Lesson 1: ASCII art

Created: 2024-05-27T01:25:55.567405+10:00

ASCII art

ASCII art

ASCII art

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming

[1]:
```
from math import ceil
import os
from re import sub
from statistics import mean

from PIL import Image
```

In Python code, `Pillow`, the Python Imaging Library, is referred to as `PIL`. Its `Image` module has an `open()` function that can be given as argument the name of a file that contains image data, `.jpeg` (for *Joint Photographic Experts Group*) files in particular:

[2]:
```
image_file_name = 'mona_lisa.jpeg'
image = Image.open('mona_lisa.jpeg')
image
```

[2]:

That image consists of $249 \times 202 = 50298$ pixels, distributed over 249 lines of 202 pixels each. Each pixel is encoded as a triple of RGB (Red Green Blue) values, ranging between 0 and 255,

measuring the colours' intensity, 0 being darkest and 255 brightest. The `getdata()` method returns an iterator that gives access to all triples, starting with those on the line at the top and proceeding from a line to the line below if any, scanning each line from left to right:

[3]:
```
image.size, image.width, image.height
pixels = list(image.getdata())
len(pixels)
pixels[: 10]
```

[3]:
```
((202, 249), 202, 249)
```

[3]:
```
50298
```

[3]:
```
[(168, 141, 62),
 (159, 132, 53),
 (156, 129, 50),
 (164, 137, 58),
 (169, 142, 63),
 (165, 138, 59),
 (162, 135, 56),
 (164, 137, 58),
 (184, 157, 78),
 (152, 125, 46)]
```

The object returned by `Image.open()` has a `convert()` method. Given `'L'` (for *luminance*) as an argument, that method translates the image to greyscale:

[4]:
```
greyscale_image = image.convert('L')
greyscale_image
```

[4]:

Applied to a greyscale image, the `getdata()` method returns an iterator that gives access to the luminance levels for all pixels, also numbers between 0 and 255:

[5]:
```
greyscale_pixels = list(greyscale_image.getdata())
greyscale_pixels[: 10]
```

[5]:
```
[140, 131, 128, 136, 141, 137, 134, 136, 156, 124]
```

The luminance of a pixel is computed as the following weighted sum of red, green and blue intensities, illustrated with the first 10 pixels of the colour image:

[6]:
```
[round(0.299 * R + 0.587 * G + 0.114 * B) for (R, G, B) in pixels[: 10]]
```

[6]:
```
[140, 131, 128, 136, 141, 137, 134, 136, 156, 124]
```

The `crop()` method returns that part of the image that fits within a rectangle with at the top left and bottom right corners, pixels whose coordinates make up the 4-element tuple passed as argument to the method. For instance, the following returns the top third left half of the greyscale image:

[7]:
```
greyscale_image.crop((0, 0, greyscale_image.width // 2,
                             greyscale_image.height // 3
                      )
                     )
```

[7]:

To convert the black and white image of Mona Lisa's painting into ASCII art, we will replace blocks of pixels of size $n \times n$ for a given natural number n, obtained with `crop()`, by ASCII characters, the latter being all the more "dense" that the average luminance of the former is low. The following is a string of 10 characters that can arguably be claimed to be listed from most dense to least dense:

[8]:
```
character_ramp = '@%#*+=-:. '
```

We will create a `.pdf` file that displays the ASCII characters in such a way that each character will be separated from a neighbouring (immediately to the left, to the right, above or below) character by 2.5mm. We want the overall ASCII picture to fit in a rectangle of width 200mm and height 250mm, which implies that there can be at most $\frac{200}{2.5} = 80$ characters horizontally, and at most $\frac{250}{2.5} = 100$ characters vertically. That translates in the following lower bound on the value of n:

[9]:
```
min_block_size = max(ceil(greyscale_image.width / 80),
                     ceil(greyscale_image.height / 100)
                    )
min_block_size
```

[9]: 3

If n does not divide `greyscale_image.width` or `greyscale_image.height` then we will crop the image horizontally or vertically, respectively, ignoring the same number (possibly ± 1) of pixels on the left and right hand sides of the image, and ignoring the same number (possibly ± 1) of pixels at the top and bottom of the image. The following function maps n to the 4-tuple consisting of

