Due Date: 22.11.2019, 23:55

CENG 113
Homework #2

Preliminary

Please understand this section (and prove your understanding to yourself by writing a couple of simple programs) before diving into the homework.

1. Reading multiple inputs in one line: In Python, you can read multiple inputs in one line. For example, you can read multiple words separated by a space character:

```
words = input("Enter words:").split(" ")
```

If user enters `Python programming lang` we will have `words = ["Python", "programming", "lang"]`

We can also read inputs of other datatypes. For instance, for reading integers separated by a comma, you can use (the built-in function *map* applies given function to each item of a given iterable)

```
numbers = list(map(int, input("Enter integers separated by comma:").split(",")))
```

or alternatively (a special syntactic construct called "list comprehension"; the syntax is similar to the "set-builder notation"):

```
numbers = [int(x) for x in input("Enter integers separated by comma:").split(",")]
```

This approach will work for other separators (including multiple characters) and other datatypes (e.g. for floating-point numbers, we replace *int*s with *float*s). If you know how many values there are, you can also assign individual values to different variables:

```
x, y, z = [float(n) for n in input("Enter 3 numbers separated by dash:").split("-")]
```

In this homework, you will need such a functionality. Notice the power of abstraction here and do not let these relatively advanced constructs confuse you: you can use a functionality as a tool even if you do not fully understand its inner mechanisms. (Here is a warm up exercise for you: read a vector (a list of numbers) of arbitrary length in a line and display the sum of all the elements.)

2. Reading multiple lines into a list: You can also read inputs from multiple lines into a single list. For example, you can read 10 numbers:

```
numbers = []
for _ in range(10):
        numbers.append(float(input("Enter a number:")))
```

Or you can read integers until "end" is entered:

```
integers = []
while True:
        integer_str = input("Enter an integer (Type END if finished):")
        if integer_str.lower() == "end":
                break
        integers.append(int(integer_str))
```

Needless to say, you can also combine these two functionalities above for reading multiple lines each of which includes multiple values. We leave this as an exercise for you. In part 2 of your solution, you will perhaps do this.

3. Using mocks: Imagine in a program of yours, you have implemented 4 sequential (not nested!) conditional statements each of which consists of "if" and "else" parts. In this program there will be 2^4=16 possible paths for a run. For covering all code, instead of testing the whole "system" in 16 runs, you can test 4 individual "unit"s separately in 2 runs for each. This very useful approach of testing a function or code snippet of yours requires intervention by you as the developer (not as the user) by assigning arbitrary values to the related variables in the middle of the code.

For example, for testing the second line in the code

```
sentence = input("Enter a sentence:")
print("There are", len(sentence.split(" ")), "words")
```

you do not need to execute the code and enter a sentence as the user. You can test it quicker:

```
sentence = "This is an example." # Temporary line of code to change before deploy
print("There are", len(sentence.split(" ")), "words")
```

This way, you can also modify the existing sentence (e.g. adding new words or punctuation marks) quickly for further tests.

The difference between these two alternative tests becomes more obvious for other examples:

```
import random # See https://docs.python.org/3/library/random.html if you are curious
n = float(input(print("Enter a number:")))
a = (n**2 / 3) + (n**0.5 * 10) - 20          # A complex expression
x = a + random.randint(-5, 5)                # A stochastic expression
if x < 3: <code block 1>
elif x < 5: <code block 2>
else: <code block 3>
```

Here for testing `code block 2`, instead of trying to find a proper value for `n` and wishing ourselves good luck with the randomly-generated number (which will make `x` greater than or equal to `3` but less than `5`), we can temporarily assign a proper value to `x`:

```
x = 4                          # Temporary line
if x < 3: <code block 1>
elif x < 5: <code block 2>
else: <code block 3>
```

In your homework you may find this approach useful for testing the given "cases" in part 2.

4. Using math module: Python's `math` module defines important mathematical constants (e.g. `math.pi`) and functions (e.g. `math.factorial` as in `math.factorial(5)`). You can use them after importing `math` module once (conventionally, Python developers write `import` statements at the top of the code unless there are good reasons for doing otherwise (e.g. lazy import)). For example:

```
import math
print("5! =", math.factorial(5))
print(math.pi, "radians =", math.degrees(math.pi), "degrees")
print("arccos(1) =", math.degrees(math.acos(1)), "degrees")
```

See https://docs.python.org/3/library/math.html for other useful functions (Be careful about degrees and radians! Trigonometric functions require/return angles in radians). You may need some of such constants and functions in part 1 and thus in part 2.

Motivation: Non-Maximum Suppression

Object detection is a standard task in computer vision. The aim of a single-image object detector is detecting the minimum "bounding box" of each instance of a set of arbitrary object classes (such as automobile, plane, person, etc.) in a given image. Figure 1 shows an example result: an image and object annotations (bounding box, class label and confidence).

