2 – Python basics

Bálint Aradi

Course: Scientific Programming / Wissenchaftliches Programmieren (Python)

https://www.bccms.uni-bremen.de/cms/people/b-aradi/wissen-progr/python/2022

Outline

- Data types
- Control structures
- Character formatting

Preparation (Python as native package)

This approach installs Python and Jupyter-Notebook as part of the OS

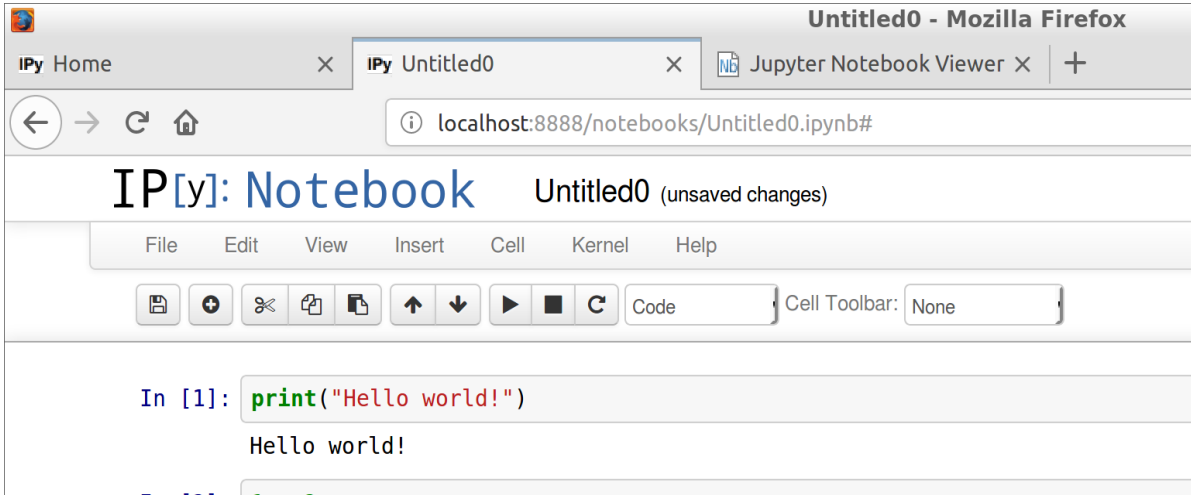
Alternatively, you can use the installation via Conda (see next slide), especially, if you wish to use JupyterLab or Anaconda-Navigator.

- Install Python 3 and the Jupyter notebook as packages on your system

```
sudo apt-get install python3 ipython3 jupyter-notebook
```

- You can start the IPython notebook by issuing:

```
jupyter-notebook
```



Preparation (Python install via Conda)

This approach installs Python and JupyterLab via Conda

- Download the latest Miniconda installer
- Execute the installer

```
bash ./Downloads/Miniconda3-py39_4.11.0-Linux-x86_64.sh
```

  - Specify ~/opt/miniconda3 as installation directory
  - Do not let conda to change your .bashrc file (to run conda init)

- Activate Conda

```
source ~/opt/miniconda3/bin/activate
```

- Install JupyterLab

```
conda install jupyterlab
```

- Start JupyterLab

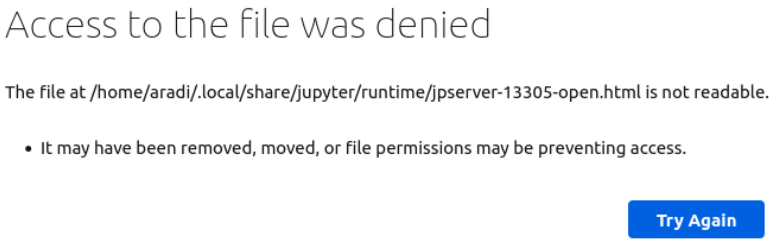
```
jupyter-lab
```

- Or you may start a Jupyter notebook directly:

```
jupyter-notebook
```

Preparation (Python install via Conda)

- If the start of a Jupyter notebook fails due to file access problems:



- Generate a config file for the Jupyter notebook

```
jupyter-notebook --generate-config
```

- Edit the generated config file

```
featherpad ~/.jupyter/jupyter_notebook_config.py &
```