- the horizontal offset (number of pixels skipped on the left hand side),
- the vertical offset (number of pixels skipped at the top),
- the number of blocks to extract in the horizontal dimension from the greyscale image, or equivalently, the number of characters on a line of the ASCII picture, and
- the number of blocks to extract in the vertical dimension from the greyscale image, or equivalently, the number of characters on a column the of ASCII picture :

[10]:
```
def dimensions(block_size):
    return greyscale_image.width % block_size // 2,\
           greyscale_image.height % block_size // 2,\
           greyscale_image.width // block_size,\
           greyscale_image.height // block_size
```

[11]:
```
dimensions(3)
dimensions(5)
dimensions(10)
dimensions(15)
```

[11]: (0, 0, 67, 83)

[11]: (1, 2, 40, 49)

[11]: (1, 4, 20, 24)

[11]: (3, 4, 13, 16)

Let us set n to the minimal value and record the corresponding 4-tuple:

[12]:
```
block_size = min_block_size
x_offset, y_offset, width, height = dimensions(block_size)
```

The block of 3 by 3 pixels that is the 10th from the left and the 25th from the top of the cropped greyscale image consists of 9 pixels whose luminance values are recorded in the following list:

[13]:
```
selected_block = greyscale_image.crop((x_offset + 9 * block_size,
                                       y_offset + 24 * block_size,
                                       x_offset + 10 * block_size,
                                       y_offset + 25 * block_size
```

```
                                        )
                                       ).getdata()
list(selected_block)
```

[13]: `[182, 175, 167, 184, 182, 164, 176, 183, 170]`

Here is the mean of those values:

[14]:
```
selected_block_mean_luminance = mean(selected_block)
selected_block_mean_luminance
```

[14]: `175.88888888888889`

Let us record the minimal and maximal luminance values in the greyscale image as well as the length of the interval they determine:

[15]:
```
max_luminance = max(greyscale_pixels)
min_luminance = min(greyscale_pixels)
luminance_range = max_luminance - min_luminance
min_luminance, max_luminance, luminance_range
```

[15]: `(2, 223, 221)`

Dividing the interval of real values between `min_luminance` and `max_luminance` into `len(character_ramp)` many intervals of equal length, we can compute the index of the interval where `mean_luminance` falls as follows. Note that in case the luminance of all pixels is `max_luminance`, the index should not be `len(character_ramp)` but `len(character_ramp) - 1`, which justifies the use of `min()` in the expression below:

[16]:
```
selected_block_index = min(len(character_ramp) - 1,
                           int((selected_block_mean_luminance - min_luminance)\
                               / luminance_range * len(character_ramp)
                              )
                          )
selected_block_index
```

[16]: `7`

So the ASCII character to replace the block under consideration is the following:

[17]:
```
character_ramp[selected_block_index]
```

[17]: `':'`

These computations have to be performed for all blocks of pixels to extract from the image. The following function puts together the code written for the selected block so that it can be called with first and second arguments referring to the vertical and horizontal indexes of the block to work with, respectively.

[18]:
```
def character_for_block(j, i, x_offset, y_offset, width, height,
                        block_size, character_ramp, greyscale_image
                       ):
    y = y_offset + j * block_size
    x = x_offset + i * block_size
    return character_ramp[min(len(character_ramp) - 1,
                              int((mean(greyscale_image.crop((x, y,
                                                              x + block_size,
                                                              y + block_size
                                                             )
                                                            ).getdata()
                                        ) - min_luminance
                                   ) / luminance_range * len(character_ramp)
                                  )
                             )
                           ]
```

[19]:
```
character_for_block(24, 9, *dimensions(block_size), block_size,
                    character_ramp, greyscale_image
                   )
```

[19]:
```
':'
```

To get the ASCII symbol for each block of pixels extracted from the image, we need to call character_for_block() with the first argument ranging between 0 and `dimensions(block_size)[4] - 1`, and the second argument ranging between 0 and `dimensions(block_size)[3] - 1`.

