם של טסטים.

כדי להשיג זאת, unittest תומך בכמה מושגים חשובים באופן מונחה עצמים:

1. יחידות טסטים (test fixture):

יחידת טסט מייצגת את ההכנה הדרושה לביצוע בדיקה אחת או יותר, וכל פעולות ניקוי נלוות. זה עשוי לכלול, למשל, יצירת מאגרי מידע זמניים או ספריות או התחלת תהליך שרת.

2. test case: בדיקת יחידות. unittest מספק מחלקת בסיס, TestCase, אשר עשויה לשמש

ליצירת מקרי בדיקה חדשים.

Unittest - Python

3. חבילת טסטים (test suite):

חבילת טסטים היא אוסף של test cases, test suites, או שניהם. היא משמשת לצבירת טסטים שיש לבצע יחד.

4. test runner

test runner הוא רכיב המתאר את ביצוע הטסטים ומספק את התוצאה למשתמש. ה-test runner יכול להשתמש בממשק גרפי, בממשק טקסטואלי, או להחזיר ערך מיוחד כדי לציין את תוצאות ביצוע הטסטים.

Unittest - Python

דוגמה: בדיקת פונקציה המקבלת מספר שלם ומחזירה אמת אם המספר הוא ראשוני:

```
import unittest
from testtests import MyFunctions

class MyTestCase(unittest.TestCase):

    def test_is_prime1(self):
        b = MyFunctions.is_prime(2)
        self.assertEqual(b, True)

    def test_is_prime2(self):
        b = MyFunctions.is_prime(8)
        self.assertEqual(b, False)

if __name__ == '__main__':
    unittest.main()
```

Output:

```
Ran 2 tests in 0.038s
OK
```

Unittest - Python

שימוש ב-runner

```
def suite():
    suit = unittest.TestSuite()
    suit.addTest(MyTestCase('test_is_prime1'))
    suit.addTest(MyTestCase('test_is_prime2'))
    return suit

if __name__ == '__main__':
    unittest.main()
    runner = unittest.TextTestRunner()
    runner.run(suite())
```

Output:

Ran 2 tests in 0.004s

OK

Unittest - Python

Method	Checks that	New in
<code>assertEqual(a, b)</code>	<code>a == b</code>	
<code>assertNotEqual(a, b)</code>	<code>a != b</code>	
<code>assertTrue(x)</code>	<code>bool(x) is True</code>	
<code>assertFalse(x)</code>	<code>bool(x) is False</code>	
<code>assertIs(a, b)</code>	<code>a is b</code>	3.1
<code>assertIsNot(a, b)</code>	<code>a is not b</code>	3.1
<code>assertIsNone(x)</code>	<code>x is None</code>	3.1
<code>assertIsNotNone(x)</code>	<code>x is not None</code>	3.1
<code>assertIn(a, b)</code>	<code>a in b</code>	3.1
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	3.1
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>	3.2
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>	3.2

Unittest - Python

The list of type-specific methods automatically used by `assertEqual()` are summarized in the following table. Note that it's usually not necessary to invoke these methods directly.

Method	Used to compare	New in
<code>assertMultiLineEqual(a, b)</code>	strings	3.1
<code>assertSequenceEqual(a, b)</code>	sequences	3.1
<code>assertListEqual(a, b)</code>	lists	3.1
<code>assertTupleEqual(a, b)</code>	tuples	3.1
<code>assertSetEqual(a, b)</code>	sets or frozensets	3.1
<code>assertDictEqual(a, b)</code>	dicts	3.1

Unittest - Python

It is also possible to check the production of exceptions.

`assertRaises()` - This statement is used to raise a specific exception

For example:

```
def sum_reverse_num(arr) -> float:
    summa = 0
    for t in arr:
        summa = summa + 1.0/t
    return summa

-----

import unittest
from testtests import MyFunctions

def test_exept(self):
    with self.assertRaises(ZeroDivisionError):
        MyFunctions.sum_reverse_num([0, 2, 10])
```

Output:

Ran 1 tests in 0.000s

OK

Unittest - Python

Calculating time difference:

```
import datetime
import time

start = time.time()
time.sleep(0.2)
end = time.time()
print("diff = ", (end - start))

-----

start = datetime.datetime.now()    #.replace(microsecond=0)
time.sleep(0.2)
end = datetime.datetime.now()    # .replace(microsecond=0)
print("diff = ", (end - start))
```

Output:

```
diff = 0.20212221145629883
diff = 0:00:00.203341
```



את השאלות יש לשאול בפורום הקורס
(מיצוץ אישי - מודול)



PYTHON

שעור 3 קבצים וטיפול בחריגים

אליזבת איצקוביץ, elizabeti@ariel.ac.il

מחלקה למדעי המחשב

אוניברסיטת אריאל בשומרון

Python - קבצים

כאשר אנו רוצים לקרוא או לכתוב לקובץ, עלינו לפתוח אותו תחילה. כשנסיים, צריך לסגור אותו כדי שהמשאבים שקשורים לקובץ ישוחררו.

