PoL Physics of Life TU Dresden

CENTER FOR SYSTEMS BIOLOGY DRESDEN

Python Algorithms

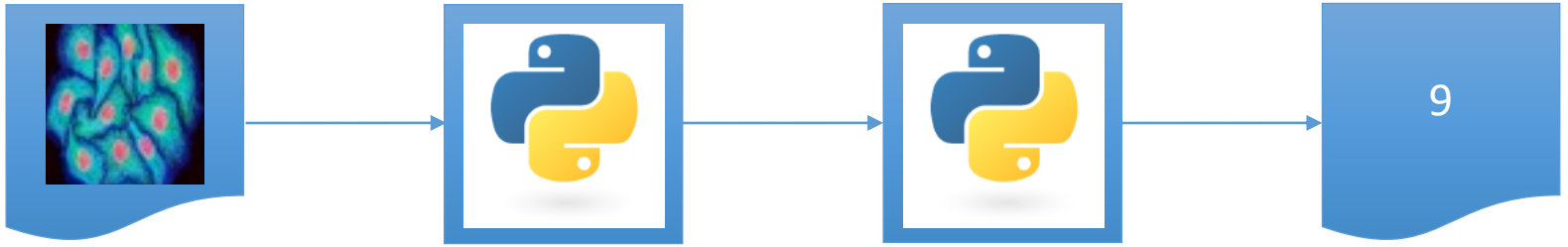
conditions, loops, functions

Till Korten, Robert Haase

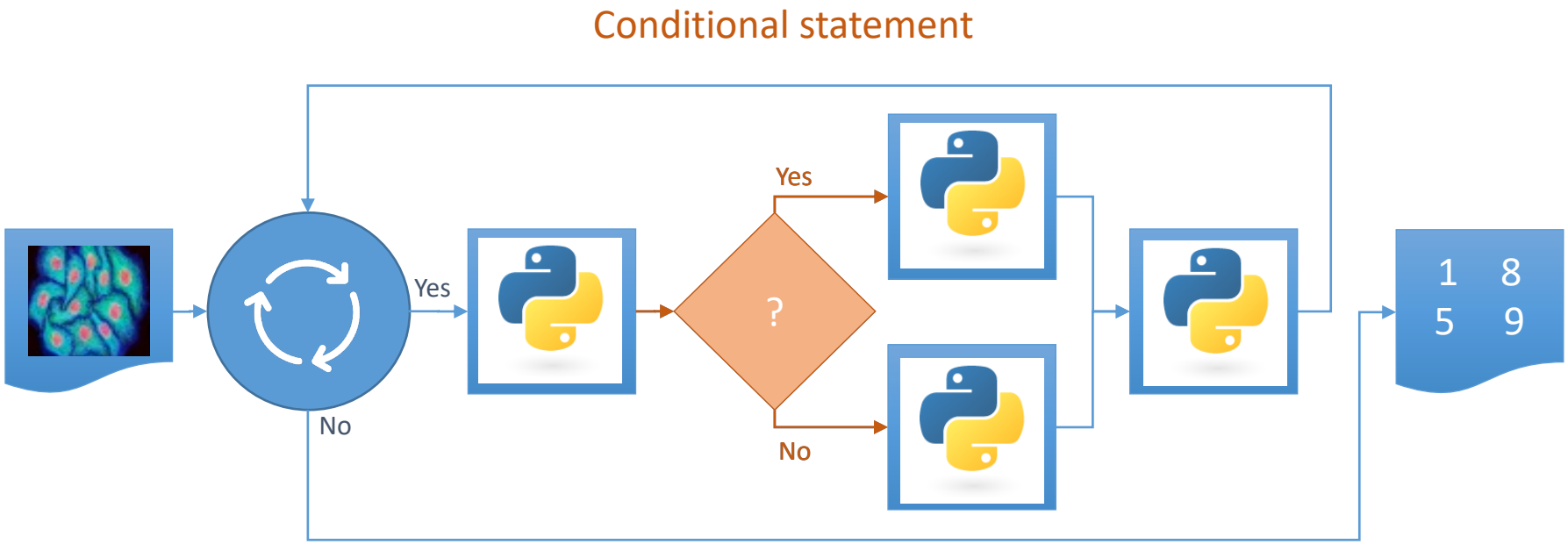
Using material from Benoit Lombardot, Scientific Computing Facility, MPI CBG