  - Change line

```
# c.NotebookApp.use_redirect_file = True
```

    into

```
c.NotebookApp.use_redirect_file = False
```

  - Save file, exit editor

Python

- Python was created by Guido von Rossum 1989
- Has a huge community
- De facto standard script language for scientific applications
  (though Julia is becoming a possible alternative)
- Python is an **interpreted** language
  - Fast development (less code, no compilation necessary)
  - Often much slower than compiled languages
    (though, speed critical parts can be written in C/C++/Fortran)

Python comes in two “flavours”:

- Python 2, **deprecated**, support ended in 2020, **don’t use it for new projects**
  - There are still some scripts around which only under Python 2
- Python3, **actively developed**
  - Language has been “cleaned up” a bit and made more consistent
  - Few things incompatible with Python 2

Learning Python

Internet

- Official Python documentation, especially Tutorial and Library Reference: https://docs.python.org/3/
- Dive into Python (for advance learner, very good for OO-concepts)
- Newsgroups, mailing lists, stackoverflow, etc.
- :

Books

- M. Lutz: Learning Python (very-very detailed)
- M. Lutz: Programming Python (programming techniques)
- :

Experiencing the python shell

Interactive shell of the Python interpreter

```
python3
Python 3.5.2 (default, Nov 23 2017, 16:37:01) ...
>>> 1 + 1
2
>>>  Press Ctrl-D to leave the Python interpreter
```

Improved interactive shell IPython

```
ipython3
Python 3.5.2 (default, Nov 23 2017, 16:37:01) ...
IPython 2.4.1 -- An enhanced Interactive Python ...

In [1]: 1 + 1
Out[1]: 2

In [2]:   Press Ctrl-D to leave the IPython interpreter
```

Python as script

- Store the Python commands in a file and pass the file name to the interpreter as argument:

```
print("Hello world!")
```

Store this in the file hello_world.py (e.g. with leafpad)

```
python3 hello_world.py
```

Execute the source file with the Python interpreter

- By placing a special command in the first line and make the script executable, the shell (Bash) can automatically invoke the Python-interpreter for a given file:

```
#!/usr/bin/env python3
print("Hello world!")
```

Store this in the file hello_world (e.g. with leafpad)

```
chmod +x hello_world
```

Make the file executable

```
./hello_world
```

Execute the script

IPython / Jupyter notebook

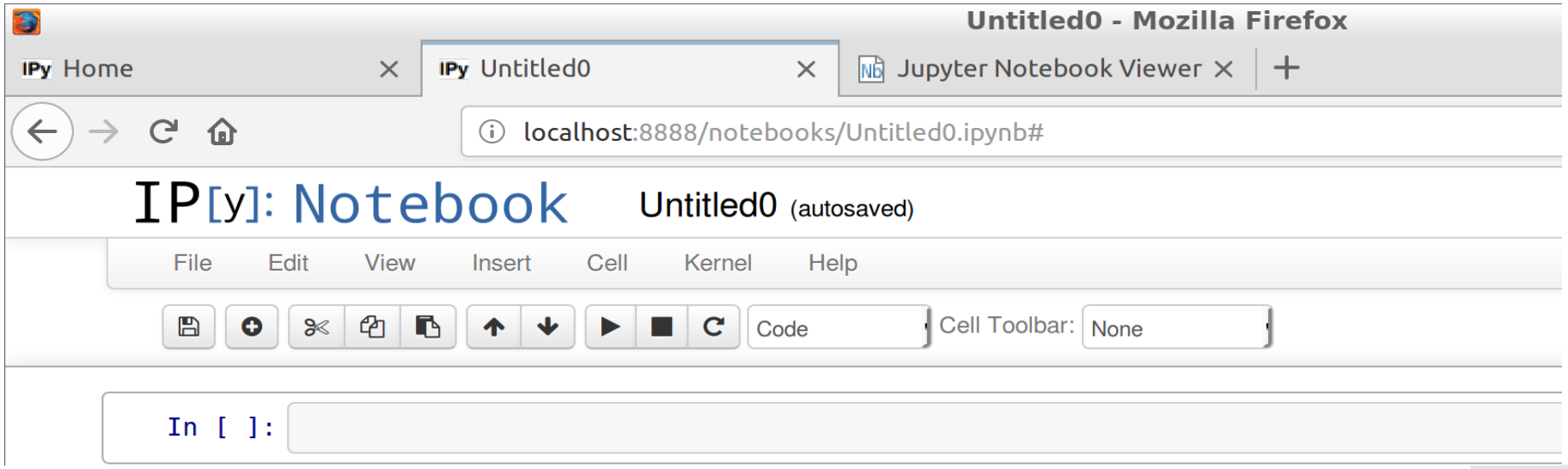
- Maple/Mathematica like web-based interface to Python
- Very practical when using Python in interactive mode (experimenting, evaluating data, producing figures for publications, etc….)

```
jupyter-notebook
```

