Teaching the Teacher: Python
Day 1 – Morning: »Python Basics 1«

Damian Trilling

d.c.trilling@uva.nl
@damian0604
www.damiantrilling.net

28 June 2021

Afdeling Communicatiewetenschap
Universiteit van Amsterdam

Today

The toolbox

- The role of software in CSS
- Python: A language, not a program

Datatypes

Functions and methods

Modifying lists and dictionaries

for, if/elif/else, try/except

Bonus: Python goodies

The toolbox

The role of software in CSS

Why program your own tool?

Vis (2013)

"Moreover, the tools we use can limit the range of questions that might be imagined, simply because they do not fit the affordances of the tool. Not many researchers themselves have the ability or access to other researchers who can build the required tools in line with any preferred enquiry. This then introduces serious limitations in terms of the scope of research that can be done."

Some considerations regarding the use of software in science

Assuming that science should be *transparent* and *reproducible by anyone*, we should

use tools that are

- platform-independent
- free (as in beer and as in speech, gratis and libre)
- which implies: open source

This ensures it can our research (a) can be reproduced by anyone, and that there is (b) no black box that no one can look inside. ⇒ ongoing open-science debate! (van Atteveldt et al., 2019)

Some considerations regarding the use of software in science

Assuming that science should be *transparent* and *reproducible by anyone*, we should

use tools that are

- platform-independent
- free (as in beer and as in speech, gratis and libre)
- which implies: open source

This ensures it can our research (a) can be reproduced by anyone, and that there is (b) no black box that no one can look inside. ⇒ ongoing open-science debate! (van Atteveldt et al., 2019)

Some considerations regarding the use of software in science

Assuming that science should be *transparent* and *reproducible by anyone*, we should

use tools that are

- platform-independent
- free (as in beer and as in speech, gratis and libre)
- which implies: open source

This ensures it can our research (a) can be reproduced by anyone, and that there is (b) no black box that no one can look inside. $\Rightarrow$ ongoing open-science debate! (van Atteveldt et al., 2019)

Why program your own tool?

Vis (2013)