August 2023

Data science workflows *rarely* look like this

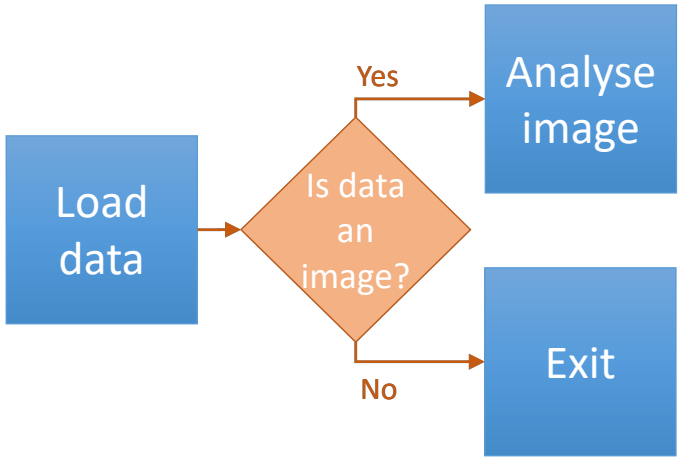


Data science workflows *rather* look like this



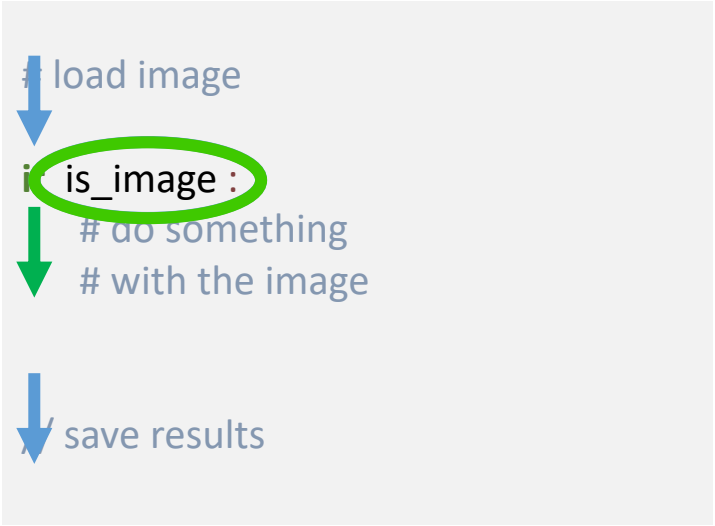
Example use cases for conditional statements

- 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



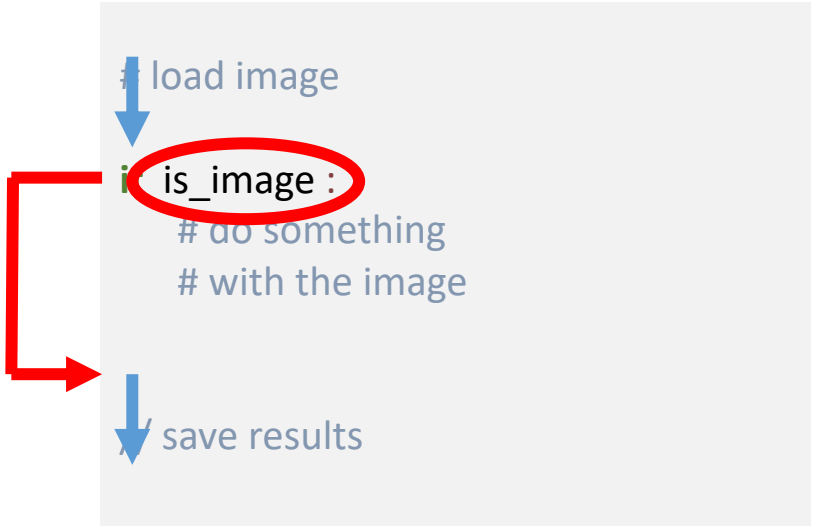
Conditionals are implemented with the `if` statement

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



Conditionals are implemented with the `if` statement

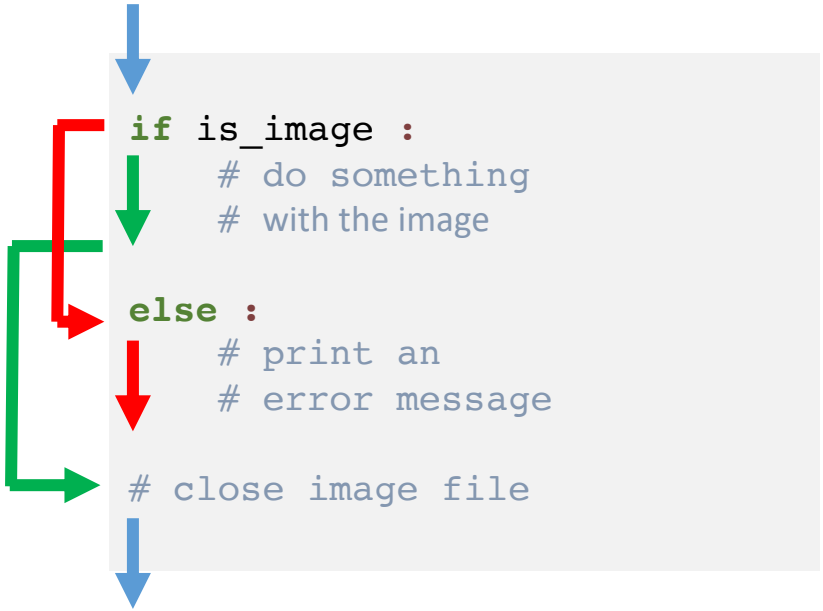
- Depending on a condition, some lines of code are executed or not.



