PoL Physics of Life TU Dresden

CENTER FOR SYSTEMS BIOLOGY DRESDEN

Python Algorithms

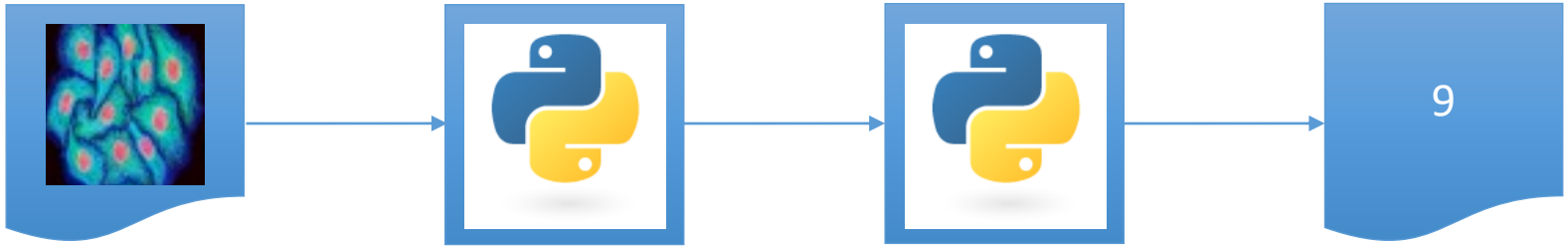
conditions, loops, functions

Till Korten, Robert Haase

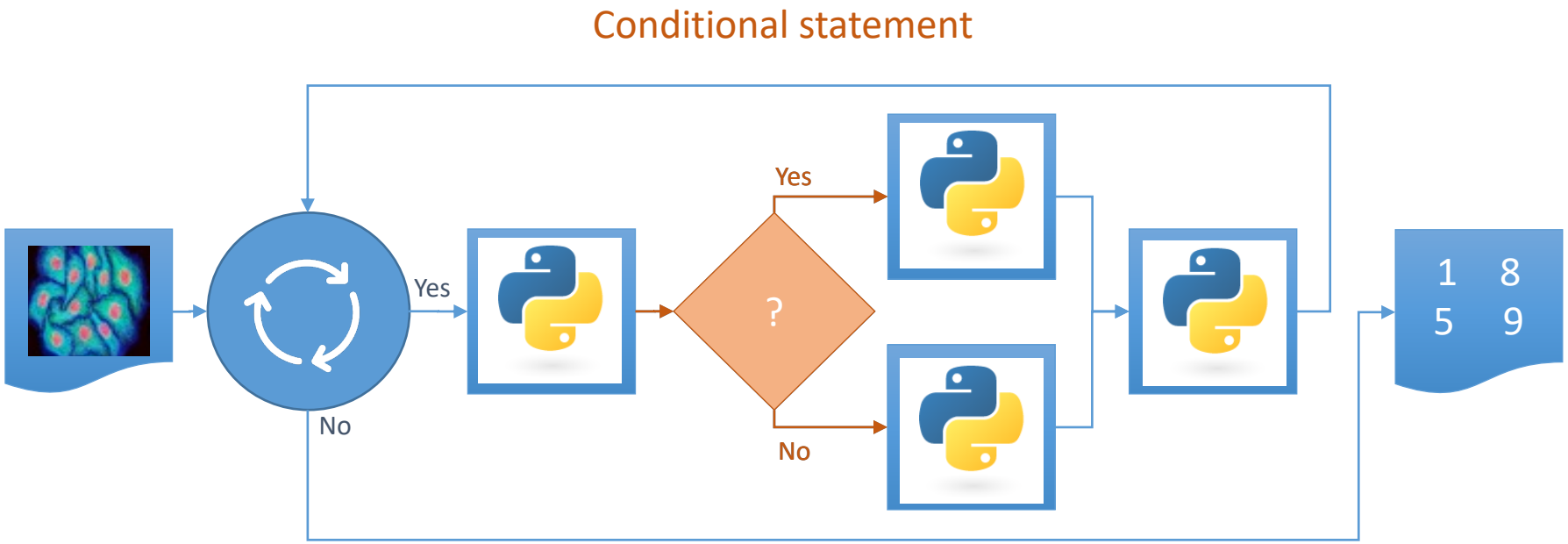
Using material from Benoit Lombardot, Scientific Computing Facility, MPI CBG

