HSLU Lucerne University of Applied Sciences and Arts

Python for Data Science: SW05

Functions
Strings
Files

Information Technology
March 13, 2025

FH Zentralschweiz

Content

Lucerne University of Applied Sciences and Arts

Python Functions

Information Technology

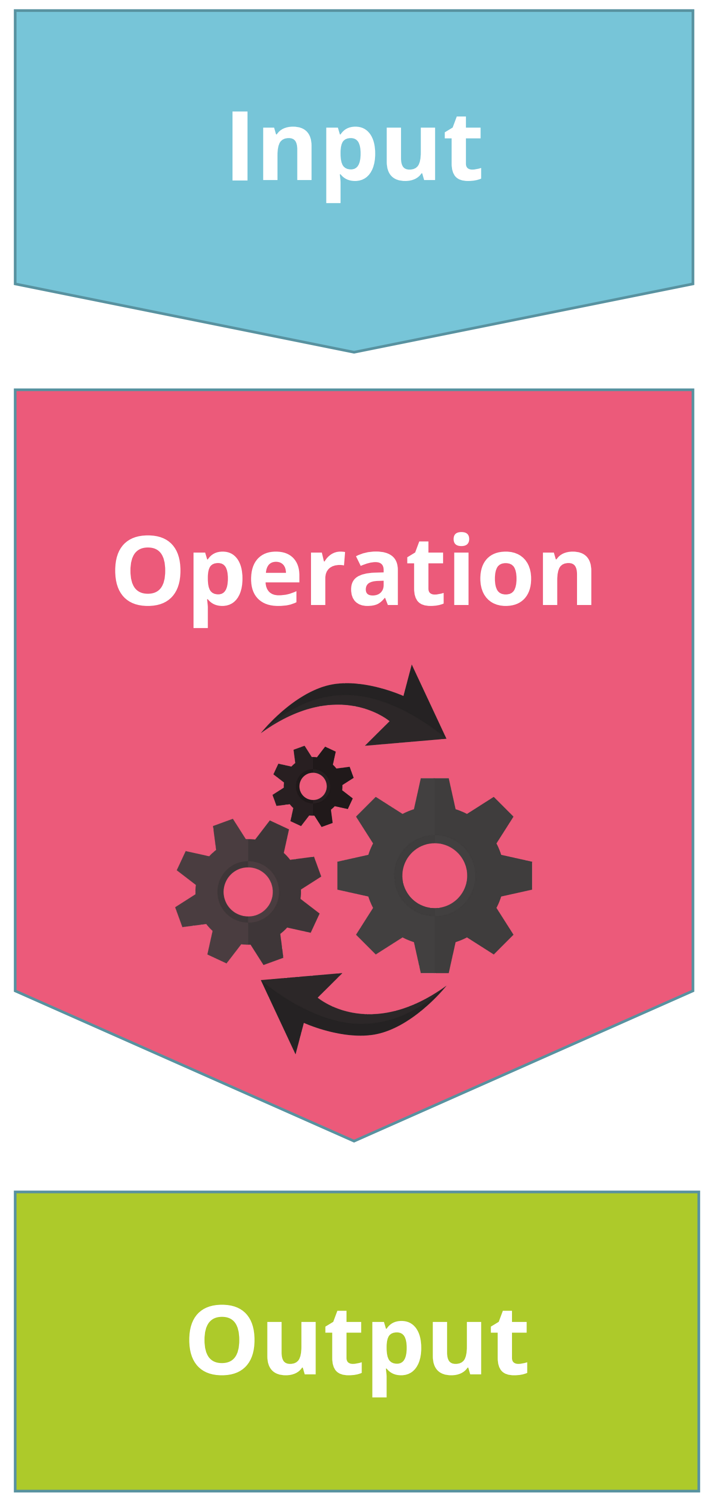
March 13, 2025

Python Functions

Functions allow to combine multiple instruction into a function block that can be executed multiple times. Main advantages of functions are:

- Ease of debugging and testing.
- Modularity (mitigates code duplication).
- Readability.

Docu: https://docs.python.org/3/howto/functional.html



Functions…

- tackle only **one** particular issue.
- are named by convention in snake case (i.e. lower case separated with '_'):
  `my_function()` or `value_converter()`
- ideally operate on its **input only** and produce some **output**.
- must be defined before use.