ב- Python פעולות על קבצים מתבצעות בסדר הבא:

- ❖ Open a file
- ❖ Read or write (perform operation)
- ❖ Close the file

פתיחת קובץ:

```
file object = open(file name[, access mode][, buffersize])
```

Python - קבצים

❖ בשיטת `open()`, הפרמטר הראשון **name** הוא שם הקובץ כולל הנתיב שלו.

❖ פרמטר מצב הגישה **access mode** הוא פרמטר אופציונלי הקובע את מטרת פתיחת הקובץ. למשל, קרא, כתוב, הוסף וכו'. יש להשתמש ב- 'w' כדי לכתוב נתונים בקובץ וב- 'r' לקריאת נתונים.

❖ הפרמטר השלישי האופציונלי **buffer size** מציין את גודל המאגר הרצוי של הקובץ:
0 פירושו ללא `buffer`,

1 פירושו שנאגר `line buffered` וערכים חיוביים מציינים את גודל המאגר.
גודל שלילי משתמש בערך ברירת המחדל. אם לא ניתן לפתוח קובץ, אז `OSError`

קבצים - Python

The following table lists the valid values of mode parameters.

Access Modes	Description
r	Opens a file for reading only.
rb	Opens a file for reading only in binary format.
r+	Opens a file for both reading and writing.
rb+	Opens a file for both reading and writing in binary format.
w	Opens a file for writing only.
wb	Opens a file for writing only in binary format.
w+	Opens a file for both writing and reading.
wb+	Opens a file for both writing and reading in binary format.
a	Opens a file for appending.
ab	Opens a file for appending in binary format.
a+	Opens a file for both appending and reading.
ab+	Opens a file for both appending and reading in binary format.

קבצים - Python

Writing to a File:

```
f=open("myfile.txt","w")  
f.write("Hello! I love Python")  
f.close()
```

- The `f=open("myfile.txt","w")` statement opens `myfile.txt` in write mode.
- The `open()` method returns the file object and assigns it to a variable `f`.
- `"w"` specifies that the file should be writable.
- This statement stores a string in the file.
- In the end, `f.close()` closes the file object.

קבצים - Python

Python provides the **writelines()** method to save the contents of a list object in a file:

```
lines=["Hello world.\n", "I love Python & Java.\n"]  
f=open("myfile.txt","w")  
f.writelines(lines)  
f.close()
```

Reading from a File

Three different methods are provided to read data from file.

readline(): reads the characters starting from the current reading position up to a newline character.

read(chars): reads the specified number of characters starting from the current position.

readlines(): reads all lines until the end of file and returns a list object.

קבצים - Python

Reading Lines:

```
f=open("myfile.txt","r")  
line=f.readline()  
print(line)  
f.close()
```

Reading all the lines from a file, using the while loop :

```
f=open("myfile.txt","r")  
line=f.readline()  
while line!='':  
    print(line)  
    line=f.readline()
```

Python - קבצים

Use the for loop to read a file easily:

```
f=open("myfile.txt","r")
for line in f:
    print(line)
f.close()
```

File Iterator The file object has an inbuilt iterator.

```
f=open("myfile.txt","r")
while True:
    try:
        line=next(f)
        print (line)
    except StopIteration:
        f.close()
```

קבצים - Python

Append Text to a File

The "w" mode will always treat the file as a new file. In other words, an existing file opened with "w" mode will lose its earlier contents. In order to add more data to existing file use the "a" or "a+" mode.

```
f=open("myfile.txt","a+")  
f.write("I love Python & Java\n")  
line=f.readline()  
f.close()
```

Opening a file with "w" mode or "a" mode can only be written into and cannot be read from. Similarly "r" mode allows reading only and not writing. In order to perform simultaneous **read/append** operations, use "a+" mode.

Python - קבצים

Delete a File

To avoid getting an error, you might want to check if the file exists before you try to delete it:

```
import os
if os.path.exists("myfile.txt"):
    os.remove("myfile.txt")
else:
    print("The file does not exist")
```

Delete a Folder:

```
import os
os.rmdir("myfolder")
```

Note: You can only remove *empty* folders.

קבצים - Python

All methods of file object is given below:

Method	Description
<code>file.close()</code>	Closes the file.
<code>file.flush()</code>	Flushes the internal buffer.
<code>next(file)</code>	Returns the next line from the file each time it is called.
<code>file.read([size])</code>	Reads at a specified number of bytes from the file.
<code>file.readline()</code>	Reads one entire line from the file.
<code>file.readlines()</code>	Reads until EOF and returns a list containing the lines.
<code>file.seek(offset, from)</code>	Sets the file's current position.
<code>file.tell()</code>	Returns the file's current position
<code>file.write(str)</code>	Writes a string to the file. There is no return value.

Python - טיפול בחריגים exceptions

Python Exception Handling Using try, except and finally statement

Python has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong).

When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled. If not handled, the program will crash.

For example, let us consider a program where we have a function A that calls function B, which in turn calls function C. If an exception occurs in function C but is not handled in C, the exception passes to B and then to A.

If never handled, an error message is displayed and our program comes to a sudden unexpected halt.