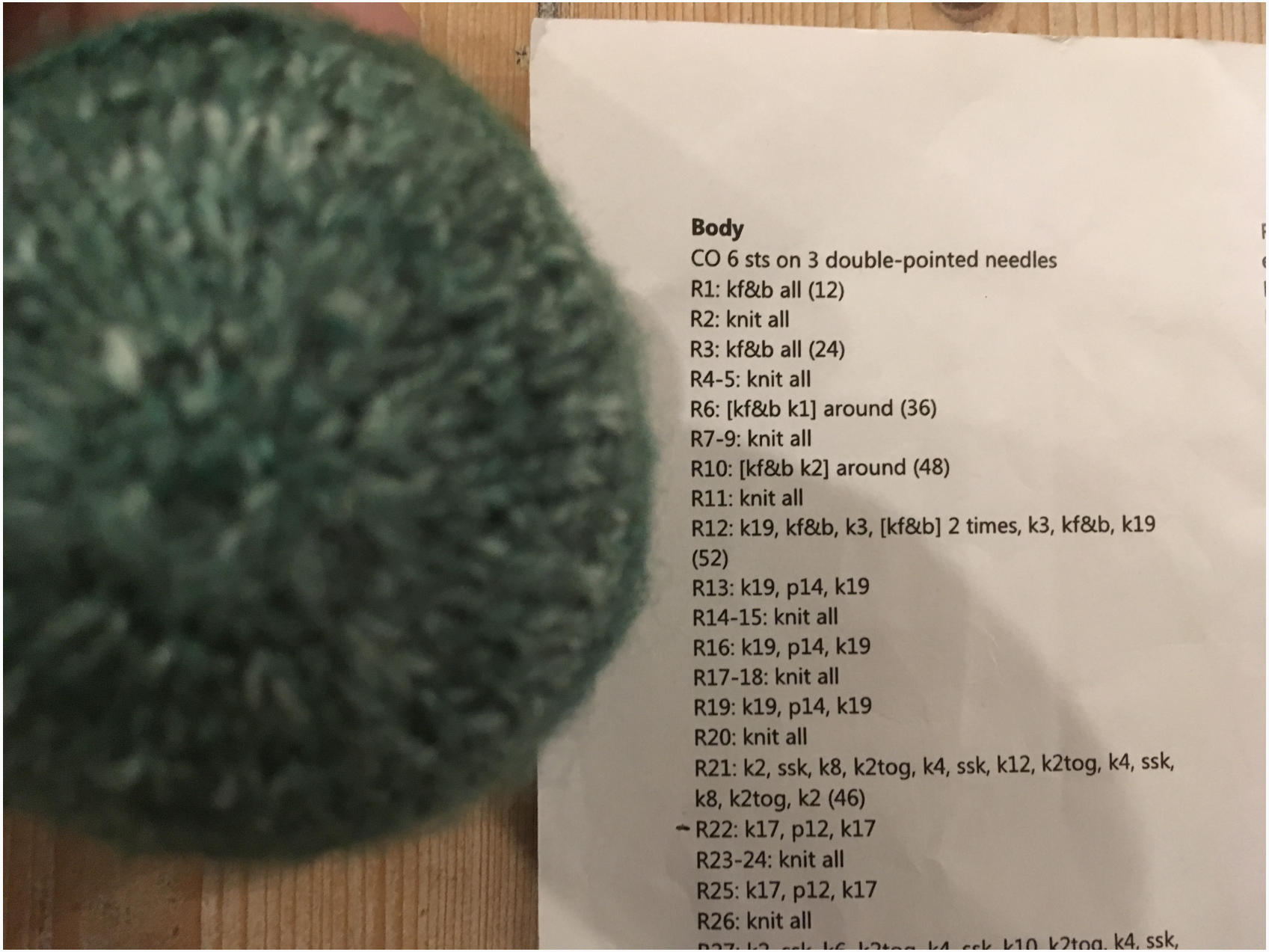
"[...] these [commercial] tools are often unsuitable for academic purposes because of their cost, along with the problematic 'black box' nature of many of these tools."

Mahrt and Scharkow (2013)

"[...] we should resist the temptation to let the opportunities and constraints of an application or platform determine the research question [...]"

The toolbox

Python: A language, not a program



An algorithm in a language that's a bit harder (I think) than Python

Python

What?

- A language, not a specific program
- Huge advantage: flexibility, portability
- One of *the* languages for data analysis. (The other one is R.)

  But Python is more flexible—the original version of Dropbox was written in Python. Some people say: R for numbers, Python for text and messy stuff.

Which version?

We use Python 3.

http://www.google.com or http://www.stackexchange.com still may show you some Python2-code, but that can easily be adapted. Most notable difference: In Python 2, you write `print "Hi"`, this has changed to `print ("Hi")`.

Python

What?

- A language, not a specific program
- Huge advantage: flexibility, portability
- One of *the* languages for data analysis. (The other one is R.)
  But Python is more flexible—the original version of Dropbox was written in Python. Some people say: R for numbers, Python for text and messy stuff.

Which version?

We use Python 3.

http://www.google.com or http://www.stackexchange.com still may show you some Python2-code, but that can easily be adapted. Most notable difference: In Python 2, you write `print "Hi"`, this has changed to `print ("Hi")`.

Python

What?

- A language, not a specific program
- Huge advantage: flexibility, portability
- One of *the* languages for data analysis. (The other one is R.)
  But Python is more flexible—the original version of Dropbox was written in Python. Some people say: R for numbers, Python for text and messy stuff.

Which version?

We use Python 3.

http://www.google.com or http://www.stackexchange.com still may show you some Python2-code, but that can easily be adapted. Most notable difference: In Python 2, you write `print "Hi"`, this has changed to `print ("Hi")`.

Let's run some Python code together!

Python lingo

Basic datatypes (variables)

int `37`

float `1.75`

bool `True`, `False`

string `"Alice"`

(**variable name** `firstname`)

"firstname" and firstname is not the same.