Python Functions

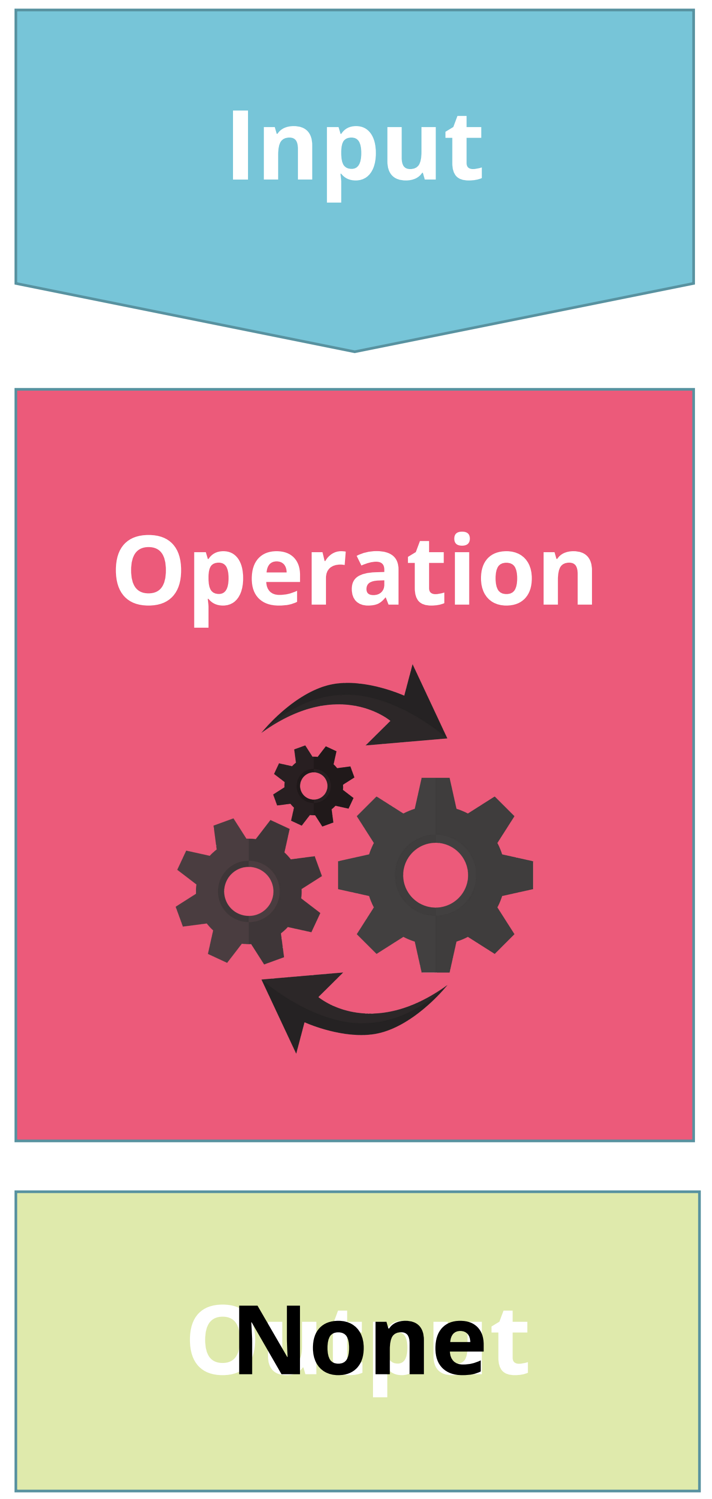
Functional style discourages functions with **side effects**:

- No internal state modification.
- All changes visible in function’s output (return).
- Output must only depend on input.
- **Purely functional** functions (no side effects).

But it’s difficult to avoid all side effects, such as when printing to the screen or pausing execution for a second:

- print()
- time.sleep()

Consequently, sometimes functions have “no” output.



Python Functions: Structure (syntax)

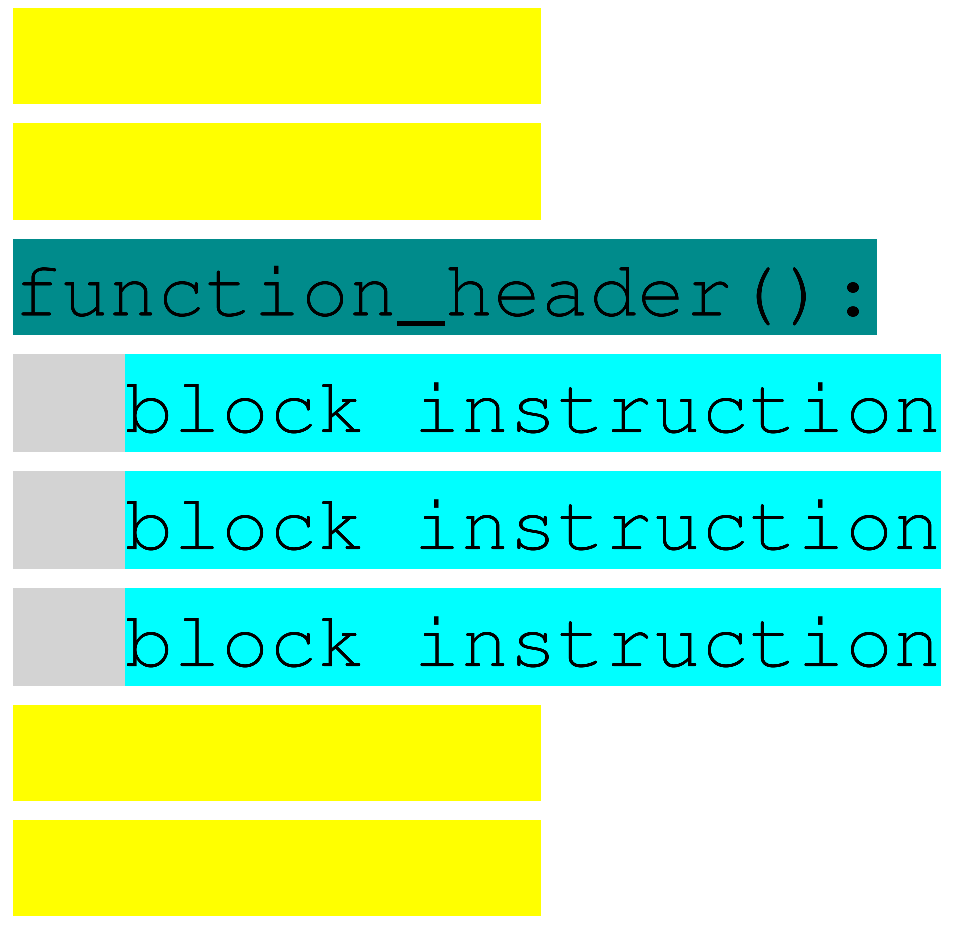
From the structural point of view, a function consist of two main elements:

