

Distances / square with hole / random words

CONTENTS

[Summary](#)[Learning objectives](#)[Exercise 1: distance measures](#)[Exercise 2: square with hole](#)[Suggested additional challenge](#)[Exercise 3: random words](#)[Exercise 4: clean up, comment and share](#)

Summary

This challenge focuses on building skills in Python by implementing various tasks involving command-line arguments, nested loops, and randomisation. The exercises cover calculating distance measures, creating graphical patterns in the console, and generating random words. The exercises are short and independent of one another, except that Exercise 2 reuses the distance measures from Exercise 1.

LEARNING OBJECTIVES

- **Use Command-Line Arguments:**
 - Learn how to pass and extract command-line arguments in a Python script using the `sys` module.
- **Implement Distance Calculations:**
 - Understand and implement the Euclidean and Manhattan distance formulas for given coordinates.
- **Work with Nested Loops:**
 - Use nested loops to create complex patterns, such as a square with a circular hole in the console.
- **Create Console-Based Graphics:**
 - Learn how to use the console to display simple "graphics" using text-based patterns.
- **Utilise Python Modules:**

- Gain awareness of using built-in modules like `sys` for command-line arguments and `random` for generating random values.
- Generate Random Words:
 - Use Python's `random` module to generate random words composed of lowercase alphabetic characters.
- Write and Comment Clean Code:
 - Practise writing clean, well-documented code, including adding meaningful comments for clarity.

Exercise 1: distance measures

Create a Python script named `distances.py` that calculates and outputs the Euclidean distance and the Manhattan distance between two points, based on user-defined input coordinates. That is, assuming you are in PowerShell on Windows, run the script from the command line as follows (on macOS or Linux, type `python3` in place of `py`):

```
command line
```

```
py distances.py x1 y1 x2 y2
```

where:

- `x1, y1`: the x and y coordinates of the first point.
- `x2, y2`: the x and y coordinates of the second point.

Implementation:

- Tip: you can access command-line arguments using the `sys` module:

```
python
```

```
import sys # sys is the only module you need to complete the script.
args = sys.argv
# args now contains a list -
# take a look at what is in that list and unpack it.
```

- Check if the script is provided with the correct number of command-line arguments. It should receive exactly 4 arguments. Note that the first item in the `sys.argv` list is the name of the script itself, so the list should contain 5 items. If it does not, print a message explaining how to run the script.
- Extract the values of `x1`, `y1`, `x2` and `y2` from the command-line arguments. They arrive as strings, so convert them to numbers. If any of them is not a number, print a message saying so.
- Calculate and print the Euclidean distance: `sqrt((x2 - x1)^2 + (y2 - y1)^2)`.
 - Here `^2` means 'squared'. In Python, 'to the power of' is written `**` (not `^`), and taking a square root is the same as raising to the power 0.5.
- Calculate and print the Manhattan distance: `abs(x2 - x1) + abs(y2 - y1)`.
- Put each distance calculation in its own function (for example, `euclidean_distance` and `manhattan_distance`), and call them from a `main` function.

CHECK YOUR ANSWER

running `py distances.py 0 0 3 4` should give a Euclidean distance of 5.0 and a Manhattan distance of 7.0.

Exercise 2: square with hole

Create a Python script named `hole.py` that generates and outputs a square pattern of uniform characters (using `#`) to the console. Store the side length of the square in a variable called `root` (the side of a square is the square root of its area), and set it to 51. In the centre of the square, there should be a circular "hole" with integer radius `r`, where `r` is typed in by the user when the script runs.

Constraints:

- `r` is greater than or equal to 1 and less than (`root / 2`). That is, `r` is at least 1, but less than half the side of the square.

