

Python Algorithms

Conditions, loops, functions and custom libraries

Robert Haase

Using material from Benoit Lombardot, Scientific Computing Facility, MPI CBG

April 2022

- Indexing also works with negative indices

```
data = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']
```

| | | | | | | | | | | |
|----------|----|----|----|----|----|----|----|----|----|---|
| Index: | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 | 0 |
| Content: | A | B | C | D | E | F | G | H | I | |

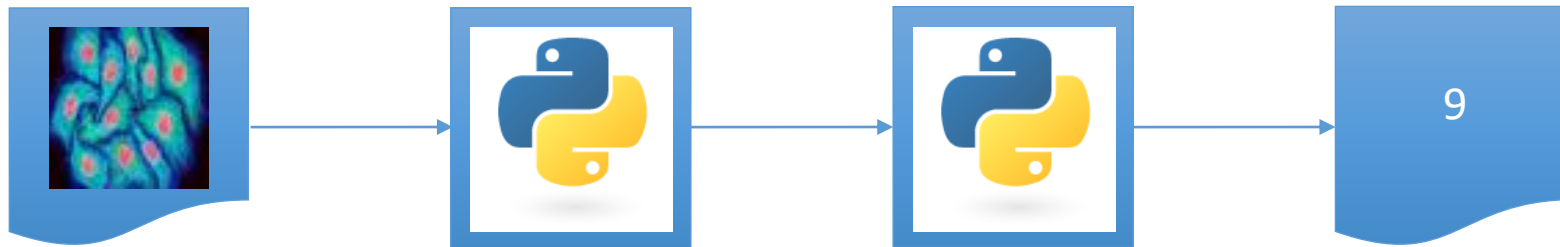
```
data[-2:]
