Assignment 3 — Image Processing

A small photo editor, one function at a time

Fakhir Shaheen *(sec - A,B)*

DUE Sun 8 Nov 2026, 11:59 PM **WEIGHT** 3.375% **COVERS** Lectures 1–14 **FILES** 2 **CHECKER** `check.py`

In this assignment, you will write the tools of a small photo editor: grayscale, negative, mirror, crop, brightness, blend and rotate. A picture is a grid of coloured dots, and each tool is one function that builds a new grid from an old one.

Objectives

- Using 2D lists to represent and manipulate grid-based data
- Writing and calling helper functions that operate on structured data
- Applying nested loops to process every element in a 2D list
- Understanding why creating *new* lists (rather than modifying existing ones in place) is often safer and more predictable

Collaboration

- **Students are NOT permitted to look at or copy each other's code or "code structure" / logic. Seeking "help" from seniors or friends is also prohibited. If caught, students will receive a 0 in A3.**
- **AI tools (ChatGPT, Claude, Gemini, Copilot, or any other) are allowed ONLY for explanations, NOTHING ELSE. Not for writing code, fixing code or debugging or writing solutions. Do not paste this sheet or your file into any AI tool. If you use one in any way, it is plagiarism and you get 0 in A3.**

Although this handout is long, the information is here to provide you with context, useful examples, and hints, so be sure to read carefully.

How this assignment works

- Work top to bottom. Read a task, answer its keys, then write its code. Tasks open one at a time: the next opens when this one passes.
- ✎ means pen on this sheet, Shell closed. Only calculators are allowed. ⌨ means at the machine, in `a3.py`.
- The deliverable is `a3.py` in your `a3` folder. Tasks 1 to 7 must PASS, and each is worth the same. Task 8 is a challenge, for bonus marks.
- `check.py` checks your functions the way you can yourself. After `a3.py` has run, it types calls into the Shell, such as `invert(img)`, and then it shows `img` again, to see that your function did not change it. Run it as often as you like, and hand in from its menu.
- `a3.py` holds functions only. Test them in a file of your own (Section 9), never at the bottom of `a3.py`.
- This assignment assumes Labs 1 to 7 and uses Lectures 1 to 14.

Getting Started

A) File Setup

Before you start: get the files

Download the A3 kit from the course calendar and unzip it: `a3.py`, `image_helpers.py`, the folder `images/` (which contains several sample image files) and `check.py`. **Save them all in the same folder**, `a3`, and do not rename them.
You will edit **only** `a3.py`. Its top part, between the two `GIVEN HELPERS` lines, holds `clamp` and `create_blank_image`: use them, and do not change them. `image_helpers.py` loads, saves and shows pictures, for your own tests (Section 9). `check.py` never uses it.
If you are using Thonny, open `a3.py` and press F5: it should print nothing and stop without an error. Then open `check.py` and press F5 to check your functions.

B) The PPM File Format

The images in this assignment are stored as **PPM files** (*Portable Pixmap*). PPM is a plain-text image format, which means you can open any `.ppm` file in a text editor and read the pixel values directly. This makes it an ideal format for learning.

A P3 PPM file has the following structure:

```
P3
# This is a comment (optional)
3 2
255
255 0 0    0 255 0    0 0 255
255 255 0  255 0 255  0 255 255
```

- **Line 1:** The magic number `P3`, which identifies this as a plain-text PPM file.
- **Line 2:** An optional comment. Any line beginning with `#` is a comment.
- **Line 3:** The image dimensions — `width` followed by `height` (in pixels). Here: 3 wide, 2 tall.
- **Line 4:** The maximum colour value. This will always be `255` in this course.
- **Remaining lines:** The pixel values, row by row, left to right. Each pixel is written as three integers: `R G B`.

The file above encodes the following 2×3 image:

Red	**Green**	**Blue**
255 0 0	0 255 0	0 0 255
Yellow	**Magenta**	**Cyan**
255 255 0	255 0 255	0 255 255

Tip: Whenever your output looks wrong, open the saved `.ppm` file in a text editor and inspect the raw numbers. You can verify individual pixel values by hand, which is far more useful than staring at a picture.

C) How Images Are Represented

A digital image is a rectangular grid of **pixels**. Each pixel stores a color as three integers — one for **Red**, one for **Green**, and one for **Blue** — each in the range $[0, 255]$. This is called the **RGB color model**.