You may need to activate the Conda environment first, if the notebook was installed via Conda:

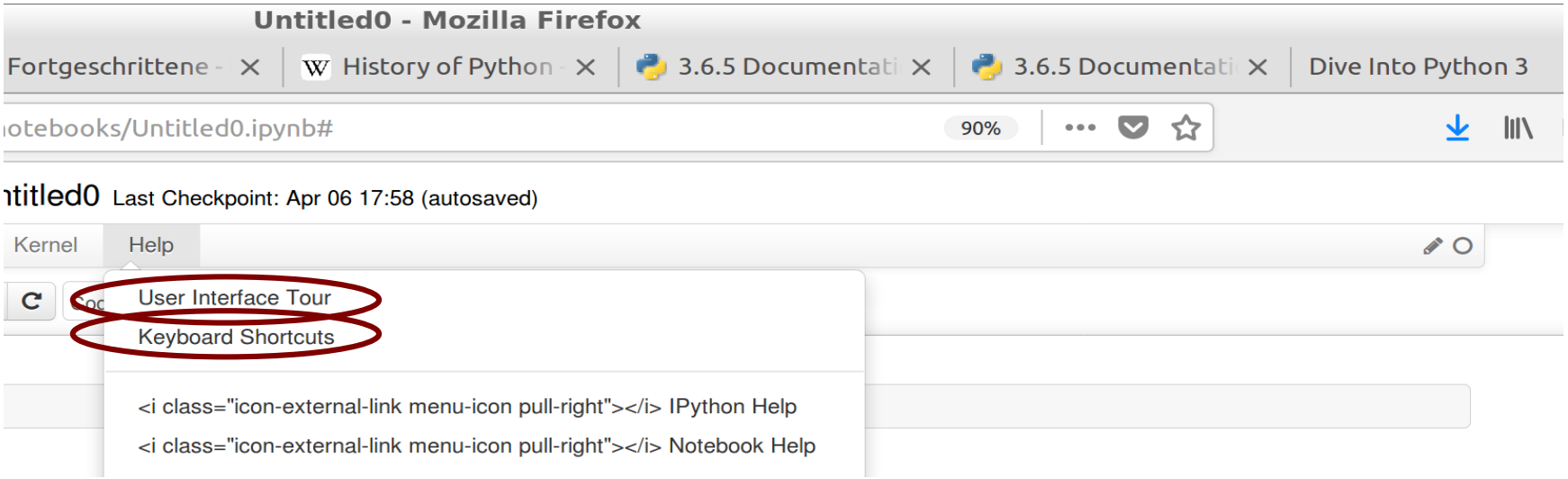
source ~/opt/miniconda3/bin/activate

Click on then **New** and then **Python3** [upper right corner]



IPython / Jupyter notebook

- Go through the User Interface Tour first
- Have a look a the Keyboard Shortcuts



Command mode: ESC

Edit mode: ENTER

Execute cell: Shift + ENTER

IPython / Jupyter notebook tips

- Tip: If you delete accidently a cell in Command mode (key ‘x’), you can undo it with key ‘z’
- You can cleanly rerun an entire worksheet by selecting following menus:
  - Kernel / Restart & Run All (to make sure all definitions are cleared)

Data types

Immutable data types

- Can not be changed once they have been created
- You must create a new (changed) instance if you want to change them
- Examples: bool (True, False), integer, float, string, tuple, frozen set, etc.