```

```
['H', 'I']
```

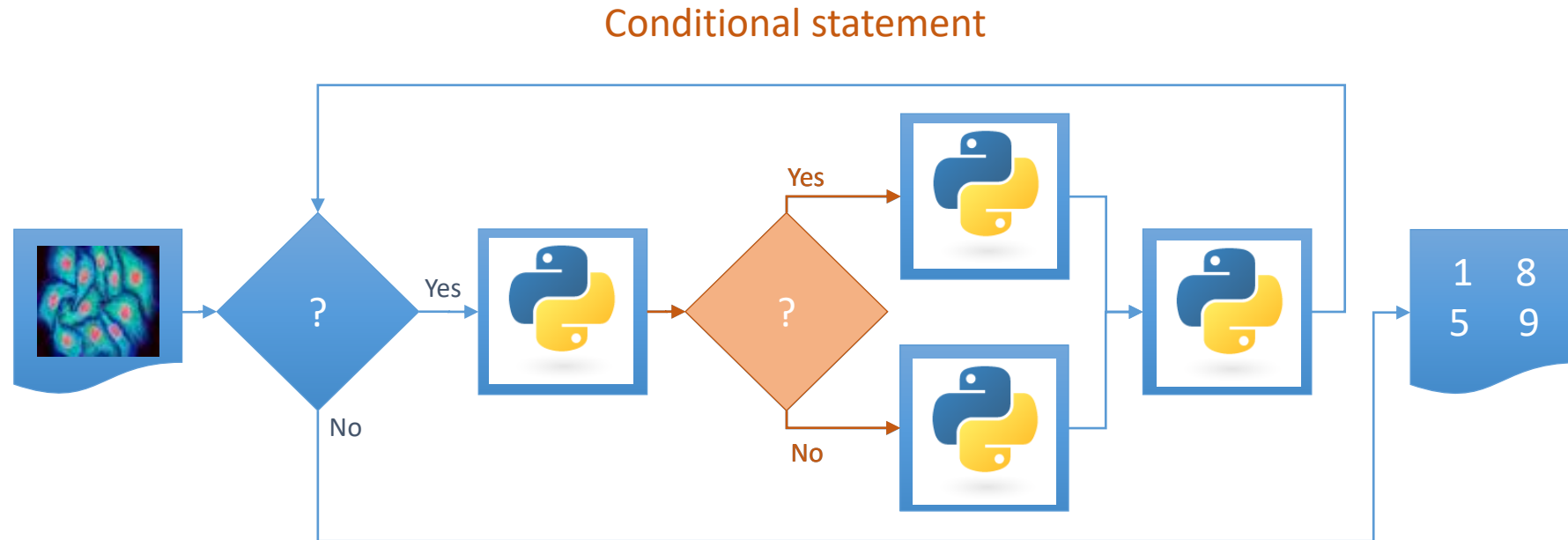
data[-2:0]
does not work

data[-2:]
works

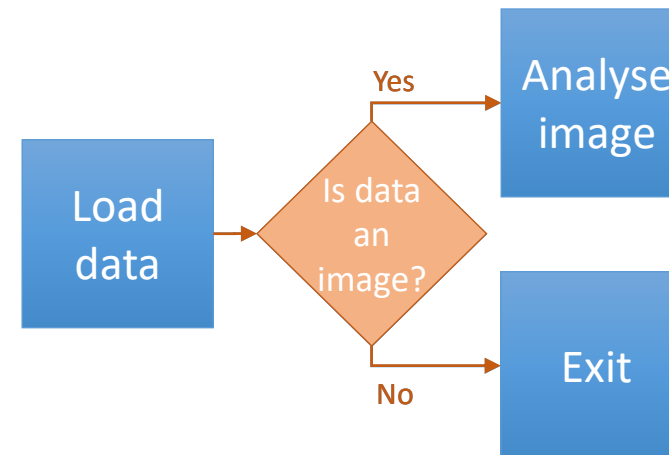
- Data science workflows *rarely* look like this



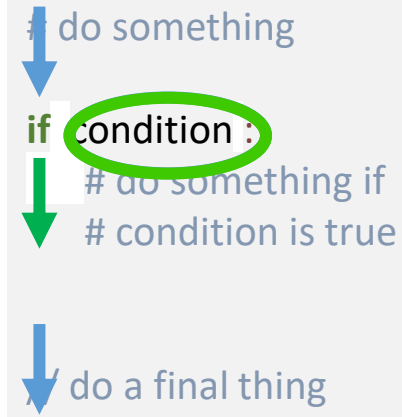
- Data science workflows *rather* look like this



- Conditional statements can be used to
 - Check if pre-requisites are met
 - Check if data has the right format
 - Check if processing results are within an expected range
 - Check for errors



- Depending on a condition, some lines of code are executed or not.

