Python Algorithms
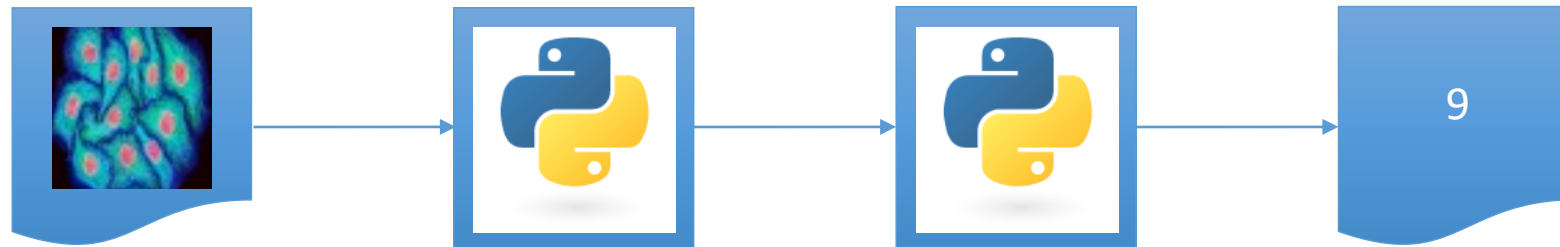## conditions, loops, functions

Till Korten, Robert Haase
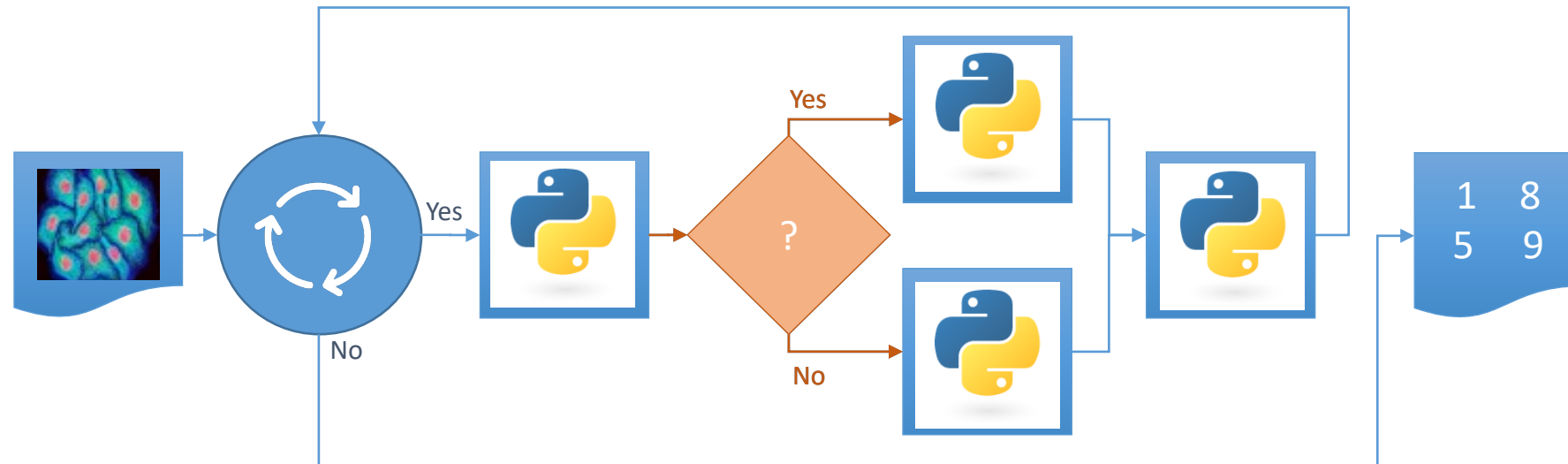
Using material from Benoit Lombardot, Scientific Computing Facility, MPI CBG
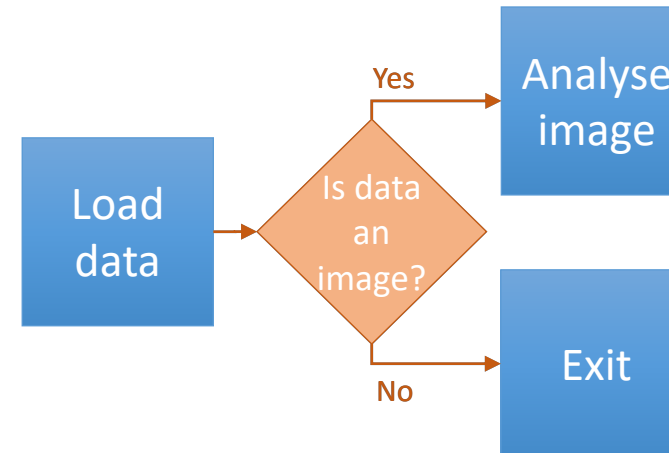
December 2022