Mutable data types

- Their content can be changed after their creation
- Examples: list, set, dictionary, file, etc.
- Handling of mutable data types can have certain “side-effects”

Integer numbers

- Range is arbitrary
- Wenn value is beyond the long int data type in C (2**63 on 64 bit machines), it could become slow (runs via emulation, not natively)

Floating point numbers

Real numbers

- Range the same as double in C
  - +/-1E-323 – +/-1E+308, Precision: 16 digits
- Can be entered either in fixed or in expontential notation

```
>>> 0.123
0.123
>>> 1.23E-1
0.123
```

Complex numbers

- Represented by a pair of real numbers
- Real and imaginary part have the same range then usual real numbers
- Input as ***RealPart + ImaginaryPartJ***

```
>>> 2.0 + 3.3j
(2+3.3j)
```

Arithmetic operators

+	Addition
-	Substraction
*****	Multiplication
/	Division
//	Integer division
%	Division remainder
-	Negation
******	Power

```
>>> 1 + 2
3
>>> 3 - 4
-1
>>> 5 * 6
30
>>> 5 / 2
2.5
>>> 5 // 2
2
>>> 5 % 2
1
>>> -8
-8
>>> 2**0.5
1.4142135623730951
```

Relation operators

==	equal
!=	unequal
<	less
<=	less equal
>	greater
>=	greater equal

Comparison gives bool type as result (True/False)

```
>>> 3 == 2
False
>>> 3 != 2
True
>>> 3 < 2
False
>>> 3 > 2
True
>>> 3 >= 2
True
>>> 3 <= 2
False
>>> 3.0+2j < 2.0-1.2j
Traceback (most recent call last):
>>> 3.0+2j == 3.0+3j
False
```

Error: Complex numbers can not be ordered

Comparing with == or != is OK

Boolean values

- They are actually numbers, only shown differently
  - **False**: 0, **True**: 1

```
>>> True
True
>>> False
False
>>> 2 * True
2
```

Logical operators

- Logical **AND** (True if both operands True)
- Logical **OR** (True if any of the operands True)
- Logical **NOT** (Negates operand)

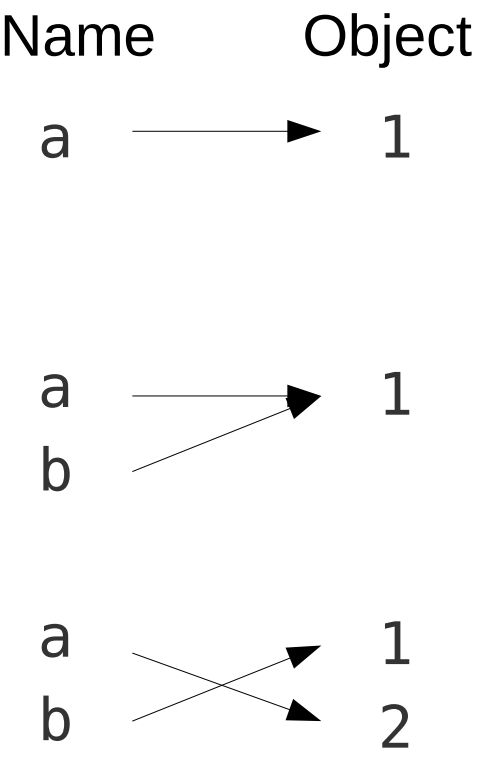
```
>>> True and False
False
>>> False or True
True
>>> not True
False
```

- In Python each object can serve as a logical value (details later)

Assignment

- An object (e.g. result of an operation) gets a name assigned (variable name)
- **Name = Objec**t
  Name should point to Object
- **Name1 = Name2**
  Name1 should point to the same object to which Name2 points
- When using a variable name in an expresssion, it will be substituted with the object it points to.
- There are **no “classic” variables** in Python, just **pointers/aliases**!

```
>>> a = 1
>>> a
1
>>> b = a
>>> b
1
>>> a = 2
>>> a
2
>>> b
1
>>> a + b
3
```