Implementation:

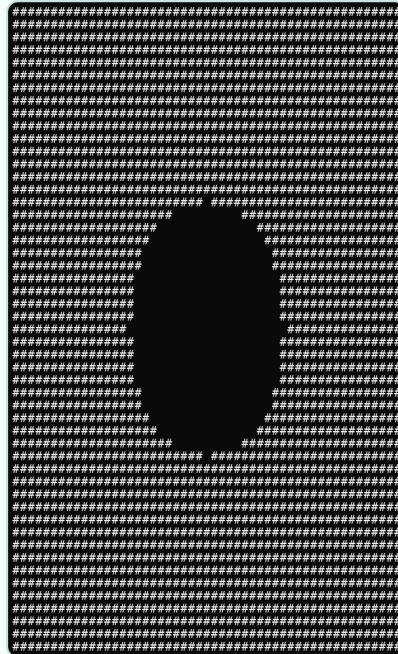
- Ask the user for `r`, and ensure that it is an integer satisfying $1 \leq r < \text{root} / 2$. Tip: `input()` returns what the user types as a string, so convert it to an integer with `int()`.
- Generate and output the square pattern with a circular hole:
 - Create a square of `#` characters with side length `root`.
 - Calculate the coordinates of the centre of the square (`middle_x`, `middle_y`).
 - Iterate through each position in the square and check if it falls within the circular hole, that is, if its distance from the centre is less than or equal to `r`.
 - For positions inside the circular hole (based on the radius `r`), replace the `#` character with a space (`" "`).
 - Display the resulting pattern on the console.
- Reuse your two distance functions from Exercise 1, and write your own functions for asking for the radius `r` and for printing the square, rather than putting everything in `main`.

Now change `root` to an even number (for example, 50) and notice the problem with the idea of the ‘middle’. Is the hole still in the centre of the square? Why not?

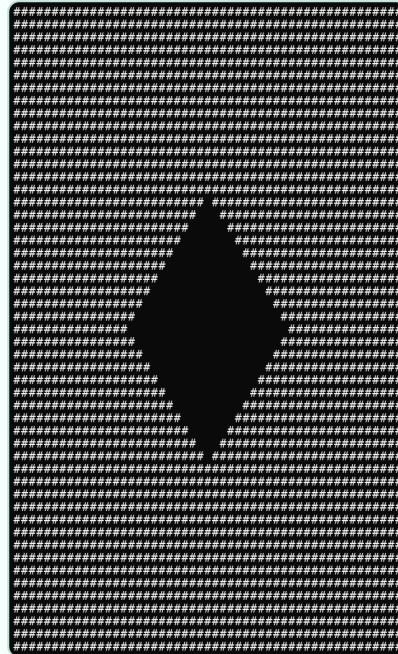
Do not worry if your circle looks taller than it is wide, as it does in the images below. Characters in a console are taller than they are wide, so a square of characters does not look square on the screen.

SUGGESTED ADDITIONAL CHALLENGE

Use different distance measures to see what happens to the characteristics of the ‘hole’, which with an assumed Euclidean distance is a circle. For example, swap the Euclidean distance for the Manhattan distance from Exercise 1, and see what shape the hole becomes. You could choose the distance measure with a command-line argument, as in Exercise 1 (for example, `py hole.py manhattan`).



Euclidean ($r = 10$)



Manhattan ($r = 10$)

Exercise 3: random words

Create a Python script named `random_words.py` that generates and prints a specified number of random words, composed of lowercase letters from the alphabet. The number of words to generate, denoted by `n_words`, will be a variable defined in the script (have its value fed in as a command-line argument, if you like).

Implementation:

- Tip: you can generate random values using the `random` module:
`random.randint(3, 10)` returns a random whole number from 3 to 10 (including 3 and 10), and `random.choice(letters)` returns one character picked at random from the string `letters`.

- If you are not using a command-line argument, then, at the start of your `main` function, define a variable `n_words` and set it to the desired number of random words to generate (e.g., 10).
- If you do use a command-line argument, check that it is a whole number of at least 1.
- Inside the script, use a loop to generate and print `n_words` random words. For each word:
 - Determine a random length for the word (between 3 and 10 characters).
 - Generate a random word of the specified length by randomly selecting lowercase letters from the alphabet.
- Write a function that returns one random word, and call it from your loop.

CHECK YOUR ANSWER

with `n_words` set to 5, your script should print 5 words, each made up of between 3 and 10 lowercase letters. Run it again and you should get different words.

Exercise 4: clean up, comment and share

It is important to comment code! You should ensure that you go over each script, clean up any code and add comments. Share your code with other students, so you can learn how they approached the problem. Each function should start with a short description of what it does.