"5" and 5 is not the same.

But you can transform it: `int("5")` will return 5.

You cannot calculate `3 * "5"` (In fact, you can. It's "555").

But you can calculate `3 * int("5")`

Python lingo

Basic datatypes (variables)

int `37`

float `1.75`

bool `True`, `False`

string `"Alice"`

(**variable name** `firstname`)

"firstname" and firstname is not the same.

"5" and 5 is not the same.

But you can transform it: `int("5")` will return 5.

You cannot calculate `3 * "5"` (In fact, you can. It's "555").

But you can calculate `3 * int("5")`

Python lingo

Basic datatypes (variables)

int `37`

float `1.75`

bool `True`, `False`

string `"Alice"`

(**variable name** `firstname`)

"firstname" and firstname is not the same.

"5" and 5 is not the same.

But you can transform it: `int("5")` will return 5.

You cannot calculate `3 * "5"` (In fact, you can. It's `"555"`.

But you can calculate `3 * int("5")`

Python lingo

More advanced datatypes

```
list firstnames = ['Alice','Bob','Cecile']
     lastnames = ['Garcia','Lee','Miller']

list ages = [18,22,45]

dict agedict = {'Alice': 18, 'Bob': 22,
     'Cecile': 45}
```

Note that the elements of a list, the keys of a dict, and the values of a dict can have any* datatype! (You can even mix them, but it's better to be consistent!)

*Well, keys cannot be mutable → see book

Python lingo

More advanced datatypes

```
list firstnames = ['Alice','Bob','Cecile']
     lastnames = ['Garcia','Lee','Miller']

list ages = [18,22,45]

dict agedict = {'Alice': 18, 'Bob': 22,
     'Cecile': 45}
```

Note that the elements of a list, the keys of a dict, and the values of a dict can have any* datatype! (You can even mix them, but it's better to be consistent!)

*Well, keys cannot be mutable → see book

Python lingo

More advanced datatypes

```
list firstnames = ['Alice','Bob','Cecile']
     lastnames = ['Garcia','Lee','Miller']
list ages = [18,22,45]
dict agedict = {'Alice': 18, 'Bob': 22,
     'Cecile': 45}
```

Note that the elements of a list, the keys of a dict, and the values of a dict can have any* datatype! (You can even mix them, but it's better to be consistent!)

*Well, keys cannot be mutable → see book

Python lingo

More advanced datatypes

```
list firstnames = ['Alice','Bob','Cecile']
     lastnames = ['Garcia','Lee','Miller']
list ages = [18,22,45]
dict agedict = {'Alice':  18, 'Bob':  22,
     'Cecile':  45}
```

Note that the elements of a list, the keys of a dict, and the values of a dict can have any* datatype! (You can even mix them, but it's better to be consistent!)

*Well, keys cannot be mutable → see book

Python lingo

More advanced datatypes

```
list firstnames = ['Alice','Bob','Cecile']
     lastnames = ['Garcia','Lee','Miller']
list ages = [18,22,45]
dict agedict = {'Alice':  18, 'Bob':  22,
     'Cecile':  45}
```

Note that the elements of a list, the keys of a dict, and the values of a dict can have any* datatype! (You can even mix them, but it's better to be consistent!)