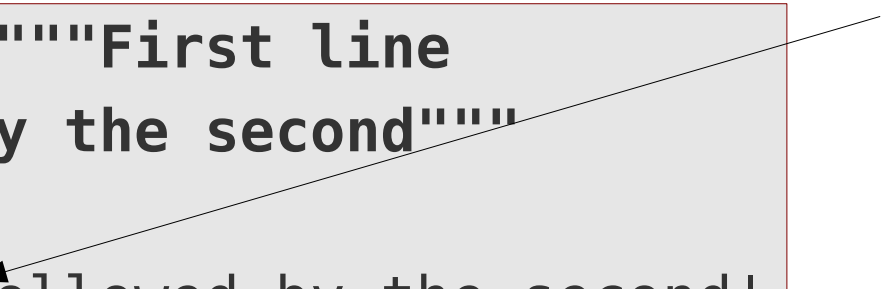
Strings

- Strings are specified between apostrophes or quotes:

```
>>> name1 = 'john'
>>> name2 = "tom"
>>> name1
'john'
>>> name2
'tom'
```

- Multilne strings can be specified between triple apostrophes or quotes:

```
>>> longstr = """First line
... followed by the second"""
>>> longstr
'First line\nfollowed by the second'
```



- Length of a string can be queried by the **len()** function:

```
>>> len(name1)
4
```

Strings

- Parts of a string can be accessed by the **[]** operator:

```
>>> txt = "some text"
>>> txt[0]
's'
>>> txt[0:4]
'some'
>>> txt[0:9:2]
'sm et'
>>> txt[:4]
'some'
>>> txt[4:]
' text'
>>> txt[8:4:-1]
'txet'
>>> txt[3:3]
''
```

Elements are enumerated **starting with zero**

When selecting ranges as **[lower:upper]**, the **lower bound** is **inclusive** the **upper bound** is **exclusive**

Range increment can be also specified with **[lower:upper:increment]**

When lower bound is omitted, range starts from the very first element (0 – range increment pos., last – range increment neg.)

When upper bound is omitted, range ends beyond last element (last element is included)

Negative range increment: iterating backwards

Empty range returns empty string

Strings

- Strings are immutable, they can not be changed once created:

```
>>> txt[0] = 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str'...does not support item assignment
```

- Strings can be concatenated by the **+ operator** or by whitespace for string literals:

```
>>> name1 + " " + name2
'john tom'
>>> "str1" "str2"
'str1str2'
```

- Strings can be **repeated** by the *** operator**:

```
>>> "ab" * 3
'abababab'
```

Converting data types into each other

- Each data type has a special function, which tries to convert its argument into an object with the given data type:

```
int(), float(), complex(), str()
```

- Argument can have arbitrary data type
- If the conversion fails, an exception is raised (error)

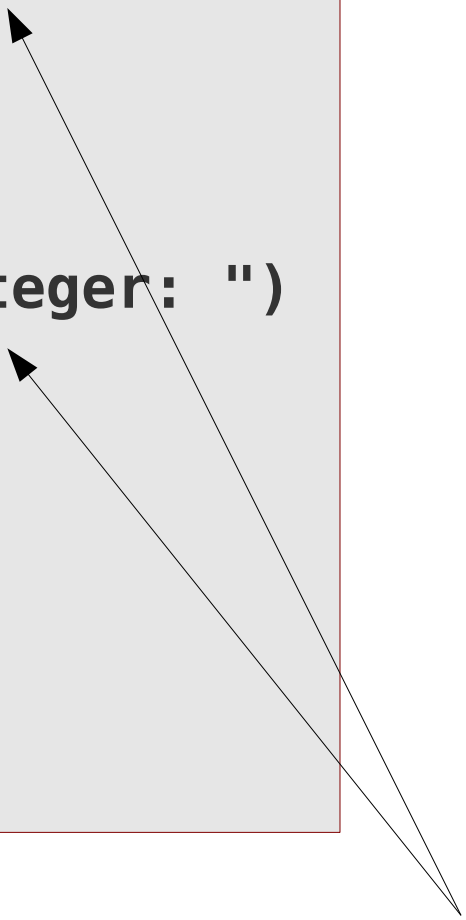
```
>>> int(3.2)
3
>>> float("12.1")
12.1
>>> complex("3+2j")
(3+2j)
>>> complex("3.0+2.0j")
(3+2j)
```

```
>>> valstr = "3"
>>> int(valstr)
3
>>> int("hello")
Traceback ...ValueError: ..
.
```

Input

- The **input()** function stores user input (one line) in a string

```
>>> answer = input("Your answer: ")
Your answer: No
>>> answer
'No'
>>> answer = input("Enter an integer: ")
Enter an integer: 12
>>> answer
'12'
>>> num = int(answer)
>>> num
12
```