- 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

Conditionals are implemented with the `if` statement

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

```
# load image

if s_image :
    # do something
    # with the image


save results
```
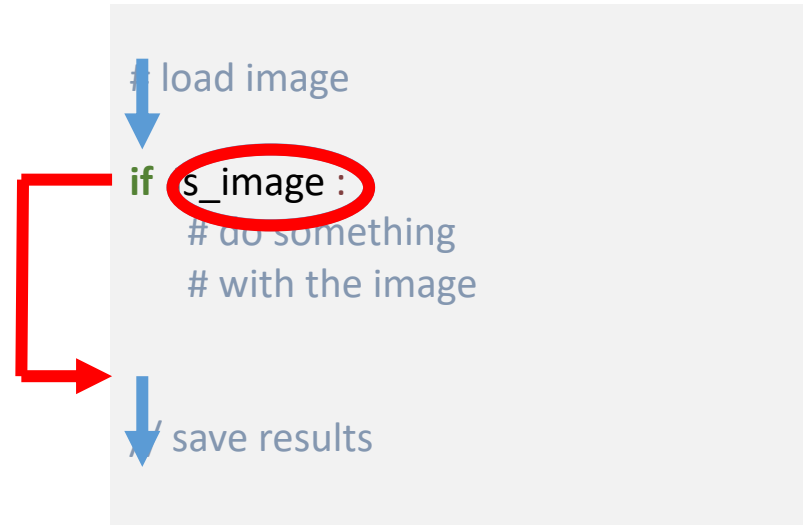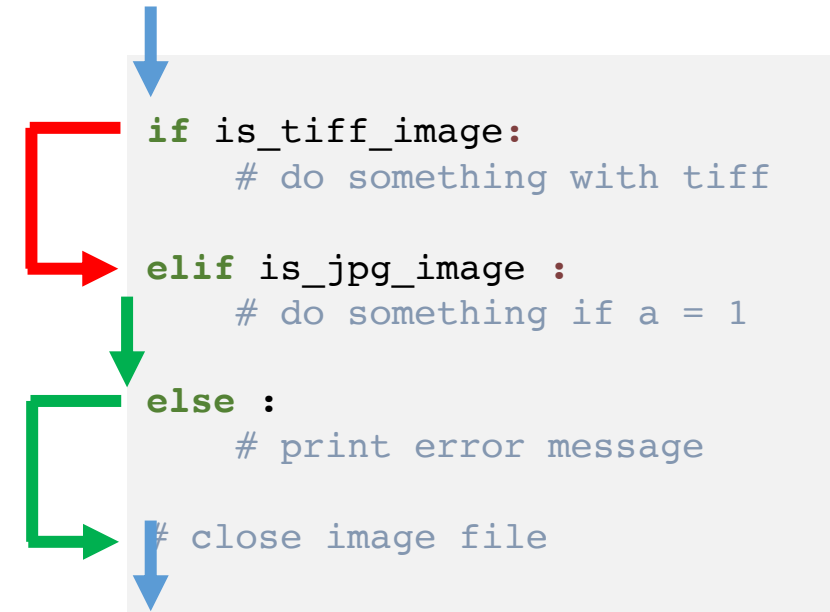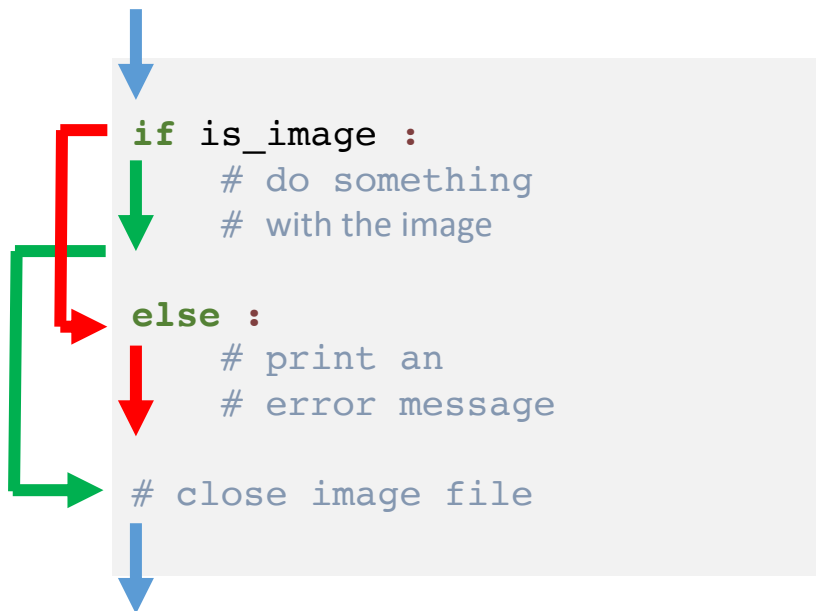
Conditionals are implemented with the `if` statement

