

# Writing readable code

Robert Haase

December 2022

## Recommendations for readable code

- Every command belongs on its own line
- Insert empty lines to separate important processing steps.
- Put comments to introduce sections
- Print out intermediate results
- Put spaces between operators and operands, because:

This is easier to read than that, or isn't it?

- Keep code blocks it short.
- Consider writing code out instead of keeping it short.

```
[1]: [2]: # initialize program
a=5
b=3
c=8

# compute result before we can evaluate it
d=(a+b)/c
```

```
[3]: d
```

```
[3]: 1.0
```

```
[4]: d = (a + b) / c
d
```

```
[4]: 1.0
```

```
[5]: print("Yin" if d==5 else "Yang")
Yang
```

```
[6]: if d == 5:
    print("Yin")
else:
    print("Yang")
```

Yang

- Write code so that others can read it. Your future self will thank you.
- Name variables and functions after English words.



```
A = w * h  
print( A )
```

```
nNucl = cntNcl(img)
```

```
volume = travelled_distance / time
```

```
percentage = "Robert"
```



```
area = width * height  
print( area )
```

```
number_of_nuclei = count_nuclei(image)
```

```
numberOfNuclei = countNuclei(image)
```

```
speed = travelled_distance / time
```

Snake-case

Camel-case

- Name variables using nouns
- Name functions as verb + noun

# Don't repeat yourself (DRY)

- Use for-loops to prevent code duplication. It also makes code maintenance easier.

```
[8]: image = cle.imread("../data/BBBC007_batch/17P1_POS0013_D_1UL.tif")
      labels = cle.voronoi_otsu_labeling(image, spot_sigma=3)
      number_of_nuclei = labels.max()
      number_of_nuclei
```

[8]: 44.0

```
[9]: image = cle.imread("../data/BBBC007_batch/20P1_POS0005_D_1UL.tif")
      labels = cle.voronoi_otsu_labeling(image, spot_sigma=3)
      number_of_nuclei = labels.max()
      number_of_nuclei
```

[9]: 41.0

```
[10]: image = cle.imread("../data/BBBC007_batch/20P1_POS0007_D_1UL.tif")
       labels = cle.voronoi_otsu_labeling(image, spot_sigma=3)
       number_of_nuclei = labels.max()
       number_of_nuclei
```

[10]: 73.0

```
[11]: folder = "../data/BBBC007_batch/"
      files = ["17P1_POS0013_D_1UL.tif",
               "20P1_POS0005_D_1UL.tif",
               "20P1_POS0007_D_1UL.tif"]
```

```
[12]: for file in files:
       image = cle.imread(folder + file)
       labels = cle.voronoi_otsu_labeling(
           image,
           spot_sigma=3)
       number_of_nuclei = labels.max()
       print(file, number_of_nuclei)
```

17P1\_POS0013\_D\_1UL.tif 44.0

20P1\_POS0005\_D\_1UL.tif 41.0

20P1\_POS0007\_D\_1UL.tif 73.0

# Don't repeat yourself (DRY)

- Use functions to prevent code duplication. It also makes code more flexible.

Default  
parameter

```
[8]: image = cle.imread("../data/BBBC007_batch/17P1_POS0013_D_1UL.tif")
labels = cle.voronoi_otsu_labeling(image, spot_sigma=3)
number_of_nuclei = labels.max()
number_of_nuclei
```

[8]: 44.0

```
[9]: image = cle.imread("../data/BBBC007_batch/20P1_POS0005_D_1UL.tif")
labels = cle.voronoi_otsu_labeling(image, spot_sigma=3)
number_of_nuclei = labels.max()
number_of_nuclei
```

[9]: 41.0

```
[10]: image = cle.imread("../data/BBBC007_batch/20P1_POS0007_D_1UL.tif")
labels = cle.voronoi_otsu_labeling(image, spot_sigma=3)
number_of_nuclei = labels.max()
number_of_nuclei
```

[10]: 73.0

```
[13]: def count_nuclei(image, spot_sigma=3):
      labels = cle.voronoi_otsu_labeling(
          image,
          spot_sigma=spot_sigma)
      number_of_nuclei = labels.max()

      return number_of_nuclei
```

```
[14]: count_nuclei(cle.imread(folder + files[0]))
```

[14]: 44.0

```
[15]: count_nuclei(cle.imread(folder + files[1]))
```

[15]: 41.0

```
[16]: count_nuclei(cle.imread(folder + files[2]))
```

[16]: 73.0

```
[18]: count_nuclei(cle.imread(folder + files[2]), spot_sigma=5)
```

[18]: 68.0

- Put all parameters of your workflow on top so that one can easily spot them.
- Give them reasonable names



The diagram illustrates the process of refactoring code to avoid magic numbers. A green arrow points from the original code (left) to the refactored code (right). A red arrow points from the output of the original code to the output of the refactored code.