Color	RGB values	Notes
Black	`[0, 0, 0]`	All channels off
White	`[255, 255, 255]`	All channels fully on
Red	`[255, 0, 0]`	Only R channel on
Green	`[0, 255, 0]`	Only G channel on
Blue	`[0, 0, 255]`	Only B channel on
Gray	`[128, 128, 128]`	All channels equal
Yellow	`[255, 255, 0]`	R + G, no B

In this assignment, an image is stored as a **2D list of pixels**:

- `image[row][col]` gives the pixel at the given row and column.
- Each pixel is itself a **list** `[R, G, B]`, where `R`, `G`, and `B` are integers in $[0, 255]$.
- **Row 0** is the *top* row. **Column 0** is the *leftmost* column.

For example, the 2×3 image below (2 rows, 3 columns):

Red	Green	Blue
White	Gray	Black

would be stored as:

```
image = [
    [[255, 0, 0],   [0, 255, 0],     [0, 0, 255]  ],  # row 0
    [[255, 255, 255],[128, 128, 128], [0, 0, 0]    ],  # row 1
]
```

Important: Do NOT change the name, input parameters, or specifications of any of the provided functions! You may add helper functions, but changing the definitions of the given functions will cause `check.py`'s calls to fail.

Testing. `check.py` calls each function by itself, from the Shell, with small images of its own. So a function must **return** its answer and print nothing. Every function that makes an image must return a **new** one and leave the image it was given unchanged: `check.py` shows that image again after the call. If your function is correct, its row in the report says PASS:

```
Task 02  Grayscale              PASS
```

1 Pixel Basics (~15 min)

Two small functions come first. Every later function uses the same indexing, so start here.

`get_pixel(image, row, col)` returns the pixel at that row and column: the list `[R, G, B]`. The row comes first.

`get_dimensions(image)` returns the size of the image as a **tuple** `(height, width)`. The height is the number of rows. The width is the number of pixels in one row. Note the order: height first. (A PPM file writes the width first.)

TASK 1 Pixel Basics

EASY a3.py ~15 min

⌨ You will call these two functions in your later tasks.

Your task: Implement the functions `get_pixel` and `get_dimensions` according to their docstrings.

Example usage, in the Shell after you run `a3.py`:

```
>>> image = [
...     [[255, 0, 0], [0, 255, 0], [0, 0, 255]],
...     [[255, 255, 255], [128, 128, 128], [0, 0, 0]],
... ]
>>> get_pixel(image, 0, 0)
[255, 0, 0]
>>> get_pixel(image, 1, 2)
[0, 0, 0]
>>> get_dimensions(image)
(2, 3)
```

Why: Row 1 is the second row, and column 2 is its third pixel: black. The image has 2 rows of 3 pixels.

Testing: run `check.py`. It types nine calls of this kind: other pixels, a one-pixel image, an image one pixel wide.

Pen on this sheet, Shell closed, before you write Task 1 in `a3.py`. Only calculators are allowed.

Key 1.1 An image has 4 rows, and each row holds 5 pixels. Which value should `get_dimensions` give?
(a) `(5, 4)`
(b) `(4, 5)`
(c) `20`

Key 1.2 This image has 3 rows of 2 pixels.

```
image = [[[10, 20, 30], [40, 50, 60]],
         [[70, 80, 90], [100, 110, 120]],
         [[130, 140, 150], [160, 170, 180]]]
```

Predict the middle number (the G value) of the list that `get_pixel(image, 2, 1)` should give.

Answer: .....

Key 1.3 The image `[[[5, 6, 7]]]` is one pixel. Which value should `get_dimensions` give?
(a) `(1, 1)`
(b) `(1, 3)`
(c) `(0, 0)`

2 Grayscale (~40 min)

Implement the function `to_grayscale(image)` according to its docstring.

A gray pixel has three equal channels. To turn a colour pixel gray, the function works out one number from its three channels, and uses it for all three. It uses the **luminance formula**:

$$\text{gray} = \text{round}(0.299 \times R + 0.587 \times G + 0.114 \times B)$$

The weights reflect the human eye's sensitivity to green light: a green of 200 looks brighter to us

than a blue of 200. The three weights add up to 1, so white stays white and black stays black.

$$[R, G, B] \longrightarrow [\text{gray}, \text{gray}, \text{gray}]$$

Your function should return a **new** image of the same size, and leave the original unchanged. Use the given `create_blank_image` to make it, and `get_dimensions` to find its size.