*Well, keys cannot be mutable → see book

Python lingo

Retrieving specific items

list `firstnames[0]` gives you the first entry

`firstnames[-2]` gives you the one-but-last entry

`firstnames[:2]` gives you entries 0 and 1

`firstnames[1:3]` gives you entries 1 and 2

`firstnames[1:]` gives you entries 1 until the end

dict `agedict["Alice"]` gives you 18

Python lingo

Retrieving specific items

list `firstnames[0]` gives you the first entry

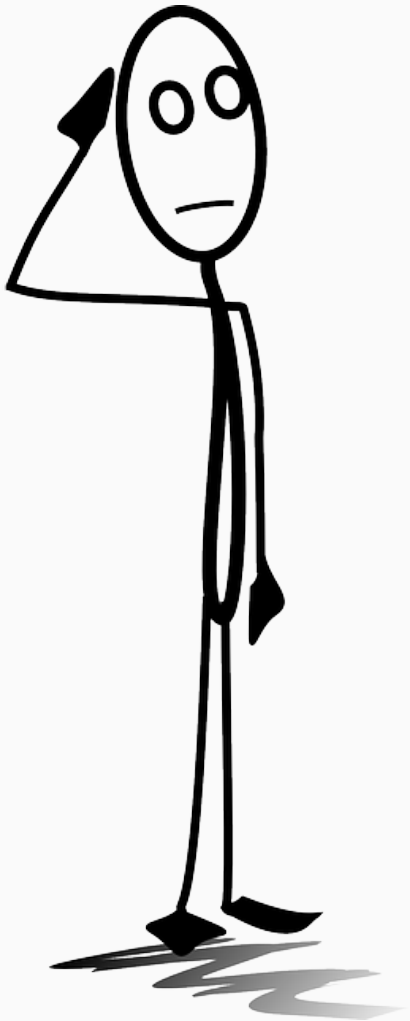
`firstnames[-2]` gives you the one-but-last entry

`firstnames[:2]` gives you entries 0 and 1

`firstnames[1:3]` gives you entries 1 and 2

`firstnames[1:]` gives you entries 1 until the end

dict `agedict["Alice"]` gives you 18



Think of at least two different ways of storing data about some fictious persons (first name, last name, age, phone number, . . .) using lists and/or dictionaries. What are the pros and cons?

Python lingo

Less frequent, but still useful datatypes

set	A collection in which each item is unique: `{1,2,3}`
tuple	Like a list, but *immutable*: `(1,2,2,2,3)`
defaultdict	A dict that does not raise an error but returns the "empty" value of its datatype (0 for int, "" for str) if you try access a non-existing key (great for storing results and counting things!)
np.array	A list-like datatype provided by the `numpy` package optimized for efficient mathematical operations.
...	...

You will come across more later

Python lingo

Functions

functions Take an input and return something else

`int(32.43)` returns the integer 32. `len("Hello")` returns the integer 5.

methods are similar to functions, but directly associated with an object. `"SCREAM".lower()` returns the string `"scream"`

Both functions and methods end with `()`. Between the `()`, *arguments* can (sometimes have to) be supplied.

Python lingo

Functions

functions Take an input and return something else `int(32.43)` returns the integer 32. `len("Hello")` returns the integer 5.

methods are similar to functions, but directly associated with an object. `"SCREAM".lower()` returns the string `"scream"`

Both functions and methods end with `()`. Between the `()`, *arguments* can (sometimes have to) be supplied.

Python lingo

Functions

functions Take an input and return something else `int(32.43)` returns the integer 32. `len("Hello")` returns the integer 5.

methods are similar to functions, but directly associated with an object. `"SCREAM".lower()` returns the string "scream"

Both functions and methods end with (). Between the (), *arguments* can (sometimes have to) be supplied.

Python lingo

Functions

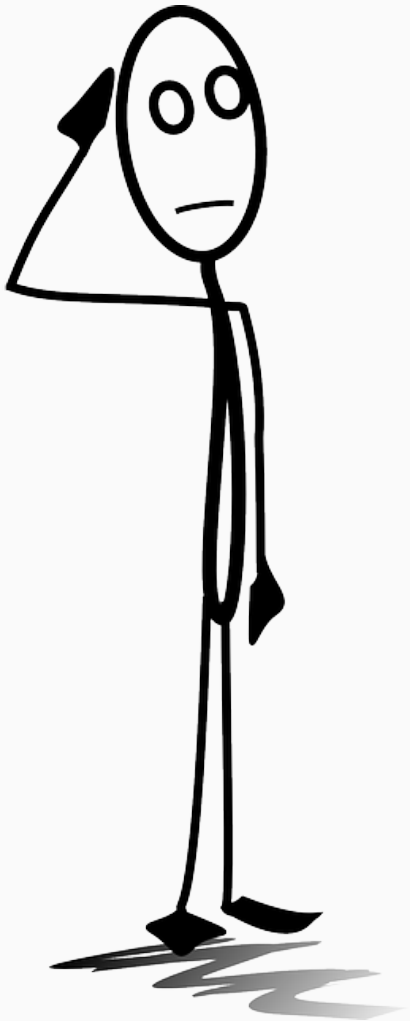
functions Take an input and return something else `int(32.43)` returns the integer 32. `len("Hello")` returns the integer 5.