**Original Code (Left):**

```
[3]: image = imread("../data/BBBC007_batch/17P1_POS0013_D_1UL.tif")

# noise removal
blurred = gaussian(image, 7)

# instance segmentation
binary = blurred > threshold_otsu(blurred)
labels = label(binary)

# quantitative measurement
labels.max()
```

**Refactored Code (Right):**

```
[4]: # enter the image filename to be processed here
file_to_process = "../data/BBBC007_batch/17P1_POS0013_D_1UL.tif"

# enter the expected radius of nuclei here, in pixel units
approximate_nuclei_radius = 3

[5]: image = imread(file_to_process)

# noise removal
blurred = gaussian(image, approximate_nuclei_radius)

# instance segmentation
binary = blurred > threshold_otsu(blurred)
labels = label(binary)

# quantitative measurement
labels.max()
```

**Outputs:**

Original output: [3]: 19

Refactored output: [5]: 37

- Prevent long, complicated macros (“spaghetti code”)

```
[2]: image = imread("../data/blobs.tif")
      footprint = disk(15)
      background_subtracted = white_tophat(image,
                                           footprint=footprint)

      particle_radius = 5
      denoised = gaussian(background_subtracted,
                          sigma=particle_radius)
      binary = denoised > threshold_otsu(denoised)
      labels = label(binary)
      requested_measurements = ["label", "area", "mean_intensity"]
      regionprops = regionprops_table(image,
                                      labels,
                                      properties=requested_measurements)

      table = pd.DataFrame(regionprops)
      mean_total_intensity = np.mean(table["area"] * table["mean_intensity"])
      mean_total_intensity
```

- At least: Write sections of code with headlines

```
[3]: # configuration
      file_to_analyze = "../data/blobs.tif"
      background_subtraction_radius = 15
      particle_radius = 5
      requested_measurements = ["area", "mean_intensity"]

      # Load data
      image = imread(file_to_analyze)

      # preprocess image
      footprint = disk(background_subtraction_radius)
      background_subtracted = white_tophat(image,
                                           footprint=footprint)
      denoised = gaussian(background_subtracted,
                          sigma=particle_radius)

      # segment image
      binary = denoised > threshold_otsu(denoised)
      labels = label(binary)

      # extract features
      regionprops = regionprops_table(image,
                                      labels,
                                      properties=requested_measurements)

      table = pd.DataFrame(regionprops)

      # descriptive statistics
      mean_total_intensity = np.mean(table["area"] * table["mean_intensity"])
      mean_total_intensity
```

- Prevent long, complicated macros (“spaghetti code”)

```
[4]: # reusable functions
➡ def preprocess_image(image, background_subtraction_radius, particle_radius):
    """Apply background removal and denoising"""
    footprint = disk(background_subtraction_radius)
    background_subtracted = white_tophat(image, footprint=footprint)
    denoised = gaussian(background_subtracted, sigma=particle_radius)
    return denoised

➡ def segment_image(image):
    """Apply thresholding and connected component analysis"""
    binary = image > threshold_otsu(image)
    labels = label(binary)
    return labels
```

- At least: Write sections of code with headlines
- Even better: Write custom functions