TASK 2 Grayscale

MEDIUM a3.py ~40 min

This is the first function that walks over every pixel. You need a loop over the rows, and inside it a loop over the columns.

Your task: Implement the function `to_grayscale` according to its docstring.

Example usage:

```
>>> image = [[[200, 100, 50], [30, 60, 90]]]
>>> to_grayscale(image)
[[[124, 124, 124], [54, 54, 54]]]
>>> image
[[[200, 100, 50], [30, 60, 90]]]
```

Why: $0.299 \times 200 + 0.587 \times 100 + 0.114 \times 50 = 124.2$, which rounds to 124. The last line shows that `image` did not change.

Testing: run `check.py`. It covers the old tests `test_to_grayscale`, `test_to_grayscale_pure_colors` and `test_to_grayscale_no_mutation`: two images, and the first one shown again after the call.

Pen on this sheet, Shell closed, before you write Task 2 in `a3.py`. Only calculators are allowed.

Key 2.1 Predict the gray value that the pixel `[100, 200, 50]` should become.

Answer: .....

Key 2.2 Which list should `to_grayscale([[[0, 0, 255], [255, 255, 255]]])` give?

(a) `[[29, 255]]`
(b) `[[[0, 0, 29], [76, 150, 29]]]`
(c) `[[[29, 29, 29], [255, 255, 255]]]`

Key 2.3 Which pixel should turn into the darkest gray?

(a) `[255, 0, 0]`
(b) `[0, 255, 0]`
(c) `[0, 0, 255]`
(d) All three give the same gray.

Key 2.4 The user types `photo = [[[90, 30, 60]]]`, then `gray = to_grayscale(photo)`. Predict the first number of `photo[0][0]` after that.

Answer: .....

3 Invert (~20 min)

Implement the function `invert(image)` according to its docstring.

Inverting an image (also called creating a "negative") flips each channel value: a channel value of v becomes $255 - v$. This reverses bright areas to dark and vice versa. Your function should return a **new** image.

$$[R, G, B] \longrightarrow [255 - R,\ 255 - G,\ 255 - B]$$

Example:

- `[255, 0, 0]` (red) → `[0, 255, 255]` (cyan)
- `[255, 255, 255]` (white) → `[0, 0, 0]` (black)
- `[128, 64, 32]` → `[127, 191, 223]`



TASK 3 **Invert** EASY a3.py ~20 min

Same loops as Task 2, a different rule for each channel.

Your task: Implement the function `invert` according to its docstring.

Example Usage:

```
>>> image = [
...     [[255, 0, 0], [0, 255, 0]],
...     [[0, 0, 255], [128, 128, 128]],
... ]
>>> invert(image)
[[[0, 255, 255], [255, 0, 255]], [[255, 255, 0], [127, 127, 127]]]
```

Why: Each channel v becomes $255 - v$: red becomes cyan, and 128 becomes 127.

Testing: run `check.py`. It covers `test_invert` and `test_invert_no_mutation`.

Pen on this sheet, Shell closed, before you write Task 3 in `a3.py`. Only calculators are allowed.

Key 3.1 Predict the first number of the pixel that `invert([[[10, 100, 255]]])` should give.

Answer: .....

Key 3.2 `invert` gave `[[[0, 55, 200]]]`. Work out the middle number (G) of the pixel that went in.

Answer: .....

Key 3.3 The image is `[[[30, 60, 90]]]`. Predict the first number of the pixel in `invert(invert(image))`.

Answer: .....

4 Flipping (~45 min)

You will implement two flipping functions. Both should return a **new** image and leave the original unchanged.

4.1 `flip_horizontal`

Implement the function `flip_horizontal(image)` according to its docstring. This function mirrors the image left-to-right: the leftmost column becomes the rightmost column, and vice versa.

Conceptually, each row is simply **reversed**:

$$\begin{bmatrix} A & B & C \\ D & E & F \end{bmatrix} \longrightarrow \begin{bmatrix} C & B & A \\ F & E & D \end{bmatrix}$$

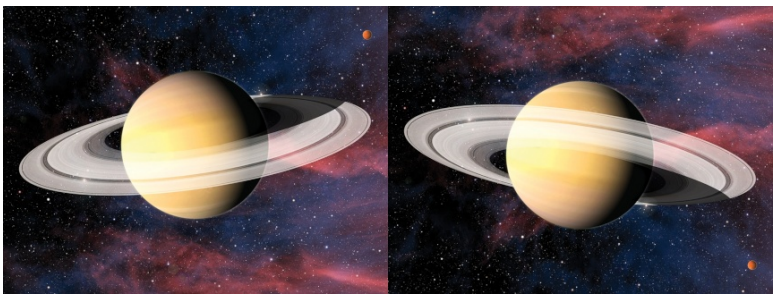


Hint: Think about how to access the pixel at column $(\text{width} - 1 - c)$ when building the new image's column c.