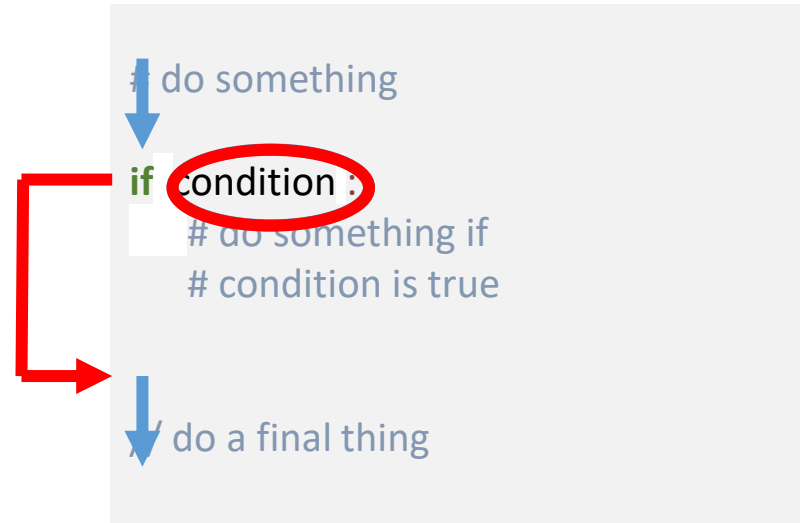


```
# do something
if condition :
    # do something if
    # condition is true

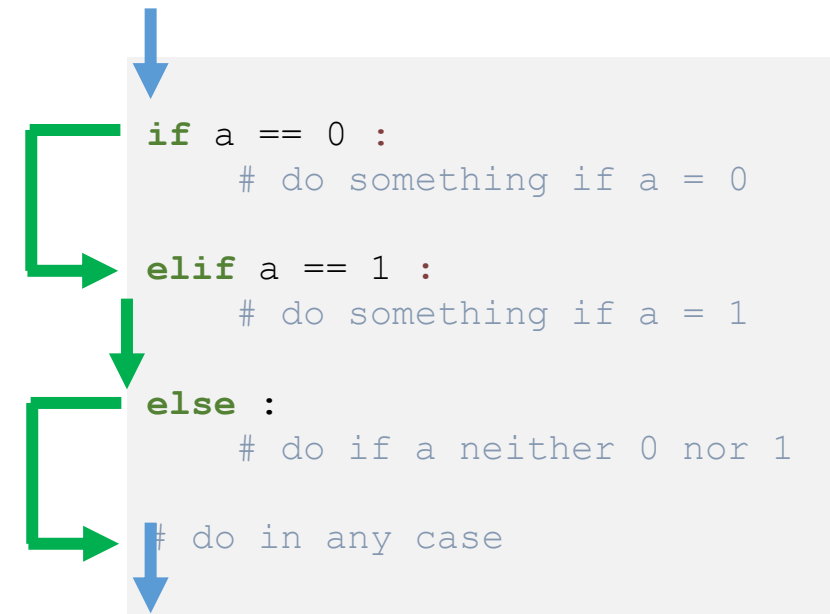
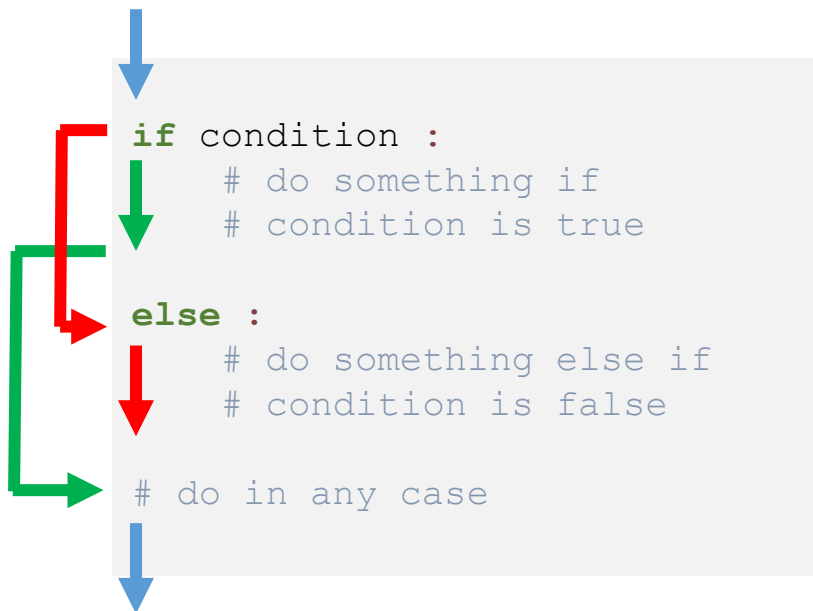
do a final thing
```

The diagram illustrates the execution flow of an if-statement. It starts with a blue arrow pointing down to the text "# do something". This is followed by a blue arrow pointing down to the text "if condition :". The word "condition" is circled in green. From "if condition :", a green arrow points down to the text "# do something if" and "# condition is true". Finally, a blue arrow points down to the text "do a final thing".

- Depending on a condition, some lines of code are executed or not.



- The **if** / **elif** / **else** statement allows to program alternatives.
- Depending on conditions, only one block is computed
- Indentation is used to mark where a block starts and ends.
- Indentation helps reading blocks,



- Comparison operators always have True (1) or False (0) as results.

```
# initialise program
quality = 99.5

# evaluate result
if quality > 99.9 :
    print("Everything is fine.")
else :
    print("We need to improve!")
```

```
In [1]: a = 4

if a = 5:
    print("Hello world")

File "<ipython-input-1-13fb587c9332>", line 3
    if a = 5:
        ^
SyntaxError: invalid syntax
```

Note: These are two equal signs!