Message to print at input line

Branching

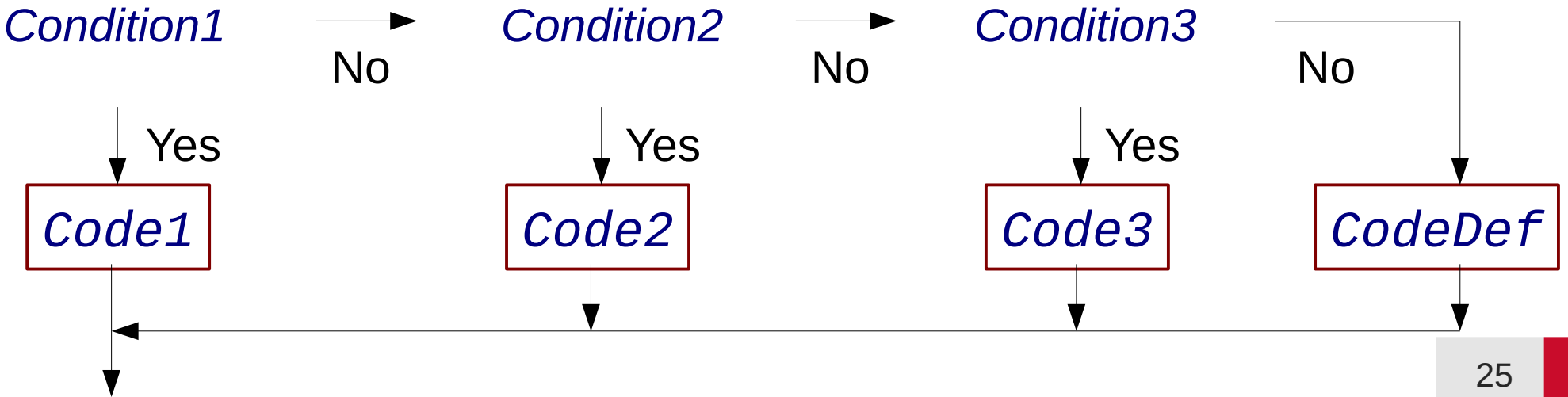
- Optional code execution based on condition evaluation

```
if Condition1:
    Code1
elif Condition2:
    Code2
elif Condition3:
    Code3
else:
    CodeDef
```

Start of a nested block

Indentation signalises nesting

- Nested blocks in Python **start with colon (:)**
- One should always use **4 spaces as indentation**
- End of nested block is signalised by an **unindented statement**



Indentation in Python

- Indentation is not optional, but part of the language semantics
- Indentation signalises nesting
- Amount of indentation signalises nesting depth
- Each nested block should be indented by exactly 4 space characters
- Inconsistent indentation leads either to syntax error or to wrong code logics

```
if answer[0] == "y":
    print("OK, you agree")
else:
    print("I see")
    print("You don't agree")
print("Let's continue")
```

Indented, belongs to if-block
(Only executed if answer[0] == “y”)

Indented, belongs to else-block
(Only executed if answer[0] != “y”)

Unindented, outside of if/else block
(Always executed)

- Use an editor which supports Python to ensure proper indentation!

If-else expression

- One can choose between two expressions with an if/else construct within an expression
- Use it only for trivial (short) cases

Syntax:

```
true_expression if condition else false_expression
```

```
mytype = "pos. semidef" if b >= 0 else "negative"
```

Evaluation as bool expression

- Each object can be evaluated as a bool expression
- Evaluation is type dependent: Numerical types are usually False, if their value is zero. Container types are usually False, if they are empty

Object type	Evaluated to False	Evaluated to True
bool	False	True
int	0	any other value
float	0.0	any other value
complex	0.0+0.0j	any other value
string	"" (empty string)	contains at least one char.
list	[] (empty list)	contains at least one element
dict	{} (empty dict)	contains at least one element

```
if num % 2:
    print("odd")
else:
    print("even")
```

```
if num % 2 != 0:
```

while loop

- Repeats a program block as long a condition is fulfilled

```
while Condition:
    Loop code
```

- If the condition is not fullfilled (any more), code execution continues after the while-block

```
num = 1
while num <= 20:
    print(num)
    num = num * 2
print("Reached 20: ", num)
```