methods are similar to functions, but directly associated with an object. `"SCREAM".lower()` returns the string "scream"

Both functions and methods end with `()`. Between the `()`, *arguments* can (sometimes have to) be supplied.

Some functions

```
1  len(x)       # returns the length of x
2  y = len(x)   # assign the value returned by len(x) to y
3  print(len(x)) # print the value returned by len(x)
4  print(y)     # print y
5  int(x)       # convert x to an integer
6  str(x)       # convert x to a string
7  sum(x)       # get the sum of x
```



How could you print the mean (average) of a list of integers using the functions on the previous slide?

Some methods

Some string methods

```
1 mystring = "Hi! How are you?"
2 mystring.lower() # return lowercased string (doesn't change original!)
3 mylowercasedstring = mystring.lower() # save to a new variable
4 mystring = mystring.lower() # or override the old one
5 mystring.upper() # uppercase
6 mystring.split() # Splits on spaces and returns a list ['Hi!', 'How', '
      are', 'you?']
```

We'll look into some list methods later.

⇒ **You can use TAB-completion in Jupyter to see all methods (and properties) of an object!**

Writing own functions

You can write an own function:

```
1  def addone(x):
2      y = x + 1
3      return y
```

Functions take some input ("argument") (in this example, we called it `x`) and *return* some result.

Thus, running

```
1  addone(5)
```

returns 6 .

Writing own functions

Attention, R users! (maybe obvious for others?)

You *cannot** apply the function that we just created on a whole list – after all, it takes an int, not a list as input.

(wait a sec foruntil we cover for loops later today, but this is how you'd do it (by calling the function for each element in the list separately):):

```
1 mynumbers = [5, 3, 2, 4]
2 results = [addone(e) for e in mynumbers]
```

* Technically speaking, you could do this by wrapping the `map` function around your own function, but that's not considered "pythonic". Don't do it ;-)

Modifying lists

Let's use one of our first methods! Each *list* has a method `.append()`:

Appending to a list

```
1 mijnlijst = ["element 1", "element 2"]
2 anotherone = "element 3" # note that this is a string, not a list!
3 mijnlijst.append(anotherone)
4 print(mijnlijst)
```

gives you:

```
1 ["element 1", "element 2", "element 3"]
```

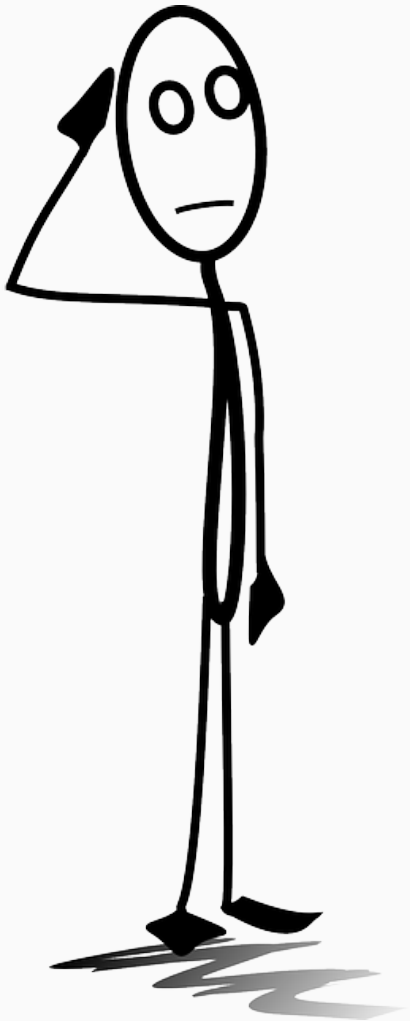
Modifying lists

Merging two lists (= extending)