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

```
# load image

if s_image :
    # do something
    # with the image


save results
```

`if` / `elif` / `else`: choose from several alternatives

- Depending on conditions, only one of several possible blocks is computed
- Indentation is used to mark where a block starts and ends.
- Indentation helps reading blocks,

```python
if is_image :
    # do something
    # with the image

else :
    # print an
    # error message

# close image file
```

```python
if is_tiff_image:
    # do something with tiff

elif is_jpg_image :
    # do something if a = 1

else :
    # print error message

# close image file
```

Comparison operators always have True (1) or False (0) as result

```python
# initialise program
image_size = 99.5

# evaluate quality
if image_size > 99.9 :
    print("Everything is fine.")
else :
    print("We need a larger image!")
```

 'We need a larger image!'

```
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
<, <=	less than, less than or equal to	a < b
>, >=	greater than, greater than or equal to	a > b
==	equal to	a == b
!=	not equal to	a != 1

Conditions can be combined with logic operators

- 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).

```python
# initialise program
image_size = 99.9
number_of_images = 3


if image_size >= 99.9 and number_of_images > 5 :
    print("The image is ok.")
```

```python
# initialise program
image_size = 99.9


if not image_size < 99.9 :
    print("The image is ok.")
```

'The image is ok.'

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

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

'There is no 3 in [1, 5, 7, 8]'

- Quite intuitive, isn't it?

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

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

'There is no 3 in [1, 5, 7, 8]'

Rules for readable code

- Every <u>command</u> belongs on its <u>own line</u>

- Insert <u>empty lines to separate</u> important processing steps

- Put <u>spaces</u> between operators and operands, because:

This is easier to read thanthat,orisnt'it?

- <u>Indent</u> every conditional block (if/else) using the TAB key
  - Python actually enforces this rule: Indentation *means* combining operations to a block

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

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

# evaluate result
```
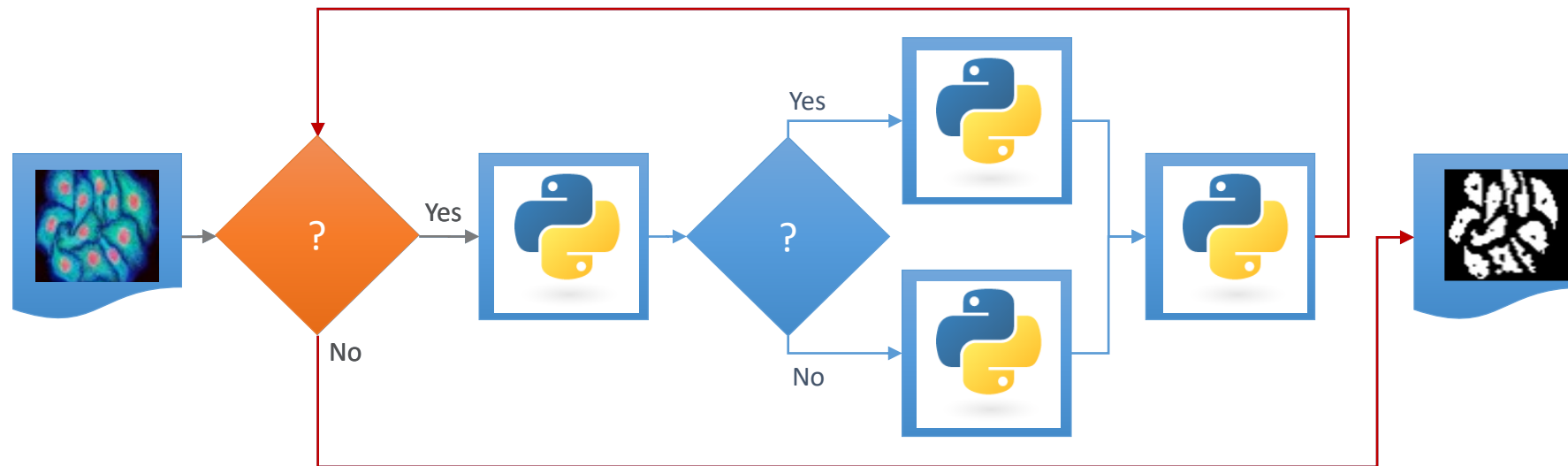
```python
if a == 5 :
a = 3
print("Yin")
else :
a = 1
print("Yang")
```

```
Cell In [2], line 3
    print("Yin")
    ^
IndentationError: expected an indented block
```