→

```
1
2
4
8
16
Reached 20: 32
```

while loop: break, continue

- Execution order in loops can be modified:
  - **break**: terminates loop and continues execution after loop block
  - **continue**: jumps back to loop header and evaluates loop condition again

```
while True:
    answer = input("Do you agree (y/n)? ")
    if answer != "y" and answer != "n":
        print("Invalid answer! Try it again!")
        continue
    if answer == "y":
        print("Good answer, thanks!")
        break
    print("Valid answer, but I don't like it!")
print("Nice that we agree!")
```

Endless loop, should be exited via break at some point

while loop: else

- OIptional else-branch of a while loop is executed, if the loop execution was aborted due to loop condition becoming False (and not due to a break statement)

```
ii = 0
while ii < 5:
    ii += 1
    answer = input("Do you agree? (y/n) ")
    if answer == "y" or answer == "n":
        break
else:
    print("Too many invalid answers, I'll assume yes.")
    answer = "y"
print("Your answer was: ", answer)
```

Note the (missing) indentation

for loop

- Iteration over given values can be realised with a **for-loop**

```
for loop_variable in iterable_object:
    loop code
```

- The iterable object can be anything, which is able to return values one-by-one (implements the iterator-interface)
- Example: string is iterable, it returns its characters one by one:

```
name1 = 'john'
for char in name1:
    print("Char: ", char)
```

```
Char:  j
Char:  o
Char:  h
Char:  n
```

Range iterator

- The **range()** function returns an iterator over integers

```
range(from, to, step)
```

- Lower bound is included, upper bound is excluded (as for substring ranges)

```
range(0, 10, 2)
```
→ `[0, 2, 4, 6, 8]`

- If step size is omitted, step is is assumed to be 1

```
range(0, 4)
```
→ `[0, 1, 2, 3]`

- If **range()** is called with one argument, it is interpreted as upper bound

```
range(4)
```
→ `[0, 1, 2, 3]`

- If selected range is empty, iterator does not return any values

```
range(4, 4)
```
→ `[]`

Note: You can use the list constructor to explicitely show the values yielded by an iterator: `list(range(4))`

for loop: break, continue

- The break and continue statements can be also used within a for-loop
  - **break**: Terminates loop execution a continues after loop-block
  - **continue**: Jumps to loop header and iterates over next item

```
for num in range(4, 8):
    if not num % 5:
        break
print("Num: ", num)
```

→ `Num:  5`

```
for num in range(4, 8):
    if not num % 5:
        continue
    print(num)
```

→

```
4
6
7
```

for loop: else

- The **else** branch of a for-loop is executed, if the loop terminated after having iterated over all elements (and not due to a break statement)

```
for num in range(6, 10):
    if not num % 5:
        break
else:
    print("No multiple of 5 found")
```

Equivalent code

```
found = False
for num in range(6, 11):
    if not num % 5:
        found = True
        break
if not found:
    print("No multiple of 5 found")
```

String formatting

See F-strings for a more modern string formatting approach

(the formatting mini-language is the same, though)

- Placeholder with special formatting can be added to strings
- Values for the placeholders can be provided by the **format()** method
- The result is a new string with the substituted values