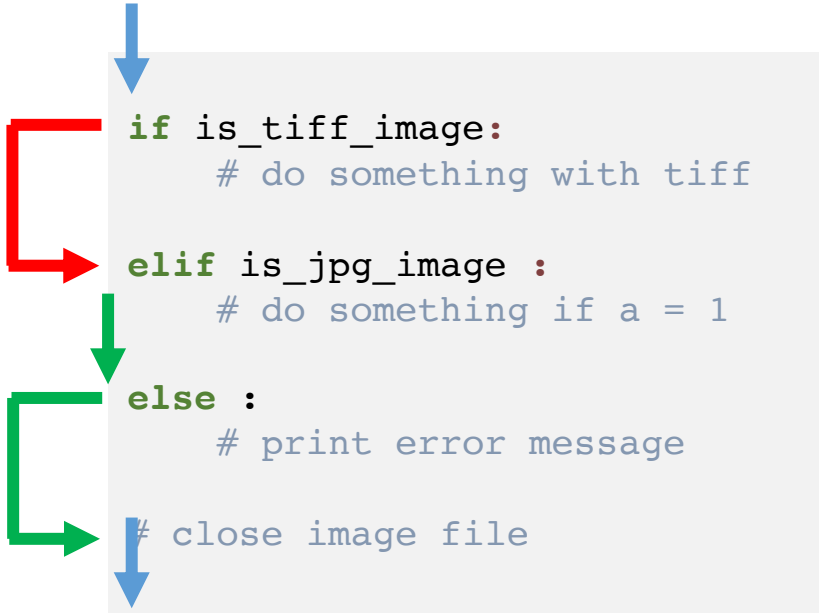
`if` / `elif` / `else`: choose from several alternatives

PoL Physics of Life TU Dresden

- 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,



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

else :
    # print an
    # error message

# close image file
```



```
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

PoL Physics of Life TU Dresden

```
# 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

PoL Physics of Life TU Dresden

- 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
image_size = 99.9
number_of_images = 3

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

```
# initialise program
image_size = 99.9

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

```
'The image is ok.'
```

The `in` statement: Checking contents of lists

```
# 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?

```
# 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 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 thanthat,orisnt'it?

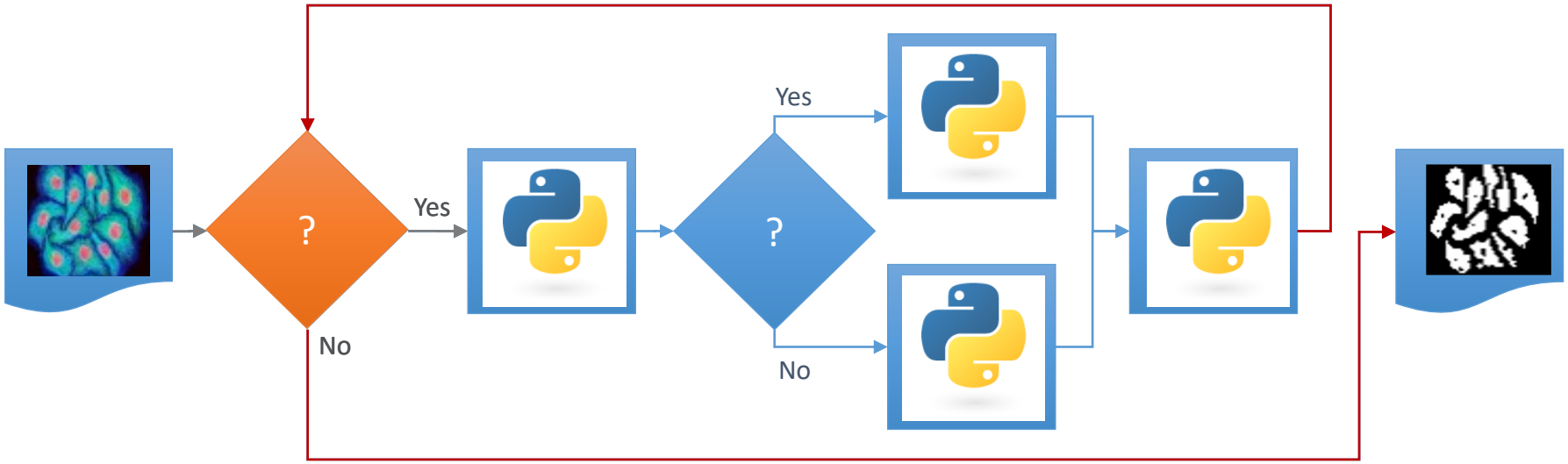
- Indent every conditional block (if/else) using the TAB key
  - Python actually enforces this rule: Indentation *means* combining operations to a block