- *Divide* complex issues into smaller, accessible issues like

```
def analyse_average_total_intensity(filename,
                                   background_subtraction_radius = 15,
                                   particle_radius = 5):
    """Load an image, segment objects and measure their mean total intensity."""
    image = imread(filename)
    denoised = preprocess_image(image,
                                background_subtraction_radius,
                                particle_radius)
    labels = segment_image(denoised)
    requested_measurements = ["area", "mean_intensity"]
    table = extract_features(image,
                             labels,
                             requested_measurements)

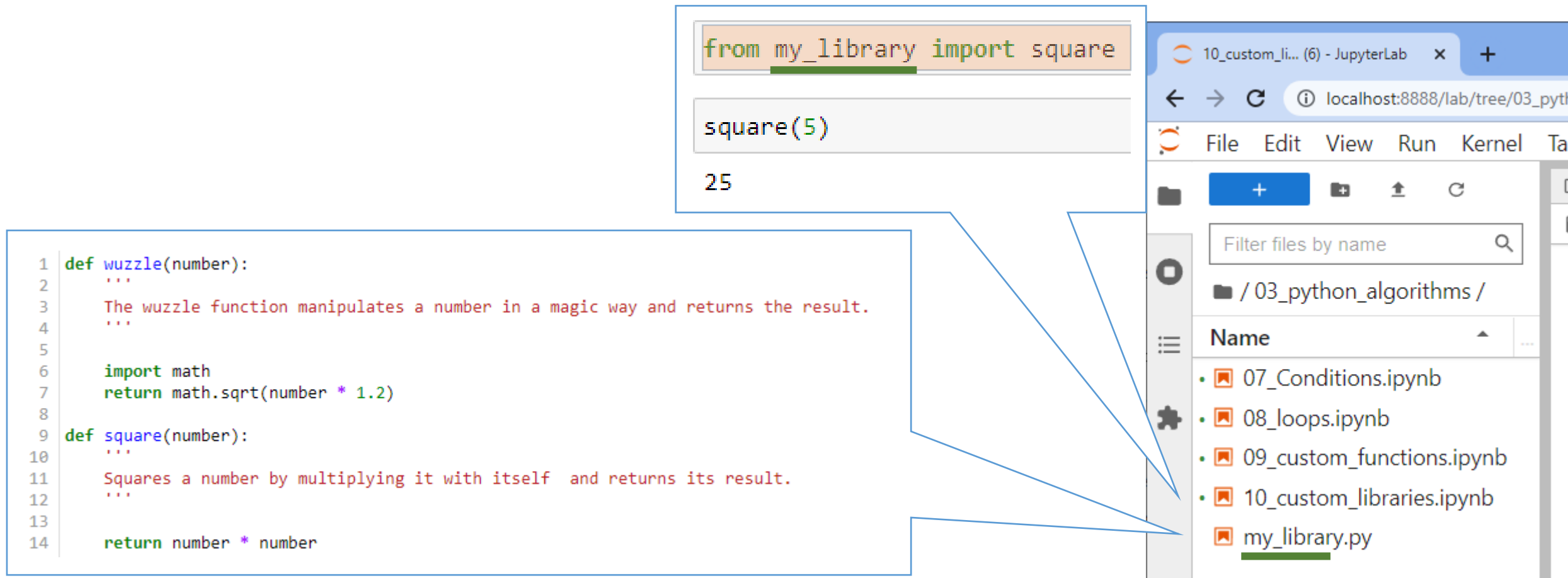
    # descriptive statistics
    mean_total_intensity = np.mean(table["area"] * table["mean_intensity"])

    return mean_total_intensity
```

- It's easier to get an overview about small functions
- It forces programmers to organise their macros

# Reusing code from custom modules

- For re-using functions between notebooks / projects, use libraries. -> Sustainability
- Simple python libraries are .py files containing multiple functions.
- The **import** statement allows you to import python files from the same folder.



```
1 def wuzzle(number):  
2     ...  
3     The wuzzle function manipulates a number in a magic way and returns the result.  
4     ...  
5  
6     import math  
7     return math.sqrt(number * 1.2)  
8  
9 def square(number):  
10     ...  
11     Squares a number by multiplying it with itself and returns its result.  
12     ...  
13  
14     return number * number
```

```
from my_library import square  
  
square(5)  
  
25
```

10\_custom\_li... (6) - JupyterLab

localhost:8888/lab/tree/03\_pyth

File Edit View Run Kernel Tal

Filter files by name

/ 03\_python\_algorithms /

Name

- 07\_Conditions.ipynb
- 08\_loops.ipynb
- 09\_custom\_functions.ipynb
- 10\_custom\_libraries.ipynb
- my\_library.py

- What does this program do?

```
image = imread("../data/blobs.tif")

# define a list of functions and a corresponding list of arguments
functions = [gaussian_blur, threshold_otsu, label]
argument_lists = [[.5], [], []]

# go through functions and argument lists pair-wise
for function, argument_list in zip(functions, argument_lists):
    # execute function with given arguments
    image = function(image, *argument_list)

result1 = image
imshow(result1)
```

- The same like that?

```
image = imread("../data/blobs.tif")

blurred = gaussian_blur(image, 5)
binary = threshold_otsu(blurred)
labels = label(binary)

result2 = labels

imshow(result2)
```

Yes

No

- There are two ways design software:
  - make it so simple that there are obviously no issues or
  - make it so complicated that there are no obvious issues.

(based loosely on Tony Hoare, "The Emperor's Old Clothes," CACM Feb. 1981)

- Take home: Keep it *short and simple*!

# Exercise: modularization

Go through the code from yesterday and identify a workflow that worked well and contained these steps:

- Image preprocessing
- Image segmentation

Copy & paste code from the notebooks you worked with yesterday into a new custom module `my_analysis.py`. Write a new notebook that demonstrates how to use the functions of your `my_analysis` module.

```
def my_workflow(input_image, ...):  
    ...  
    return label_image
```