To create the `.pdf` file, we let the code create a `.tex` file structured as follows:

```
\documentclass[10pt]{article}
\usepackage{fancyvrb}\DefineShortVerb{\º}
\usepackage{tikz}
\usepackage[margin=0cm]{geometry}

\begin{document}

\vspace*{\fill}
\begin{center}
\begin{tikzpicture}[x=2.5mm, y=-2.5mm]
\node at (0, 0) {º-º};
\node at (1, 0) {º-º};
\node at (2, 0) {º-º};
...
\node at (64, 82) {º@º};
\node at (65, 82) {º%º};
\node at (66, 82) {º%º};
\end{tikzpicture}
\end{center}
\vspace*{\fill}
```

```
\end{document}
```

This is Latex code that can be understood as follows:

- `\documentclass[10pt]{article}`

  is used to create a document of the article class, with 10pt as font size.

- `\usepackage{fancyvrb}\DefineShortVerb{\º}`

  allows us to write `º...º` for `\verbº...º`. The purpose of `\verbº...º` is to write `...` as verbatim text, which is necessary because many characters have a special meaning in Latex, and `\verbº...º` offers a way to treat all characters in `...` literally. In general, `\verb` expects `...`, the characters to write literally, to be surrounded by an arbitrary character that is not one of those in `....` In order to allow `...` to include any ASCII character, without exception, we have chosen as surrounding character a non-ASCII character, `º`, which on a Mac is obtained by pressing Option 0; but any other non-ASCII character would do.

- `\usepackage{tikz}`

  is needed to use the tikz package and create a tikz picture, thanks to the commands preceded and followed by

  `\begin[x=2.5mm, y=-2.5mm]{tikzpicture}`

  `\end{tikzpicture}`

  The options `x=2.5mm` and `y=-2.5mm` change the coordinate sytem so that a point of coordinate (i, j) is mapped to the point of coordinates $(0.25i, -0.25j)$. It is indeed preferable to use integers to refer to the ASCII characters to write, (i, j) denoting the $(i + 1)$th character on the $(j + 1)$th line. For Pillow, the y axis points downwards whereas for tikz, the y axis points upwards, consistently with standard mathematical practice, which explains the minus sign in $(0.25i, -0.25j)$.

- `\usepackage[margin=0cm]{geometry}`

  is used to suppress all margins, making it easier to centre the ASCII picture on the page, and also suppress the page number.

- `\vspace*{\fill}`

  `\vspace*{\fill}`

  is used to centre the ASCII picture vertically, adding the same amount of space at the top and at the bottom of the page (without * after `\vspace`, vertical space would be ignored at those locations).

- `\begin{center}`

  `\end{center}`

  is used to centre the the ASCII picture horizontally.

- The ASCII picture is created thanks to commands of the form `\node at (i, j) {ºcº};`, the number of such commands being the number of blocks extracted from the greyscale image, to write character c associated with the block that has i blocks to the left and j blocks above

at the right location. So these commands write the ASCII characters from the one in the top left corner to the one in the bottom right corner of the picture.

We let the name of the created `.tex` file start with `mona_lisa_` followed by an indication of the block size and the code points of the characters in the character ramp (the characters themselves could be invalid in file names). The following function returns this (pretty long) name:

[20]:
```
def filename_for_ascii_picture(block_size, character_ramp):
    return sub('\..*', '', image_file_name) + '_'\
                        + str(block_size) + '_as_block_size_'\
                        + '_'.join(str(ord(e)) for e in character_ramp)\
                        + '_as_character_ramp'

filename_for_ascii_picture(block_size, character_ramp)
```

[20]:
```
'mona_lisa_3_as_block_size_64_37_35_42_43_61_45_58_46_32_as_character_ramp'
```

We can now complete the implementation with a function that, given the block size and the character ramp,

- calls `dimensions()`;
- calls `filename_for_ascii_picture()`;
- computes how many blocks fit in the image horizontally and vertically, so how many ASCII characters to write horizontally and vertically;
- writes to the desired `.tex` file the Latex commands previously described, making use of `character_for_block()`, called within two loops to range over the extracted blocks of pixels in the greyscale image;
- calls `os.system()` to call `lualatex` to create the `.pdf` file from the `.tex` file, both having the same name (the extension excluded), which also creates two intermediate files with that name and extensions `.aux` and `.log`, that are eventually deleted by calls to `os.remove()`.