| Operator | Description | Example |
|----------|-----------------------------------|---------|
| <, <= | smaller than, smaller or equal to | a < b |
| >, >= | greater than, greater or equal to | a > b |
| == | equal to | a == b |
| != | not equal to | a != 1 |

- Logic operators always take conditions as operands and result in a condition.
 - and
 - or
 - not
- Also combined conditions can be either True (1) or False (0).

```
# initialise program
quality = 99.9
age = 3

if quality >= 99.9 and age > 5 :
    print("The item is ok.")
```

```
# initialise program
quality = 99.9

if not quality < 99.9 :
    print("The item is ok.")
```

- Checking contents of lists can be done intuitively using the `in` statement

```
# initialise program
my_list = [1, 5, 7, 8]
item = 3

if item in list :
    print("The item is in the list.")
else :
    print("There is no", item, "in", my_list )
```

```
# initialise program
my_list = [1, 5, 7, 8]
item = 3

if item not in list :
    print("There is no", item, "in", my_list )
else :
    print("The item is in the list.")
```

Rules for readable code

- Every command belongs on its own line
- Insert empty lines to separate important processing steps
- Put spaces between operators and operands, because:

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

- Indent every conditional block (if/else) using the TAB key

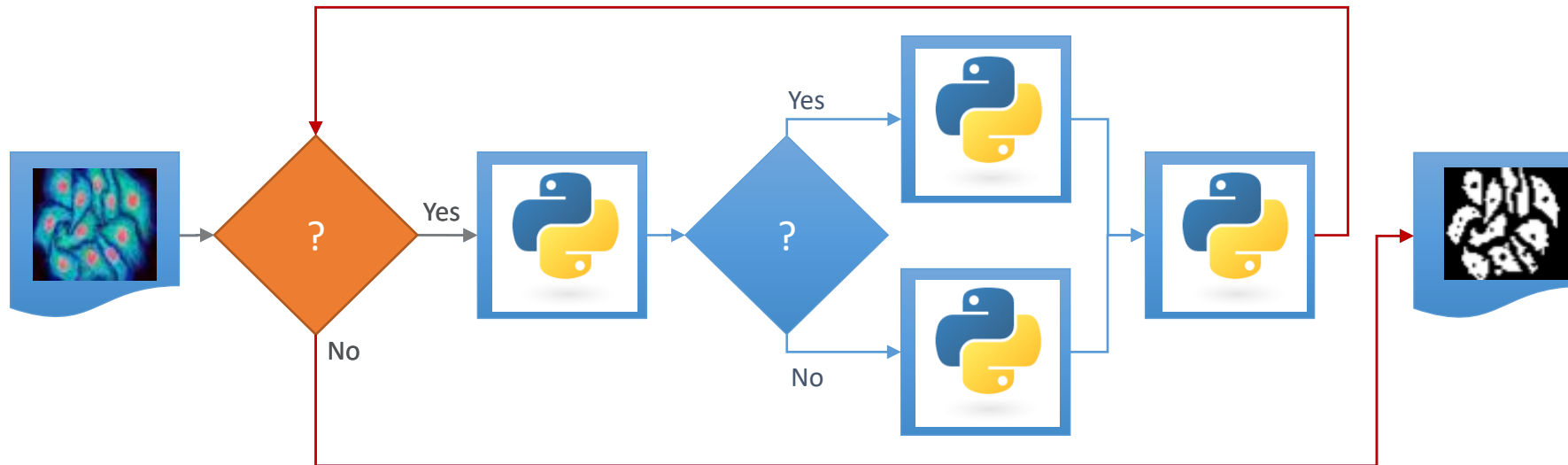
```
# initialise program
a = 5
b = 3
c = 8

# execute algorithm
d = (a + b) / c

# evaluate result
```