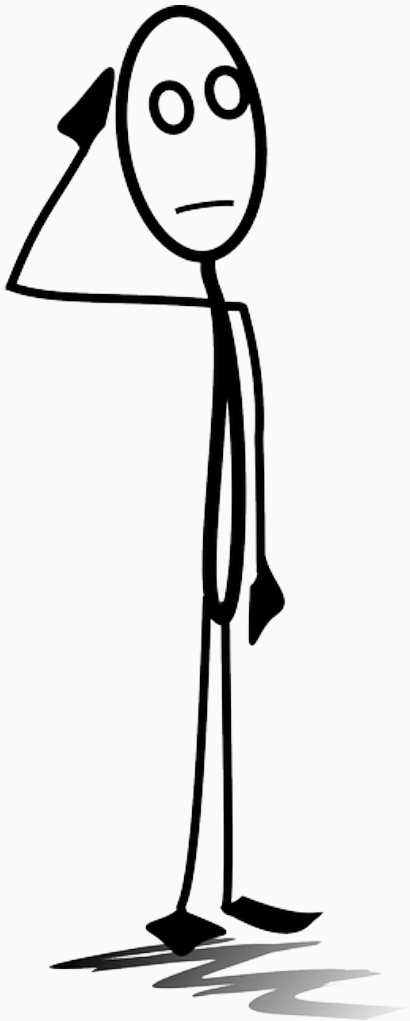
```
1 mijnlijst = ["element 1", "element 2"]
2 anotherone = ["element 3", "element 4"]
3 mijnlist.extend(anotherone)
4 print(mijnlijst)
```

gives you:

```
1 ["element 1", "element 2", "element 3", "element 4]
```



What would have happened if we had used `.append()` *instead of* `.extend()`*?*



Why do you think that the Python developers implemented `.append()` *and* `.extend()` *as methods of a list and not as functions?*

Modifying dicts

Adding a key to a dict (or changing the value of an existing key)

```
1 mydict = {"whatever": 42, "something": 11}
2 mydict["somethingelse"] = 76
3 print(mydict)
```

gives you:

```
1 {'whatever': 42, 'somethingelse': 76, 'something': 11}
```

If a key already exists, its value is simply replaced.

How can we structure our program?

If we want to *repeat* a block of code, exectute a block of code only *under specific conditions*, or more generally want to structure our code, we use *indention*.

Indention: The Python way of structuring your program

- Your program is structured by TABs or SPACEs.
- Jupyter (or your IDE) handles (guesses) this for you, but make sure to not interfere and not to mix TABs or SPACEs!
- Default: four spaces per level of indention.

Indention

Structure

A first example of an indented block – in this case, we want to *repeat* this block:

```
1  agedict = {'Zeus': None, 'Denis': 96, 'Alice': 18, 'Rebecca': 20 , 'Bob
       ': 22, 'Cecile': 45}
2
3  myfriends = ['Alice','Bob','Cecile']
4
5  print ("The names and ages of my friends:")
6  for buddy in myfriends:
7    print (f"My friend {buddy} is {agedict[buddy]} years old")
```

Output:

```
1  My friend Alice is 18 years old
2  My friend Bob is 22 years old
3  My friend Cecile is 45 years old
```

What happened here?

```
1  for buddy in myfriends:
2      print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The for loop

1. Take the first element from `myfriends` and call it `buddy` (like `buddy = myfriends[0]`) (line 1)
2. Execute the indented block (line 2, but could be more lines)
3. Go back to line 1, take next element (like `buddy = myfriends[1]`)
4. Execture the indented block ...
5. ... repeat until no elements are left ...

The f-string (*formatted* string)

If you prepend a string with an `f`, you can use curly brackets `{}` to insert the value of a variable

What happened here?