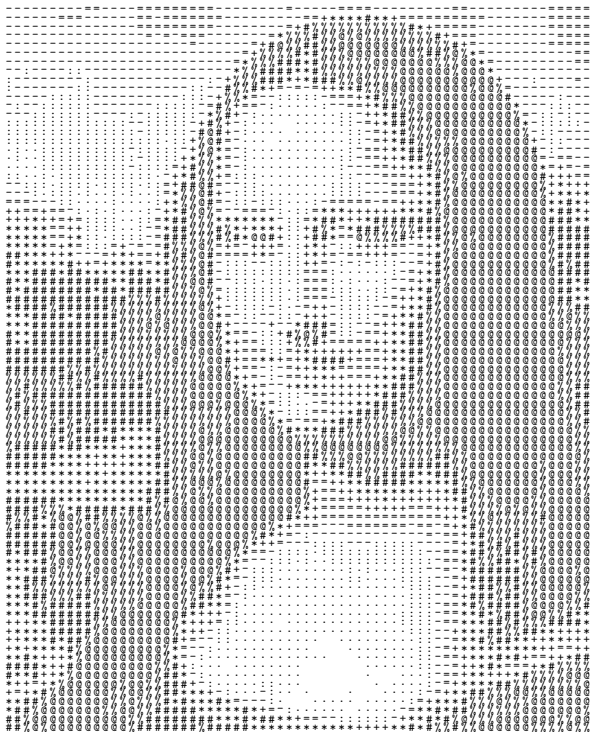
[21]:
```
def create_ascii_picture(block_size, character_ramp):
    x_offset, y_offset, width, height = dimensions(block_size)
    ascii_image_name = filename_for_ascii_picture(block_size, character_ramp)
    tex_filename = ascii_image_name + '.tex'
    width = greyscale_image.width // block_size
    height = greyscale_image.height // block_size
    with open(ascii_image_name + '.tex', 'w') as tex_file:
        print('\\documentclass[10pt]{article}\n'
              '\\usepackage{fancyvrb}'
              '\\DefineShortVerb{\\º}\n'
              '\\usepackage{tikz}\n'
              '\\usepackage[margin=0cm]{geometry}\n'
              '\n'
              '\\begin{document}\n'
              '\n'
              '\\vspace*{\\fill}\n'
              '\\begin{center}\n'
              '\\begin{tikzpicture}[x=2.5mm, y=-2.5mm]', file=tex_file
```

```
            )
        for j in range(height):
            y = y_offset + j * block_size
            for i in range(width):
                x = x_offset + i * block_size
                c = character_for_block(j, i, x_offset, y_offset, width,
                                        height, block_size, character_ramp,
                                        greyscale_image
                                       )
                # Using {{ and }} to output { and }, respectively.
                print(f'\\node at ({i}, {j}) {{º{c}º}};', file=tex_file)
        print('\\end{tikzpicture}\n'
              '\\end{center}\n'
              '\\vspace*{\\fill}\n\n'
              '\\end{document}', file=tex_file
             )
    os.system('lualatex ' + tex_filename)
    for file in (ascii_image_name + ext for ext in ('.aux', '.log')):
        os.remove(file)
```

Let us test our function with `block_size` set to 3 and `character_ramp` to `'@%#*+=-:. '` (lualatex takes a few seconds to generate the `.pdf` file):

[22]:
```
create_ascii_picture(3, '@%#*+=-:. ')
```

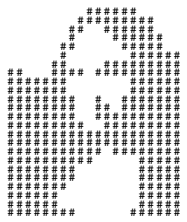
That should create `.tex` and `.pdf` files of name (without extension) `mona_lisa_3_as_block_size_64_37_35_42_43_61_45_58_46_32_as_character_ramp`, the `.pdf` file viewing as follows:



Let us test our function with `block_size` set to 10 and `character_ramp` to `'# '`:

[23]:
```
create_ascii_picture(10, '# ')
```

That should create a `.tex` and `.pdf` files of name (without extension) `mona_lisa_10_as_block_size_35_32_as_character_ramp`, the `.pdf` file viewing as follows:



The program `ascii_art.py` does not define functions but does not hard code the name of the image and is meant to be run from the command line, making use of command line arguments:

- the required `--image_file` argument, to be followed by the name of the image file,
- the optional `--block_size` argument, set to 3 by default,
- the optional `--character_ramp` argument, set to `'@%#*+=-:. '` by default.

So instead of both calls to `create_ascii_picture()` above, we could execute:

- `python3 ascii_art.py --image_file mona_lisa.jpeg`
- `python3 ascii_art.py --image_file mona_lisa.jpeg --block_size 10 --character_ramp '# '`