Python - קבצים

import module sys to get the type of exception

import sys

arr = [10, 0, 2]

for entry in arr:

try:

rand = 1/int(entry)

print("rand = ", rand)

except:

print(sys.exc_info()[0], " occurred.")

Output:

rand = 0.1

<class 'ZeroDivisionError'> occurred

rand = 0.5

Python - קבצים

If no exception occurs, the **except** block is skipped and normal flow continues(for last value). But if any exception occurs, it is caught by the except block (second values).

Since every exception in Python inherits from the base Exception class, we can also perform the above task in the following way:

```
arr = [10, 0, 2]
for entry in arr:
    try:
        rand = 1/int(entry)
        print("rand = ", rand)
    except Exception as e:
        print(sys.exc_info()[0], " occurred.")
```

Output:

```
rand = 0.1
<class 'ZeroDivisionError'> occurred
rand = 0.5
```

קבצים - Python

A try clause can have any number of except clauses to handle different exceptions, however, only one will be executed in case an exception occurs.

We can use a tuple of values to specify multiple exceptions in an except clause. Here is an example pseudo code.

```
arr = [0, 4, 2]
try:
    r = 1/arr[0]
    pass
except ValueError:
    # handle ValueError exception
    pass
except (TypeError, ZeroDivisionError):
    # handle multiple exceptions
    # TypeError and ZeroDivisionError
    pass
except:
    # handle all other exceptions
    pass
```

Python - קבצים

Python try with else clause

In some situations, you might want to run a certain block of code if the code block inside try ran without any errors. For these cases, you can use the optional else keyword with the try statement.

Note: Exceptions in the else clause are not handled by the preceding except clauses.

program to print the reciprocal of even numbers

```
try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)
```

Python - קבצים

Python try...finally

The **try** statement in Python can have an optional **finally** clause. This clause is executed no matter what, and is generally used to release external resources.

```
global f
try:
    f = open("test.txt",encoding = 'utf-8')
    # perform file operations
finally:
    f.close()
```

This type of construct makes sure that the file is closed even if an exception occurs during the program execution.



את השאלות יש לשאול בפורום הקורס
(מיצוץ אישי - מודל)

Logging - Python

When we run a python script, we want to know what part of the script is getting executed and inspect what values the variables hold.

Usually, we may just 'print()' out meaningful messages so we can see them in the console. And this probably all we need when we are developing small programs.

The problem is, when we use this approach on larger projects with multiple modules we want a more flexible approach.

Logging - Python

Why?

Because, the code could go through different stages as in *development, debugging, review, testing* or in *production*.

The type of messages we want to print out during development can be very different from what we want to see once it goes into production. Depending on the purpose, we want the code to print out different types of messages.

Logging - Python

What it means by that is, during a certain '*testing*' run, we want to see only warnings and error messages.

Whereas during '*debugging*', we not only want to see the warnings and error messages but also the debugging-related messages.

Imagine doing this with 'if else' statements on a multi-module project.

If we want to print out which module and at what time the codes were run, our code could easily get messier.

Logging - Python

All these issues are nicely addressed by the **logging** module.

Using logging, you can:

- Control message level to log only required ones.
- Control where to show or save the logs.
- Control how to format the logs with built-in message templates.
- Know which module the messages is coming from.

Logging - Python

Python provides an in-built **logging** module which is part of the python standard library. So we don't need to install anything.

To use **logging**, all we need to do is setup the basic configuration using `logging.basicConfig()`.

Then, instead of `print()`, we call

`logging.{level}(message)` to show the message in console.

The printed log message has the following default

format: {LEVEL}:{LOGGER}:{MESSAGE}.

Logging - Python

The 5 levels of logging

logging has 5 different hierarchical levels of logs that a given logger may be configured to.

Let's see what the python docs has to say about each level:

Logging - Python

DEBUG: Detailed information, for diagnosing problems. *Value=10.*

INFO: Confirm things are working as expected. *Value=20.*

WARNING: Something unexpected happened, or indicative of some problem. But the software is still working as expected. *Value=30.* This is a default level.

ERROR: More serious problem, the software is not able to perform some function. *Value=40.*

CRITICAL: A serious error, the program itself may be unable to continue running. *Value=50.*

Logging - Python

```
import logging
logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
logging.basicConfig(filename='app.log', filemode='w',
                    format='%(name)s - %(levelname)s - %(message)s')
logging.warning('This will get logged to a file')
```

OUTPUT ON CONSOLE

```
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
WARNING:root:This will get logged to a file
```


Logging - Python

```
logging.basicConfig(filename='app.log', filemode='w',format='%(name)s -  
%(levelname)s -%(message)s')  
logging.warning('This will get logged to a file')  
a, b = 5, 0  
try:  
    c = a / b  
except Exception as e:
```

File app.log

```
root - WARNING - This will get logged to a file
```

```
root - ERROR - Exception occurred
```

```
Traceback (most recent call last):
```

```
File
```

```
"C:\Users\itsko\PycharmProjects\python_1\python_2\Threads\Log_Levels.py",  
line 18, in <module>
```

```
    c = a / b
```

```
ZeroDivisionError: division by zero
```



את השאלות יש לשאול בפורום הקורס
(מיצוץ אישי - מודול)