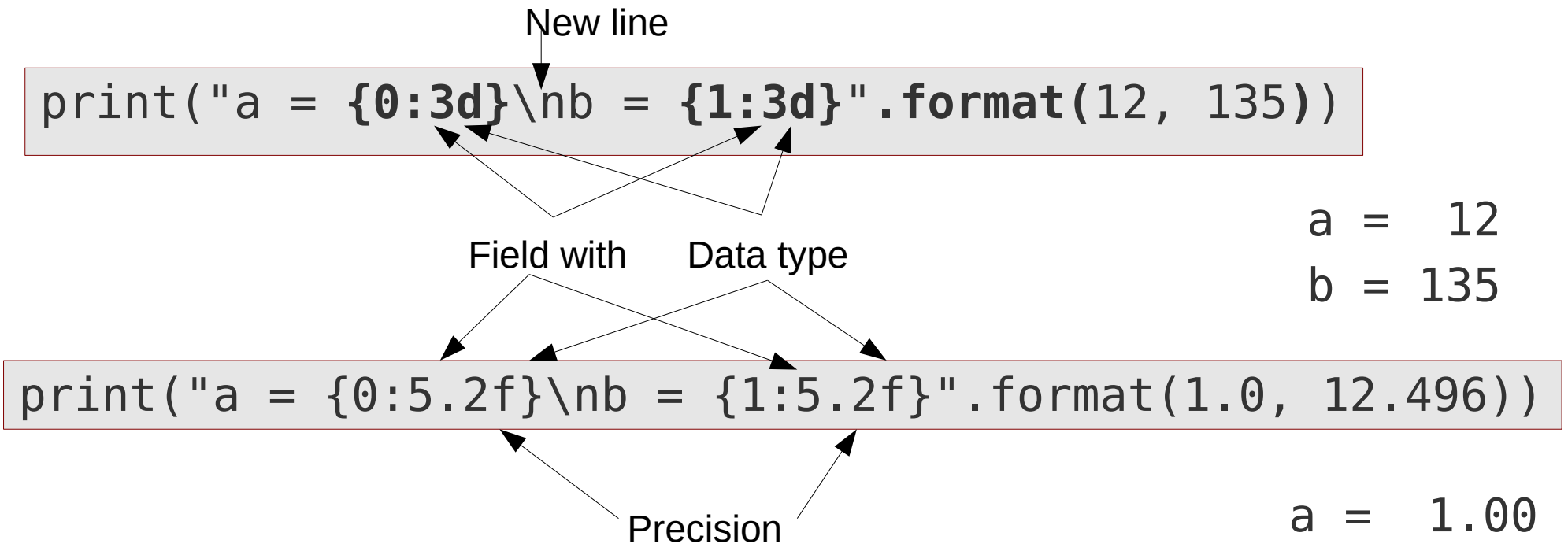
```
"a0 = {0}, a1 = {1}".format(12, 31)  → 'a0 = 12, a1 = 31'
```

- The numbers in the placeholder indicate which argument of **format()** should be substituted.
- A given argument of format() can be substituted multiple times

```
"{0} * 1 = {0}".format(31)  → '31 * 1 = 31'
```

String formatting

- Type specific formatting options are specified after placeholder number, **separated by a colon (:)**



- The type of the arguments of **format()** must match the type specific options

```
"The {0:d}. number".format(2)      → 'The 2. number'
"The {0:d}. number".format(2.0)    → ValueError: ...
```

A few styling options

:*W*d	integer number
:*W*.*P*f	floating point number in fixed notation
:*W*.*P*e	floating point number in exponential notation (with small e)
:*W*.*P*E	floating point number in exponential notation (with capital E)
:*W*.*P*g	:f or :e depending on the value of the floating point
:*W*.*P*G	:f or :e depending on the value of the floating point
:W*s*	string (converts given object to a string)

W (width) minimal field width

.P (precision) number of decimal places

} optional

```
"{0:12.4E}".format(1.2)
"{0:12E}".format(1.2)
"{0:.4E}".format(1.2)
"{0:5s}".format("ab")
```

→

```
'  1.2000E+00'
'1.200000E+00'
'1.2000E+00'
'ab   '
```

} Numbers aligned right

String aligned left

See also Format specification mini language

Few remarks on string formatting

- If the field with is too small for the given represenation, it will be automatically expanded

```
"|{0:1d}|".format(123)  ⟶  '|123|'
```

- If you need literal curly braces in the formatting string, they must be doubled:

```
"{{{0:d}}}".format(123)  ⟶  '{123}'
```

- Since Python 3.1 you can leave away the sequential numbers in the place holders, they will be numbered then automatically

```
"{:d} + {:d} = {:d}".format(3, 4, 7)  ⟶  '3 + 4 = 7'
```

F-strings (Python >= 3.6)

- Arbitrary Python expressions can be inserted into an **f-string**

```
f"..{<Python expression>:<formatting options>}.."
```

- The expression is evaluated, the result is inserted into the string
- Colon and formatting options are optional

```
a = 13.4
print(f"Value of a: {a}")
Value of a: 13.4

print(f"Value of a: {a:13.4E}")
Value of a:    1.3400E+01

print(f"Evaluating 1 + 2: {1 + 2}")
Evaluating 1 + 2: 3
```