```
for buddy in myfriends:
    print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a :

More in general, the : + indention indicates that

- the block is to be executed repeatedly (for statement) – e.g., for each element from a list, or until a condition is reached (while statement)
- the block is only to be executed under specific conditions (if, elif, and else statements)
- an alternative block should be executed if an error occurs in the block (try and except statements)
- a file is opened, but should be closed again after the block has been executed (with statement)

What happened here?

```
1  for buddy in myfriends:
2      print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a :

More in general, the : + indention indicates that

- the block is to be executed repeatedly (`for` statement) – e.g., for each element from a list, or until a condition is reached (`while` statement)
- the block is only to be executed under specific conditions (`if`, `elif`, and `else` statements)
- an alternative block should be executed if an error occurs in the block (`try` and `except` statements)
- a file is opened, but should be closed again after the block has been executed (`with` statement)

What happened here?

```
1  for buddy in myfriends:
2      print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a :

More in general, the : + indention indicates that

- the block is to be executed repeatedly (`for` statement) – e.g., for each element from a list, or until a condition is reached (`while` statement)
- the block is only to be executed under specific conditions (`if`, `elif`, and `else` statements)
- an alternative block should be executed if an error occurs in the block (`try` and `except` statements)
- a file is opened, but should be closed again after the block has been executed (`with` statement)

What happened here?

```
1  for buddy in myfriends:
2      print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a `:`

More in general, the `:` + indention indicates that

- the block is to be executed repeatedly (`for` statement) – e.g., for each element from a list, or until a condition is reached (`while` statement)
- the block is only to be executed under specific conditions (`if`, `elif`, and `else` statements)
- an alternative block should be executed if an error occurs in the block (`try` and `except` statements)
- a file is opened, but should be closed again after the block has been executed (`with` statement)

What happened here?

```
1  for buddy in myfriends:
2      print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a :

More in general, the : + indention indicates that

- the block is to be executed repeatedly (`for` statement) – e.g., for each element from a list, or until a condition is reached (`while` statement)
- the block is only to be executed under specific conditions (`if`, `elif`, and `else` statements)
- an alternative block should be executed if an error occurs in the block (`try` and `except` statements)
- a file is opened, but should be closed again after the block has been executed (`with` statement)

What happened here?

```
1  for buddy in myfriends:
2      print (f"My friend {buddy} is {agedict[buddy]} years old")
```

The line *before* an indented block starts with a *statement* indicating what should be done with the block and ends with a `:`

More in general, the `:` + indention indicates that

- the block is to be executed repeatedly (`for` statement) – e.g., for each element from a list, or until a condition is reached (`while` statement)
- the block is only to be executed under specific conditions (`if`, `elif`, and `else` statements)
- an alternative block should be executed if an error occurs in the block (`try` and `except` statements)
- a file is opened, but should be closed again after the block has been executed (`with` statement)

Can we also loop over dicts?

Sure! But we need to indicate how exactly:

```
1  mydict = {"A":100, "B": 60, "C": 30}
2
3  for k in mydict: # or mydict.keys()
4    print(k)
5
6  for v in mydict.values():
7    print(v)
8
9  for k,v in mydict.items():
10   print(f"{k} has the value {v}")
```

Can we also loop over dicts?

The result:

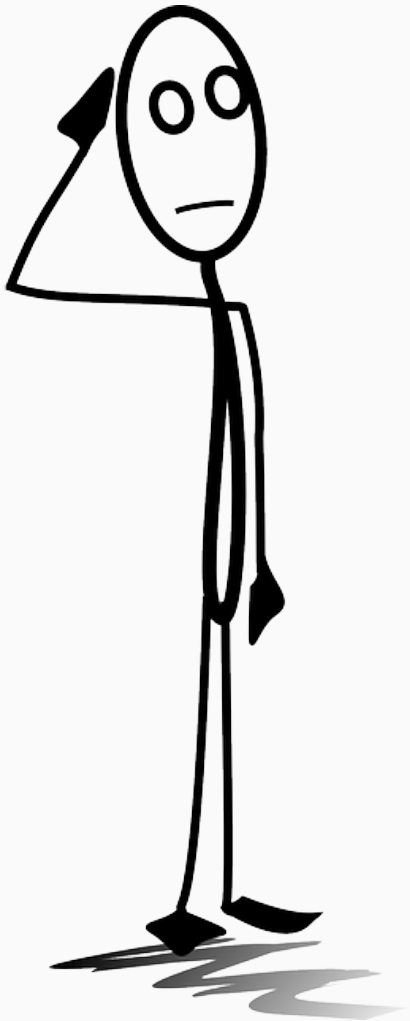
```
A
B
C

100
60
30

A has the value 100
B has the value 60
C has the value 30
```