4.2 `flip_vertical`

Implement the function `flip_vertical(image)` according to its docstring. This function mirrors the image top-to-bottom: the top row becomes the bottom row, and vice versa.

$$\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \longrightarrow \begin{bmatrix} G & H & I \\ D & E & F \\ A & B & C \end{bmatrix}$$



Hint: Think about how to access row (height $-1-r$) of the original image when building row r of the new image.

TASK 4 Flipping

MEDIUM `a3.py` ~45 min

⌨ The task is both functions of this section. A pixel does not change; it only moves to another place.

Your task: Implement the functions `flip_horizontal` and `flip_vertical` according to their docstrings.

Example Usage:

```
>>> image = [
...     [[255, 0, 0], [0, 255, 0], [0, 0, 255]],
...     [[255, 255, 0], [255, 0, 255], [0, 255, 255]],
... ]
>>> flip_horizontal(image)
[[[0, 0, 255], [0, 255, 0], [255, 0, 0]], [[0, 255, 255], [255, 0, 255], [255, 255, 0]]]
>>> image = [
...     [[255, 0, 0], [0, 255, 0], [0, 0, 255]],
...     [[255, 255, 0], [255, 0, 255], [0, 255, 255]],
...     [[255, 255, 255], [128, 128, 128], [0, 0, 0]],
... ]
>>> flip_vertical(image)
[[[255, 255, 255], [128, 128, 128], [0, 0, 0]], [[255, 255, 0], [255, 0, 255], [0, 255,
    255]], [[255, 0, 0], [0, 255, 0], [0, 0, 255]]]
```

Why: `flip_horizontal` reverses each row: red, green, blue becomes blue, green, red. `flip_vertical` reverses the order of the rows.

Testing: run `check.py`. It covers `test_flip_horizontal`, `test_flip_horizontal_no_mutation`, `test_flip_vertical` and `test_flip_vertical_no_mutation`, and flips an image one pixel wide and an image one row high.

Pen on this sheet, Shell closed, before you write Task 4 in `a3.py`. Only calculators are allowed.

Key 4.1 An image is 7 pixels wide. `flip_horizontal` moves the pixel in column 2 of a row. Predict the column it lands in.

Answer: .....

Key 4.2 Each letter below stands for one pixel. The image has 3 rows: `[[G, H], [J, K], [M, N]]`. Predict the letter at row 0, column 1 of `flip_vertical(image)`.

Answer: .....

Key 4.3 The image has one row: `[[X, Y, Z]]`, a letter per pixel. Which image should `flip_vertical(image)` give?
(a) `[[Z, Y, X]]`
(b) `[[X, Y, Z]]`
(c) An error: there is no other row to swap with.

Key 4.4 The image is `[[P, Q, R], [S, T, U]]`, a letter per pixel. Predict the letter at row 0, column 0 of `flip_vertical(flip_horizontal(image))`.

Answer: .....

5 Crop (~40 min)

Implement the function `crop(image, top_row, left_col, bottom_row, right_col)` according to its docstring.

Cropping extracts a rectangular **sub-region** of the image. The parameters follow Python's standard slice convention: `top_row` and `left_col` are **inclusive**, while `bottom_row` and `right_col` are **exclusive**. In other words, the cropped image contains:

- Rows from `top_row` up to **but not including** `bottom_row`
- Columns from `left_col` up to **but not including** `right_col`

The resulting image has dimensions (`bottom_row` − `top_row`) × (`right_col` − `left_col`).



You may assume that the provided row and column indices are valid *(i.e., within the bounds of the image)*.

TASK 5 **Crop** MEDIUM a3.py ~40 min

The new image has its own size, so it has its own rows and columns: its row 0 is the original's `top_row`.

Your task: Implement the function `crop` according to its docstring.