Object detectors do not always initially yield perfect results, and often find multiple bounding boxes for single instance. Fortunately, these bounding boxes strongly overlap and thus we can perhaps eliminate the duplicates. This post-processing stage which is called non-maximum suppression is illustrated in fig. 2.

In this homework, we implement a very simple method which can be used for non-maximum suppression. Our method, for some reason, involves fixed-size bounding circles instead of minimum bounding rectangles.

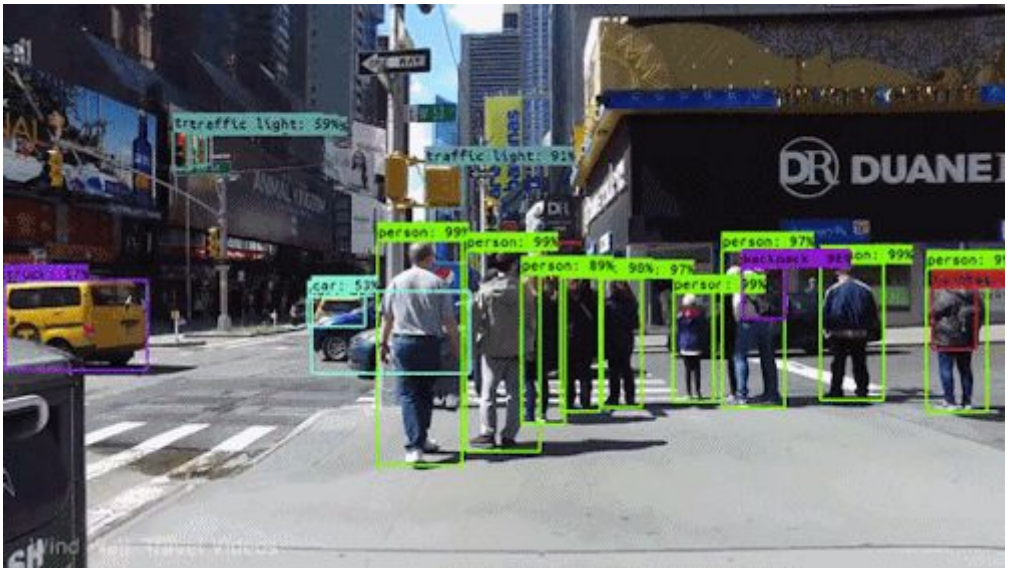

Fig. 1: Object detection

[https://towardsdatascience.com/object-detection-using-deep-learning-approaches-an-end-to-end-theoretical-perspective-4ca27eee8a9a]

Fig. 2: Non-maximum suppression

[https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c]

Part 1: Two Circles

In this subprogram, calculate the **intersection** area, **union** area and **intersection over union** (IoU = intersection / union) percentage of two given filled circles of the same size. In order to calculate these results, ask user for a single radius (Do not confuse radius with diameter! radius = diameter / 2) and the coordinates of the center of both circles (For each circle, an ordered pair of x and y values separated by a space character). You can assume user will enter a positive number for radius, and two positive numbers separated by space for each circle.

The program should run as follows (It is okay if there is small "numerical error" in the results):

```
Enter radius:10.5
Enter the first circle:1.0 10.9
Enter the second circle:5.5 8.3
Intersection: 238.3457296004042
Union: 454.3754505161452
Intersection over union: 52.45567940117732 %
```

The main challenge in this part is to find a formula for the intersection area. Do this using pen and paper. After that the solution is trivial. Test your code using the example above as well as other examples of your own.

Part 2: Many Circles

In this subprogram, **n** filled circles of the same size (defined by **radius**) and a **threshold** are given by user. You can assume a perfect user ($\mathbf{n \in \mathbb{Z}^+}$, $\mathbf{0<threshold\leq 1}$ and other assumptions in part 1). Let this list of circles is named **C** and we index individual circles as $\mathbf{C_i}$ such that $\mathbf{0 \leq i < n}$. We have formal and informal definitions of our desired functionality:

- (Formal) For all $\mathbf{C_i}$, eliminate it if and only if there is $\mathbf{C_k}$ which is not eliminated such that $\mathbf{k < i}$ and $\mathbf{IoU(C_i, C_k) \geq threshold}$.
- (Informal) Keep some of these circles and eliminate the others based on three principles:
    1. We do not want a strong overlap between any two circles. (Strong overlap means $\mathbf{IoU \geq threshold}$.)
    2. First come first served: in order to get rid of a strong overlap, we eliminate the latter. (Order of the circles in the original list is not random; they are somehow pre-sorted by user based on "confidence".)
    3. Minimal elimination policy. (Unless there is a reason to eliminate a circle, we keep it.)