if statements

Structure

Only execute block if condition is met

```
1  x = 5
2  if x <10:
3    print(f"{x} is smaller than 10")
4  elif x > 20:
5    print(f"{x} is greater than 20")
6  else:
7    print("No previous condition is met, therefore 10<={x}<=20")
```



Can you see how such an if statement could be particularly useful when nested in a for loop?

try/except

Structure

If executed block fails, run another block instead

```
1  x = "5"
2  try:
3    myint = int(x)
4  except:
5    myint = 0
```

Again, more useful when executed repeatedly (in a loop or function):

```
1  mylist = ["5", 3, "whatever", 2.2]
2  myresults = []
3  for x in mylist:
4    try:
5      myresults.append(int(x))
6    except:
7      myresults.append(None)
8  print(myresults)
```

try/except

Structure

If executed block fails, run another block instead

```
1  x = "5"
2  try:
3    myint = int(x)
4  except:
5    myint = 0
```

Again, more useful when executed repeatedly (in a loop or function):

```
1  mylist = ["5", 3, "whatever", 2.2]
2  myresults = []
3  for x in mylist:
4    try:
5      myresults.append(int(x))
6    except:
7      myresults.append(None)
8  print(myresults)
```

List comprehensions

Structure

A for loop that .append()s to an empty list can be replaced by a one-liner:

```
1 mynumbers = [2,1,6,5]
2 mysquarednumbers = []
3 for x in mynumbers:
4     mysquarednumbers.append(x**2))
```

is equivalent to:

```
1 mynumbers = [2,1,6,5]
2 mysquarednumbers = [x**2 for x in mynumbers]
```

Optionally, we can have a condition:

```
1 mynumbers = [2,1,6,5]
2 mysquarednumbers = [x**2 for x in mynumbers if x>3]
```

List comprehensions

Structure

A for loop that .append()s to an empty list can be replaced by a one-liner:

```
1 mynumbers = [2,1,6,5]
2 mysquarednumbers = []
3 for x in mynumbers:
4     mysquarednumbers.append(x**2))
```

is equivalent to:

```
1 mynumbers = [2,1,6,5]
2 mysquarednumbers = [x**2 for x in mynumbers]
```

Optionally, we can have a condition:

```
1 mynumbers = [2,1,6,5]
2 mysquarednumbers = [x**2 for x in mynumbers if x>3]
```

List comprehensions

A very pythonic construct

- Every for loop can also be written as a for loop that appends to a new list to collect the results.
- For very complex operations (e.g., nested for loops), it can be easier to write out the full loops.
- But mostly, list comprehensions are really great! (and much more concise!)

$\Rightarrow$ **You really should learn this!**

Mahrt, M., & Scharkow, M. (2013). The value of Big Data in digital media research. *Journal of Broadcasting & Electronic Media*, *57*(1), 20–33. https://doi.org/10.1080/08838151.2012.761700

van Atteveldt, W., Strycharz, J., Trilling, D., & Welbers, K. (2019). Toward Open Computational Communication Science : A Practical Road Map for Reusable Data and Code University of Amsterdam , the Netherlands. *International Journal of Communication*, *13*, 3935–3954.

Vis, F. (2013). A critical reflection on Big Data: Considering APIs, researchers and tools as data makers. *First Monday*, *18*(10), 1–16. https://doi.org/10.5210/fm.v18i10.4878