Example: Given a 3×4 image:

```
image = [
    [[1,0,0],  [2,0,0],  [3,0,0],  [4,0,0] ],  # row 0
    [[5,0,0],  [6,0,0],  [7,0,0],  [8,0,0] ],  # row 1
    [[9,0,0],  [10,0,0], [11,0,0], [12,0,0]],  # row 2
]
```

Calling `crop(image, 0, 1, 2, 3)` extracts rows 0–1 and columns 1–2:

```
>>> crop(image, 0, 1, 2, 3)
[[[2, 0, 0], [3, 0, 0]], [[6, 0, 0], [7, 0, 0]]]
```

Why: Row 2 and column 3 are the ends, and the ends are not included. So the new image is 2 rows of 2 pixels.

Testing: run `check.py`. It covers `test_crop`, `test_crop_full_image`, `test_crop_single_pixel` and `test_crop_no_mutation`.

Pen on this sheet, Shell closed, before you write Task 5 in `a3.py`. Only calculators are allowed.

Key 5.1 An image is 10 pixels high and 10 wide. Predict how many pixels the image `crop(image, 1, 2, 4, 7)` should hold.

Answer: .....

Key 5.2 For Keys 5.2 and 5.3, the image has 4 rows of 5 pixels. Each pixel is written here by its first number (the second and third are 0):

```
row 0:   1   2   3   4   5
row 1:   6   7   8   9  10
row 2:  11  12  13  14  15
row 3:  16  17  18  19  20
```

Predict the first number of the pixel at row 1, column 2 of `crop(image, 1, 1, 3, 4)`.

Answer: .....

Key 5.3 Predict the first number of the top-left pixel of `crop(image, 3, 4, 4, 5)`.

Answer: .....

Key 5.4 Sana says that `crop(image, 0, 0, 2, 2)` holds 4 pixels. Bilal says 9. Who is right?
(a) Sana: row 2 and column 2 are not included.
(b) Bilal: both ends are included, so rows 0 to 2 and columns 0 to 2.
(c) Neither: it holds the 2 pixels from (0, 0) to (1, 1).

6 Brightness Adjustment (~35 min)

Implement the function `adjust_brightness(image, factor)` according to its docstring.

Each channel value is multiplied by `factor` and then rounded to the nearest integer. Since channel values must stay in the range $[0, 255]$, the result must be **clamped**:

$$\text{new_channel} = \texttt{clamp}(\text{round}(\text{channel} \times \texttt{factor}))$$

- `factor > 1`: brightens the image (values increase)
- `0 < factor < 1`: darkens the image (values decrease)
- `factor = 1`: image is unchanged

- `factor = 0`: image becomes fully black

Use the provided `clamp` function. Your function should return a **new** image.

Example:

- `[100, 150, 200]` with `factor = 2.0` → `[200, 255, 255]` *(255 clamped)*
- `[100, 150, 200]` with `factor = 0.5` → `[50, 75, 100]`
- `[80, 120, 200]` with `factor = 1.5` → `[120, 180, 255]`



TASK 6 Brightness Adjustment

MEDIUM a3.py ~35 min

`clamp` is given at the top of `a3.py`: a value under 0 becomes 0, and a value over 255 becomes 255.

Your task: Implement the function `adjust_brightness` according to its docstring.

Example Usage:

```
>>> image = [
...     [[100, 150, 200], [50, 50, 50]],
... ]
>>> adjust_brightness(image, 2.0)
[[[200, 255, 255], [100, 100, 100]]]
>>> adjust_brightness(image, 0.5)
[[[50, 75, 100], [25, 25, 25]]]
```

Why: $150 \times 2.0 = 300$ and $200 \times 2.0 = 400$ are over 255, so both become 255.

Testing: run `check.py`. It covers `test_adjust_brightness_increase`, `test_adjust_brightness_decrease`, `test_adjust_brightness_clamp` and `test_adjust_brightness_no_mutation`.

Pen on this sheet, Shell closed, before you write Task 6 in `a3.py`. Only calculators are allowed.

Key 6.1 Predict the middle number of the pixel that `adjust_brightness([[[40, 100, 160]]], 0.25)` should give.

Answer: .....

Key 6.2 Predict the middle number of the pixel that `adjust_brightness([[[50, 90, 10]]], 3)` should give.

Answer: .....

Key 6.3 The pixel is `[0, 255, 100]` and the factor is 1.2. Count how many of its three numbers should change.

Answer: .....

Key 6.4 `adjust_brightness([[[80, 40, 120]]], factor)` gave `[[[200, 100, 255]]]`. Work out the factor.

Answer: .....