Display updated list of circles called "circles after elimination" in an ascending order of indices (i.e. starting from the smallest number). Note that there is always a unique solution given any inputs for this problem (i.e. There is only one correct solution in any case). Before displaying the updated list, we also want to see intermediate results: circle pairs with their level of overlaps ($\mathbf{IoU \geq threshold}$ is strong, $\mathbf{0 < IoU < threshold}$ is weak, $\mathbf{IoU = 0}$ is no overlap. Note that in terms of elimination, "weak overlap"s are no different than "no overlap"s: we only consider "strong overlap"s). These intermediate results should (i) guide you find the solution, (ii) help you test and debug your code, (iii) help us evaluate your homework. Note that the circle pairs should include the smaller number on the left (e.g. if m < n, we never talk about "n and m").

The program should run as follows:

```
INPUTS
Enter radius:5.0
Enter the number of circles:3
Enter circle 0:1.0 2.0
Enter circle 1:3.0 4.0
Enter circle 2:50.0 60.0
Enter threshold:0.3

CIRCLE PAIRS
0 and 1 : strong overlap
0 and 2 : no overlap
1 and 2 : no overlap
```

```
CIRCLES AFTER ELIMINATION
0
2
```

The challenges in this part include (i) reading data into an appropriate structure (as we mentioned in the "Preliminary" section), (ii) iterating all "unordered pairs" of circles $\{C_i, C_k\}$, and (iii) proper elimination in every cases (Hint: start from 0 to question elimination and continue in the ascending order of the circle numbers).

For the sake of clarity, let us consider the following cases and the correct solutions:

Case: Let us have three circles (0, 1 and 2), and there be strong overlaps between "0 and 1" and between "1 and 2" (However, there is no strong overlap between "0 and 2"; there is either weak or no overlap).
Solution: Circles after elimination: 0, 2. (It is a good strategy to start with the strong overlaps in the form of "0 and x" (where x is any number) and eliminate x: Considering "0 and 1", we eliminate 1. Because we already eliminated 1, we ignore the strong overlaps which include 1 (i.e. "1 and 2"). Here ignoring means we do not need to get rid of it, the problem is already solved. There is no other other strong overlap. We will keep every circle that survived the process.)

Case: Let us have four circles (0, 1, 2 and 3), and there be strong overlaps between "0 and 1", "1 and 2", and "2 and 3".
Solution: Circles after elimination: 0, 2. (We start with considering the strong overlaps in the form of "0 and x": Because of "0 and 1", we eliminate 1. Because we already eliminated 1, we ignore the strong overlaps which include 1 (i.e. "1 and 2"). We continue with "1 and x". We do not have any. We continue with "2 and x". We have "2 and 3": we keep 2 and eliminate 3.)

Case: Let us have six circles (0, 1, 2, 3, 4 and 5), and there be strong overlaps between "0 and 1", "0 and 2", "0 and 3", "0 and 4", "1 and 2", and "4 and 5".
Solution: Circles after elimination: 0, 5. (We start with the strong overlaps in the form of "0 and x": Because of "0 and 1", "0 and 2", "0 and 3", and "0 and 4", we eliminate 1, 2, 3, and 4. These eliminations mean we ignore the strong overlaps which include 1, 2, 3, or 4 (i.e. "1 and 2" and "4 and 5").)

Case: Let us have six circles (0, 1, 2, 3, 4 and 5), and there be strong overlaps between "1 and 2", "1 and 3", "3 and 4", and "4 and 5".
Solution: Circles after elimination: 0, 1, 4 (We start with the strong overlaps in the form of "0 and x". There is no such strong overlap. We continue with "1 and x". Because of "1 and 2" and "1 and 3", we eliminate 2 and also 3. Because of these eliminations we can ignore the strong overlap between "3 and 4". Now we only have the strong overlap between "4 and 5" to consider. Considering it, we eliminate 5.)

Use these examples for having a deeper understanding of the problem as well as testing your final program (For testing purposes you can either find appropriate coordinates which will satisfy these overlaps or simply use "mocking").

Outline

Your code should be surrounded by a command-line menu in the following form:

```
# Student ID: <Student ID>
import math
while True:
    print("Menu:")
    print("[1] Two circles")
    print("[2] Many circles")
    print("[3] Exit program")
    option = input("Please enter an option:")
    if option == "1":
            <Write your subprogram 1 here>
    elif option == "2":
```

```
            <Write your subprogram 2 here>
        elif option == "3": break
        else: print("Invalid option.")
        input("Please enter to continue.")
```

Alternative Homework

You can alternatively implement a simplified version of this homework: replace circles with squares (And ask for length of an edge instead of radius). In this case, the formulas for calculating intersection and union will be considerably simpler (And there will be no need for trigonometric functions). However, your grade will be reduced by 10 points (out of 100 points).

Submission Rules

- You should submit your solution to CMS until due date.
- Your homework should be named as ceng113_hw<HomeworkNo>_<StudentID>.zip (e.g. ceng113_hw2_123456789.zip)
    - This compressed file should include 1 file: non_maximum_suppression.py
- Write your student ID as a comment at the beginning of your code.
- Cheating, including teamwork, will not be tolerated.