December 2022

Data science workflows *rarely* look like this

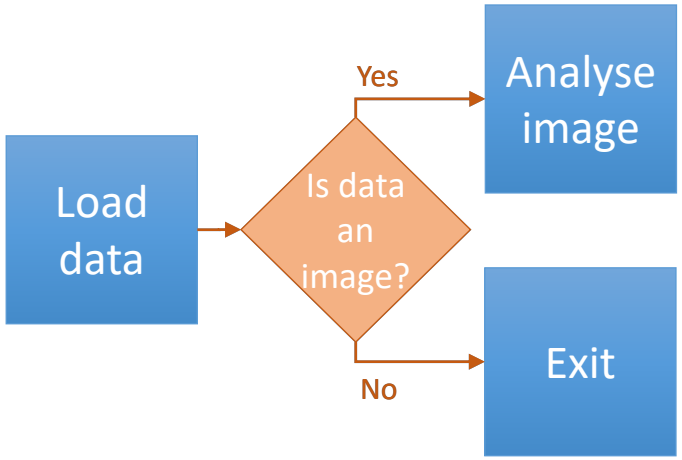


Data science workflows *rather* look like this



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.

```
# do something

if condition :
    # do something if
    # condition is true

# do a final thing
```

Conditionals are implemented with the `if` statement

- 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
```

`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,

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

else :
    # do something else if
    # condition is false

# do in any case
```

```
if a == 0 :
    # do something if a = 0

elif a == 1 :
    # do something if a = 1

else :
    # do if a neither 0 nor 1

# do in any case
```

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

```
# initialise program
quality = 99.5

# evaluate quality
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
<, <=	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).

```
# 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.")
```

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 )
```

- 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.")
```

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?

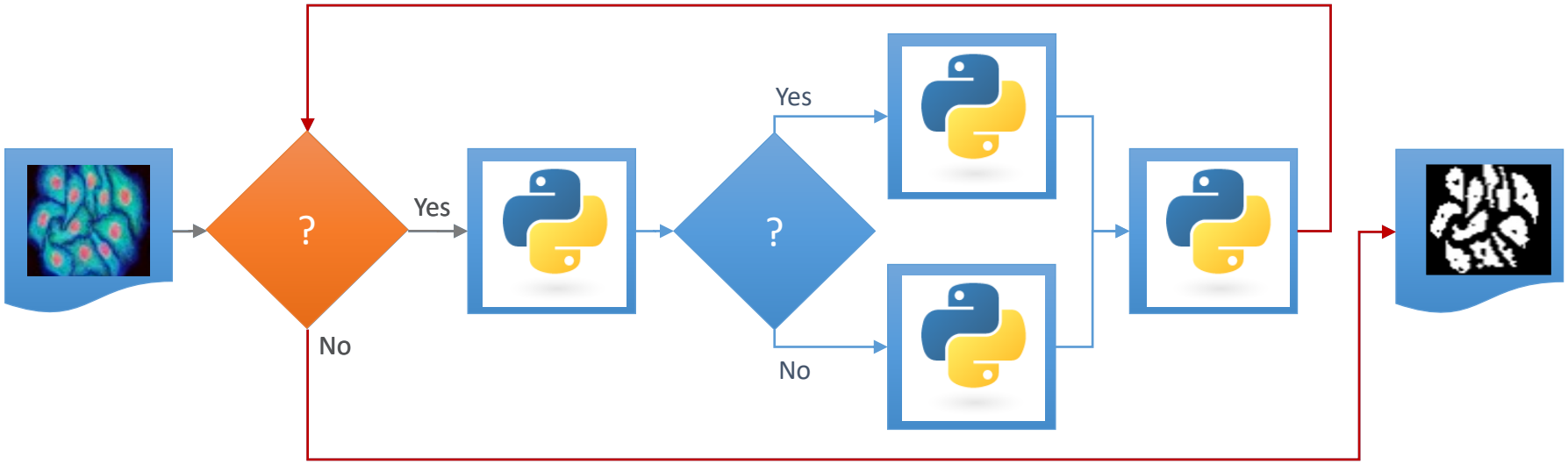
- Indent every conditional block (if/else) using the TAB key
  - Python actually enforces this rule: Indentation *means* combining operations to a block

```
# 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)

```
# do something

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

# do something
```

`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!

Generating arrays within `for`-loops

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

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

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

print(numbers)
```

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

print(numbers)
```

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

Generating arrays within `for`-loops

- 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 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
```

Skip iterations with the `continue` statement

```
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
```

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
```

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`