To repeat actions, you run code in loops

Loop statement

`for`: execute some lines of code *for* a number of times

- typically for all items in an array-like thing (lists, tuples, images)

```
# open array of time-lapse images

for <image> in <image array> :
    # process image


# save results
```

`for-in`: Loop over items of a list

- Example list :

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

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

```
Cat
Dog
Mouse
```

`range` creates numbers on the fly:

`range(start, stop, step)`

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

```
0
1
2
3
4
```

for-loop syntax pitfalls

- Indent the code within the `for` loop remember: indentation *means* combining operations to a block

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

> Don't forget to indent!

```
  File "<ipython-input-15-59c457ae0ac9>", line 3
    print(i)
          ^
IndentationError: expected an indented block
```

- Colon necessary

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

> Don't forget the colon!

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

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

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

# and add elements
for i in range(0, 5):
    numbers.append(i * 2)

print(numbers)
```

```python
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

```python
# 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]
```

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

print(numbers)
```

```
[2, 6]
```

```
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
```

```python
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
```

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

```
0
1
2
7
8
9
```

Functions

- 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.
- Defined using the `def` keyword
- Indentation is crucial.
- Functions must be defined before called

- Definition

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

name (parameters)

body commands

return statement (optional)

- Call

```
c = sum_numbers(4, 5)
print(c)
```
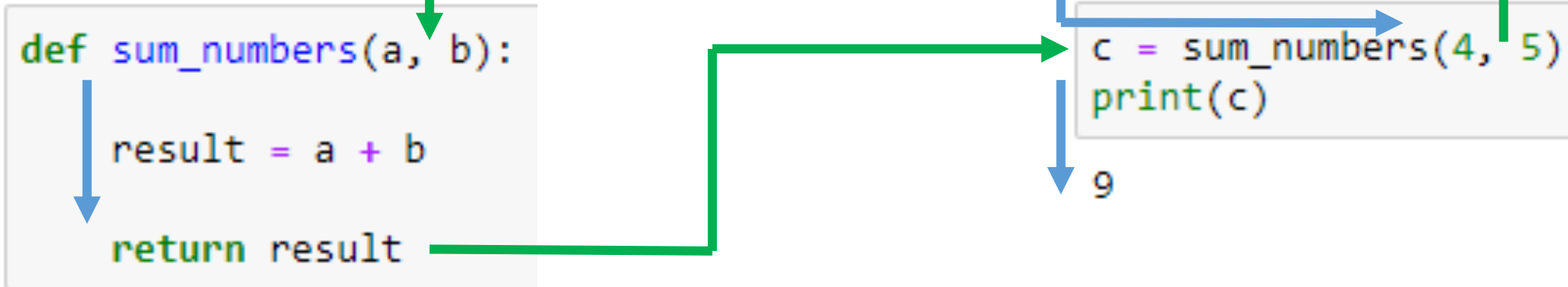
9

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

11

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

7

- 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.
- Defined using the `def` keyword
- Indentation is crucial.
- Functions must be defined before called

- Definition

- Call

```
def sum_numbers(a, b):

    result = a + b

    return result
```

```
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

```python
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* with a **?** 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.
```

- Hint: most integrated development environments (=coding software) provide automatisms to create a documentation template for your function. Look for *autodocstring* or similar.

Summary

Today, you learned

- Python
  - Conditions: `if` / `elif` / `else`
  - Loops: `for .. in` / `while` / `break` / `continue`
  - Functions: `def`