7 Blending (~35 min)

Implement the function `blend(image1, image2, alpha)` according to its docstring.

Blending combines two images into one by taking a **weighted average** of their pixels. The parameter `alpha` is a float in the range $[0.0, 1.0]$ that controls how much of each image appears in the result:

$$\text{new_channel} = \text{round}(\texttt{alpha} \times \text{channel}_1 + (1 - \texttt{alpha}) \times \text{channel}_2)$$

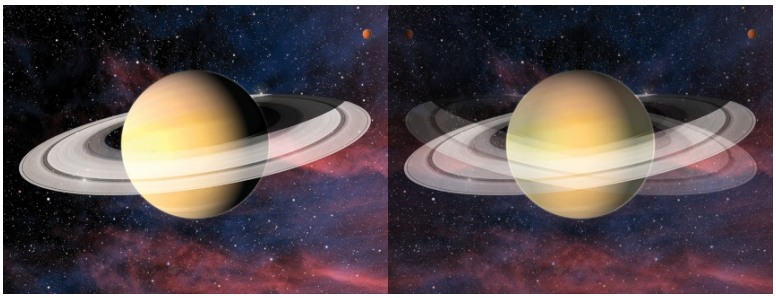
- `alpha = 1.0`: result is identical to `image1`
- `alpha = 0.0`: result is identical to `image2`
- `alpha = 0.5`: equal mix of both images

You may assume that `image1` and `image2` have the **same dimensions**. Your function should return a **new** image.

Example:

For a single pixel: blending `[200, 100, 0]` and `[0, 100, 200]` with `alpha = 0.5`:

$$\begin{bmatrix} \text{round}(0.5 \times 200 + 0.5 \times 0), & \text{round}(0.5 \times 100 + 0.5 \times 100), \\ \text{round}(0.5 \times 0 + 0.5 \times 200) \end{bmatrix} = [100, 100, 100]$$



TASK 7 **Blending** MEDIUM a3.py ~35 min

⌨ This function walks over two images at once: the pixel at `[r][c]` of the first meets the pixel at `[r][c]` of the second.

Your task: Implement the function `blend` according to its docstring.

Example Usage:

```
>>> image1 = [[[200, 100, 0],  [255, 255, 255]]]
>>> image2 = [[[0,   100, 200],[0,   0,   0]  ]]
>>> blend(image1, image2, 0.5)
[[[100, 100, 100], [128, 128, 128]]]
>>> blend(image1, image2, 1.0)
[[[200, 100, 0], [255, 255, 255]]]
>>> blend(image1, image2, 0.0)
[[[0, 100, 200], [0, 0, 0]]]
```

Why: Half of 255 plus half of 0 is 127.5, which `round` turns into 128.

Testing: run `check.py`. It covers `test_blend_half`, `test_blend_full_image1`, `test_blend_full_image2` and `test_blend_no_mutation` (both images shown again after the calls).

Pen on this sheet, Shell closed, before you write Task 7 in `a3.py`. Only calculators are allowed.

Key 7.1 Predict the middle number of the pixel that `blend([[[100, 40, 0]]], [[[0, 80, 60]]], 0.5)` should give.

Answer: .....

Key 7.2 Predict the first number of the pixel that `blend([[[200, 0, 0]]], [[[40, 0, 0]]], 0.75)` should give.

Answer: .....

Key 7.3 The user blends with `alpha = 0.2`. Which image should the result look most like?
(a) `image1`
(b) Both equally: a blend is always half and half.
(c) `image2`

Key 7.4 The image is `[[[90, 45, 10]]]`. Predict the first number of the pixel in `blend(image, image, 0.3)`.

Answer: .....

8 Rotation *(Challenge, bonus)* (~60 min)

Implement the function `rotate_90_clockwise(image)` according to its docstring.

Rotating an image 90° clockwise means that:

- The **top** row of the original becomes the **rightmost** column of the result.
- The **leftmost** column of the original becomes the **top** row of the result.
- The new image has its **height and width swapped**: if the original is $H \times W$, the result is $W \times H$.

The mapping from original coordinates (r, c) to new coordinates (r', c') is:

$$r' = c \qquad c' = H - 1 - r$$

where H is the height of the **original** image. Your function should return a **new** image of dimensions $W \times H$.