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

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

# evaluate result
```

```
if a == 5 :
a = 3
print("Yin")
else :
a = 1
print("Yang")
```

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

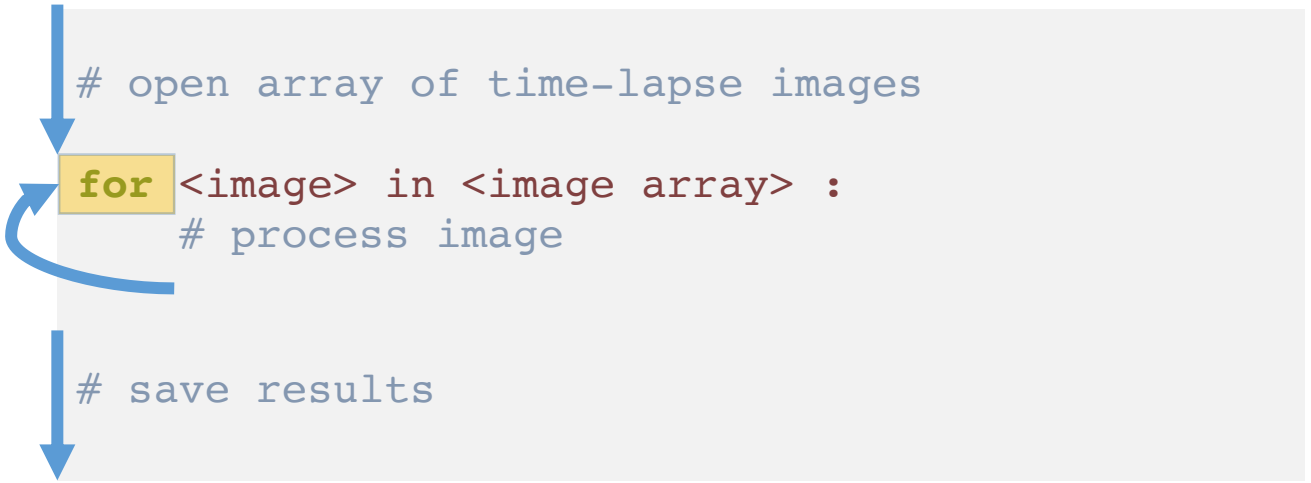
To repeat actions, you run code in **loops**



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)



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

- Example list :

```
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)`

```
# 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)
```

```
  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!

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
```

Functions run a block of code with one command

- 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
```

- Call

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

```
9
```

Keep it short and simple (KISS)

- Let's assume we want to write a function that grades student exams

```
def grade_student_exams(points_achieved: int, total_points_in_exam: int) -> int:
percentage = points_achieved / total_points_in_exam * 100
if percentage > 95:
    grade = 1
elif percentage > 80:
    grade = 2
elif percentage > 60:
    grade = 3
elif percentage > 50:
    grade = 4
else:
    grade = 5
return grade
```

Keep it short and simple (KISS)

- Now we want to extend that function to also grade pass/fail exams

```
def grade_student_exams(points_achieved: int, total_points_in_exam: int ,
pass_fail: bool = True) -> int:
percentage = points_achieved / total_points_in_exam * 100
if percentage > 95:
    grade = 1
elif percentage > 80:
    grade = 2
elif percentage > 60:
    grade = 3
elif percentage > 50:
    grade = 4
else:
    grade = 5
if pass_fail:
    if grade < 5:
        return True
    else:
        return False
else:
    return grade
```

This is rather messy:

It is not clear what the function returns

If `pass_fail` is `False`, we return an integer,

Otherwise a boolean.

Also, reading what the function does is difficult

Keep it short and simple (KISS)

- If we split this into two, we get two nice short and simple functions again

```
def grade_student_exams(points_achieved: int, total_points_in_exam: int) -> int:
percentage = points_achieved / total_points_in_exam * 100
if percentage > 95:
    grade = 1
elif percentage > 80:
    grade = 2
elif percentage > 60:
    grade = 3
elif percentage > 50:
    grade = 4
else:
    grade = 5
return grade

def grade_pass_fail_exam(points_achieved: int, total_points_in_exam: int) -> bool:
grade = grade_student_exams(points_achieved, total_points_in_exam)
if grade < 5:
    return True
else:
    return False
```

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* 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`