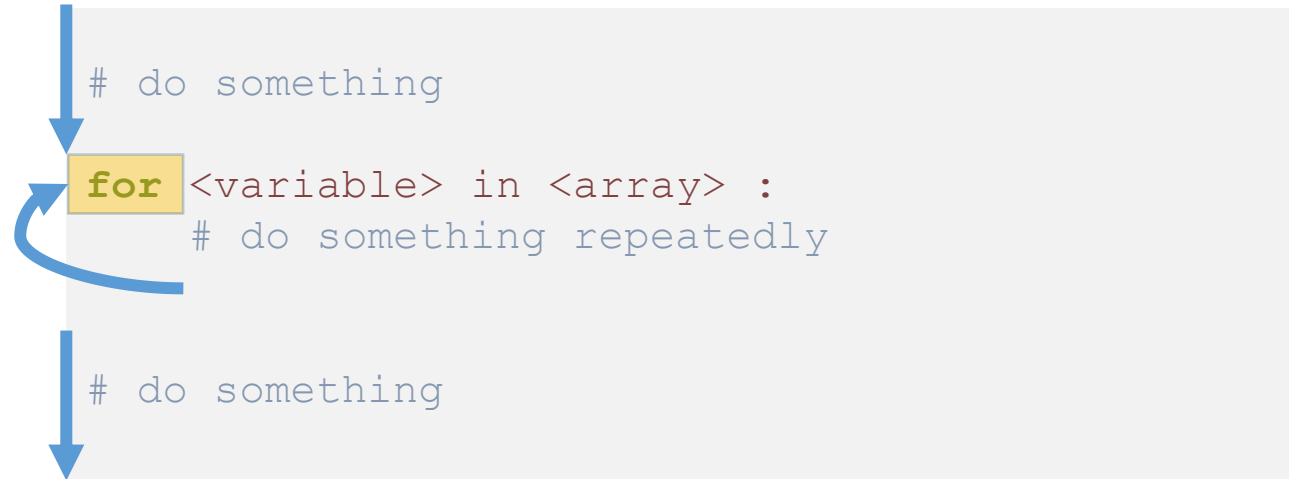
```
if a == 5 :
    print("Yin")
else :
    print("Yang")
```

- To repeat actions, you run code in loops



Loop statement

- The `for` statement allows us to execute some lines of code *for* several times,
- typically for all items in an array-like thing (list, tuples, images)



```
# do something  
for <variable> in <array> :  
    # do something repeatedly  
  
# do something
```

- Example list : `range(start, stop, step)`

```
▶ # for loops
  for i in range(0, 5):
    print(i)
```

0
1
2
3
4

```
▶ animal_set = ["Cat", "Dog", "Mouse"]

  for animal in animal_set:
    print(animal)
```

Cat
Dog
Mouse

- Indentation *means* combining operations to a block

```
❏ # for loops
for i in range(0, 5):
print(i)
```

```
File "<ipython-input-15-59c457ae0ac9>", line 3
  print(i)
    ^
```

IndentationError: expected an indented block

Don't forget to indent!

- Colon necessary

```
❏ # for loops
for i in range(0, 5)
  print(i)
```

```
File "<ipython-input-13-23157c0ed137>", line 2
  for i in range(0, 5)
    ^
```

SyntaxError: invalid syntax

Don't forget the colon!

- There is a long and a short way for creating arrays with numbers.

```
# we start with an empty list  
numbers = []
```

```
# and add elements
```

```
for i in range(0, 5):  
    numbers.append(i * 2)
```

```
print(numbers)
```

```
[0, 2, 4, 6, 8]
```

```
numbers = [i * 2 for i in range(0, 5)]
```

```
print(numbers)
```

```
[0, 2, 4, 6, 8]
```

- Also a combination with the if-statement is possible

```
# we start with an empty list
numbers = []

# and add elements
for i in range(0, 5):
    # check if the number is odd
    if i % 2:
        numbers.append(i * 2)

print(numbers)
```

[2, 6]

```
numbers = [i * 2 for i in range(0, 5) if i % 2]

print(numbers)
```

[2, 6]

- While loops keep executing indented code as long as a **condition** is met:

```
number = 1024  
  
while (number > 1):  
    number = number / 2  
    print(number)
```

Works the same as
with the **if** statement

```
512.0  
256.0  
128.0  
64.0  
32.0  
16.0  
8.0  
4.0  
2.0  
1.0
```

- Using the `break` statement, you can leave a loop

```
number = 1024


while (True):
    number = number / 2
    print(number)

    if number < 1:
        break;
```