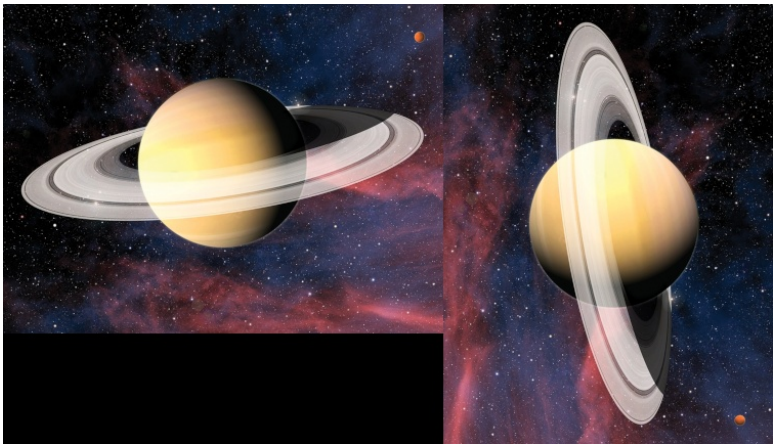
Example: Consider the following 2×3 image (2 rows, 3 columns):

$$\begin{bmatrix} A & B & C \\ D & E & F \end{bmatrix} \xrightarrow{\text{rotate 90° CW}} \begin{bmatrix} D & A \\ E & B \\ F & C \end{bmatrix}$$

The result is a 3×2 image (3 rows, 2 columns).

Verification of the formula:

- A is at $(r{=}0, c{=}0)$. New position: $r'{=}0$, $c'{=}2-1-0=1$. ✓ (top-right)
- D is at $(r{=}1, c{=}0)$. New position: $r'{=}0$, $c'{=}2-1-1=0$. ✓ (top-left)
- F is at $(r{=}1, c{=}2)$. New position: $r'{=}2$, $c'{=}2-1-1=0$. ✓ (bottom-left)



Hints:

1. The new image has a different shape than the original. Make sure you call `create_blank_image` with the *swapped* dimensions: `create_blank_image(old_width, old_height)`.
2. Loop over each pixel `(r, c)` in the original image, compute its new position `(r', c')`, and copy the pixel across.

TASK 8 Rotation

HARD a3.py ~60 min

⌨ This is the challenge, for bonus marks. `check.py` shows it as a bonus row: it opens after Task 7, and Tasks 1 to 7 pass without it.

Your task: Implement the function `rotate_90_clockwise` according to its docstring.

Example Usage:

```
>>> image = [
...     [[255, 0, 0],   [0, 255, 0],   [0, 0, 255]  ],  # row 0: R, G, B
...     [[255, 255, 0], [255, 0, 255], [0, 255, 255] ],  # row 1: Y, M, C
... ]
>>> rotate_90_clockwise(image)
[[[255, 255, 0], [255, 0, 0]], [[255, 0, 255], [0, 255, 0]], [[0, 255, 255], [0, 0,
    255]]]
```

Why: The result has 3 rows: Y, R, then M, G, then C, B. The first column of the original, read from the bottom, is the new top row.

Testing: run `check.py`. It covers `test_rotate_90_clockwise`, `test_rotate_90_clockwise_square`, `test_rotate_90_clockwise_non_square` and `test_rotate_90_clockwise_no_mutation`.

Pen on this sheet, Shell closed, before you write Task 8 in `a3.py`. Only calculators are allowed.

Key 8.1 An image has 3 rows and 5 columns. Predict how many rows the rotated image should have.

Answer: .....

Key 8.2 An image has 4 rows. The pixel at row 1, column 3 moves when the image is rotated. Predict the column it lands in.

Answer: .....

Key 8.3 The image is `[[G, H], [J, K], [M, N]]`, a letter per pixel. Predict the letter at row 0, column 0 of the rotated image.

Answer: .....

Key 8.4 The image is `[[W, X], [Y, Z]]`. Which image is it after a turn of 90° clockwise?
(a) `[[Y, W], [Z, X]]`
(b) `[[X, Z], [W, Y]]`
(c) `[[W, Y], [X, Z]]`

Key 8.5 The image is `[[W, X], [Y, Z]]` again. Predict the letter at row 0, column 0 of `rotate_90_clockwise(rotate_90_clockwise(image))`.

Answer: .....

9 Testing Your Code Visually

In addition to `check.py`, you are encouraged to test your functions visually. Create a test script that imports the modules and uses the helper functions. Save it in the same folder, as `my_tests.py` for example, never inside `a3.py`:

```
import a3
import image_helpers

img = image_helpers.load_image("images/amazon.jpg")
gray = a3.to_grayscale(img)
image_helpers.display_image(gray)
image_helpers.save_image(gray, "images/amazon_gray.jpg")
```

Change the lines for whichever function you want to test and run the file. If the visual output looks correct, your implementation is likely right. Use `save_image` to save a result for comparison.