512.0
256.0
128.0
64.0
32.0
16.0
8.0
4.0
2.0
1.0
0.5

- The `continue` statement allows to skip iterations

```
for i in range(0, 10):  
    if i >= 3 and i <= 6:  
        continue  
    print(i)
```



0
1
2
7
8
9

- In case repetitive tasks appear that cannot be handled in a loop, custom functions are the way to go.
- Functions allow to re-use code in different contexts.
- Indentation is crucial.
- Functions must be defined before called

- Definition

```
def sum_numbers(a, b):
```

name (parameters)

```
    result = a + b
```

body commands

```
    return result
```

return statement
(optional)

- Call

```
c = sum_numbers(4, 5)  
print(c)
```

9

```
sum_numbers(5, 6)
```

11

```
sum_numbers(3, 4)
```

- In case repetitive tasks appear that cannot be handled in a loop, custom functions are the way to go.
- Functions allow to re-use code in different contexts.
- Indentation is crucial.
- Functions must be defined before called

- Definition

```
def sum_numbers(a, b):  
    result = a + b  
    return result
```

- Call

```
c = sum_numbers(4, 5)  
print(c)
```

9

- Document your functions to keep track of what they do.
- Describe what the functions does and what the parameters are meant to be

```
def square(number):  
    '''  
    Squares a number by multiplying it with itself and returns its result.  
    '''  
  
    return number * number
```

- You can then later print the *documentation* if you can't recall how a function works.

```
square?
```

```
Signature: square(number)
```

```
Docstring: Squares a number by multiplying it with itself and returns its  
result.
```

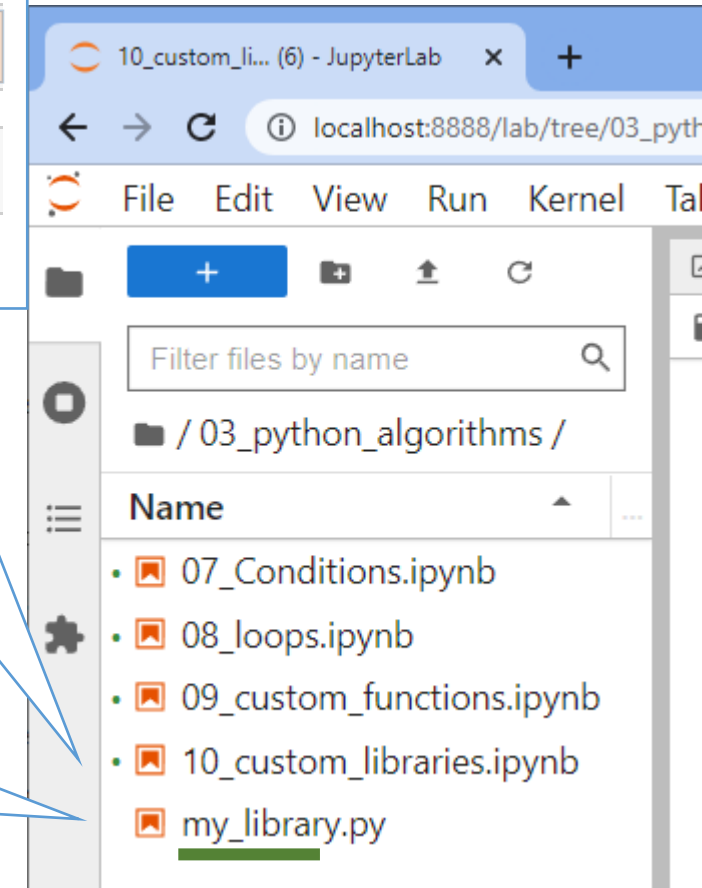

- For re-using functions between notebooks / projects, use libraries. -> Sustainability
- Simple python libraries are .py files containing multiple functions.
- The **import** statement allows you to import python files from the same folder.

```
from my_library import square
```

```
square(5)
```

25

```
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
```



Today, you learned

- Python
 - Conditions
 - Loops
 - Functions
 - Libraries
- How to write readable code.

Coming up next

- Image processing with Napari and Python
 - Filtering
 - Segmentation