Note. `load_image` and `save_image` work with `.ppm` files on any computer. A `.jpg` file, and `display_image`, need the free program ImageMagick. Without it, use `images/saturn.ppm`, save your result as a `.ppm` file, and open it in an image viewer.

Keep your test lines out of `a3.py`. `check.py` runs `a3.py` before every task, and the grader runs it without `image_helpers.py` and without the `images` folder. A test line inside `a3.py` makes every task fail there.

Marking and Hand-in

Marks. The marking scheme is the old one. Tasks 1 to 7 are worth 5 marks each, 35 in all: 3 for a correct function, 1 for leaving the image it was given unchanged, and 1 for the edge cases (one pixel, one row, the whole image, a factor of 0). Task 8, Rotation, is a bonus of up to 10 marks: 7 for a correct function and 3 for the edge cases.

`check.py`'s points are a first mark only: it contains only a subset of the tests that will be run to determine the grade, so passing all of its tasks does not guarantee full credit on Assignment 3. After the deadline, every function is called again on four hidden cases: a bigger image, the image after the call, and edge cases.

Naming files. Save your solutions in `a3.py`. **Do not ignore this step or save your file with a different name! `check.py` will not be able to find your file if you do and you will receive no marks.**

Final submission. **Be sure to run `check.py` and make sure all the tasks pass.** Then hand in with `check.py` too, from its menu. Only `a3.py` is sent. You can hand in as often as you like: the best whole hand-in counts. **Due Sun 8 Nov 2026, 11:59 PM.** Late: minus 10% for each day, up to three days. Nothing after that.

Before you hand in

- ☐ `check.py` says PASS for Tasks 1 to 7 (and for Task 8, if you did the challenge).
- ☐ The file is still named `a3.py`.
- ☐ The given helpers at the top of `a3.py` are as the starter gave them, and so are the names and parameters of the ten functions.
- ☐ `a3.py` has no `import`, no `print` and no test lines. Your tests are in a file of your own.
- ☐ Every function returns a new image, and you checked that the image you passed in is the same after the call.
- ☐ You looked at your results as pictures too, with `image_helpers`.
- ☐ You typed every line yourself, and can explain every line.

Appendix: Function Summary

The table below summarises all functions you must implement.

Function	Input(s)	Output
`get_pixel(image, row, col)`	image, int, int	`[R, G, B]` list
`get_dimensions(image)`	image	`(height, width)` tuple
`to_grayscale(image)`	image	new image
`invert(image)`	image	new image
`flip_horizontal(image)`	image	new image
`flip_vertical(image)`	image	new image
`crop(image, top_row, left_col, bottom_row, right_col)`	image, int×4	new image
`adjust_brightness(image, factor)`	image, float	new image
`blend(image1, image2, alpha)`	image, image, float	new image
`rotate_90_clockwise(image)`	image	new image

Given at the top of `a3.py`: `clamp(value)` and `create_blank_image(height, width, color=None)`.

Remember: Every function that returns a new image must **not** modify the original image. This is tested explicitly.

Helpful Debugging Tips

- **Off-by-one errors** are common in image processing. Double-check that your loops run over the correct range of rows and columns using `range(height)` and `range(width)`.
- **Mutation bugs:** If `check.py` reports that the image you passed in changed (the line that shows `img` again after the call), your function is changing the original image. Check that you are using `create_blank_image` and copying pixel values, not pointing the new image's pixels at the same lists as the original.
- **Printing small arrays** is often the fastest way to debug. Try constructing a 2×2 or 2×3 image by hand and printing the result of your function to check it against your manual calculation. Do this in the Shell, or in your own test file.
- **Using `display_image`** gives you quick visual feedback. If the image looks like a mirrored or rotated version of what you expected, re-examine your index arithmetic.
- **Crop bounds:** Remember that `bottom_row` and `right_col` are exclusive (just like Python list slicing). `crop(image, 0, 0, height, width)` should return a copy of the full image.
- For `rotate_90_clockwise`, verify your formula on a tiny example (e.g. 2×3) by hand before coding it up.
- If the report says `None`, your function has no `return` yet, or it prints its answer instead of returning it.