Function header

- Function name
- Parameter list

Function body

- List of instructions (operations)
- Return value

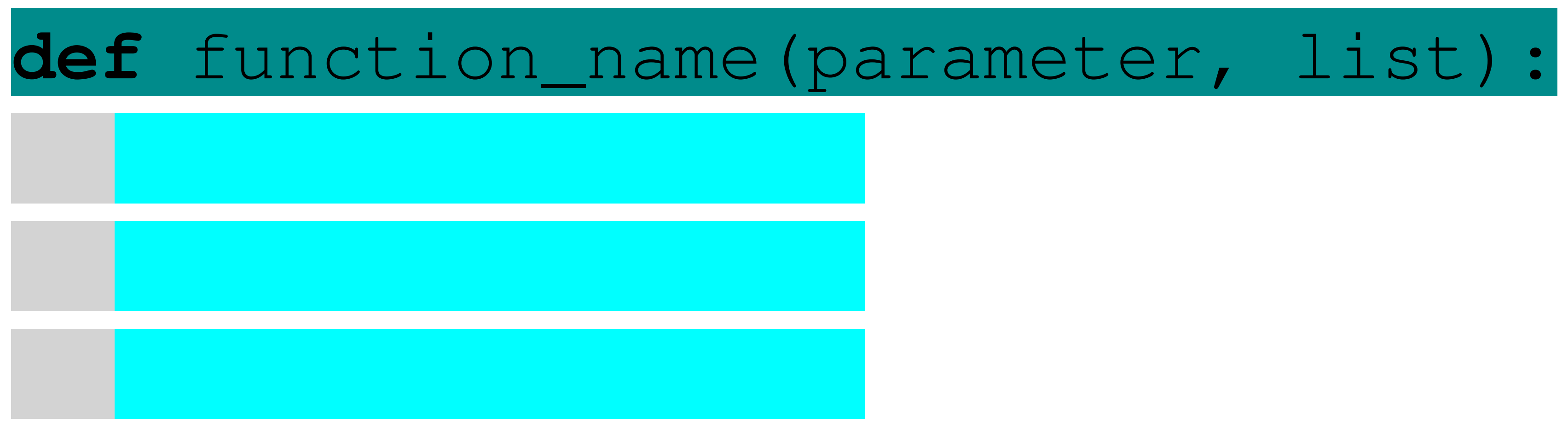


Python Functions: Structure (syntax)

From the structural point of view, a function consist of two main elements:

Function header

- Function name
- Parameter list

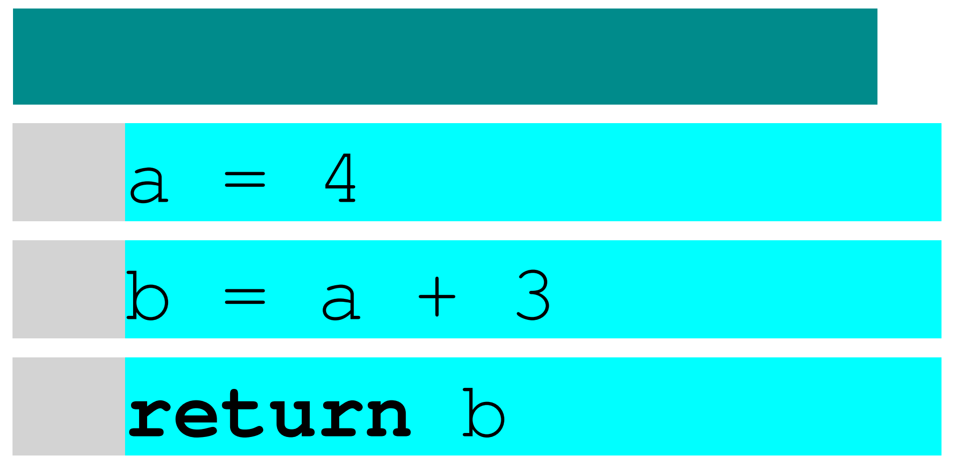


```
def function_name(parameter, list):
```

- Function is defined with: **`def`**
- Parameter (input) list can be of arbitrary length.

Function body

- List of instructions (operations)
- Return value

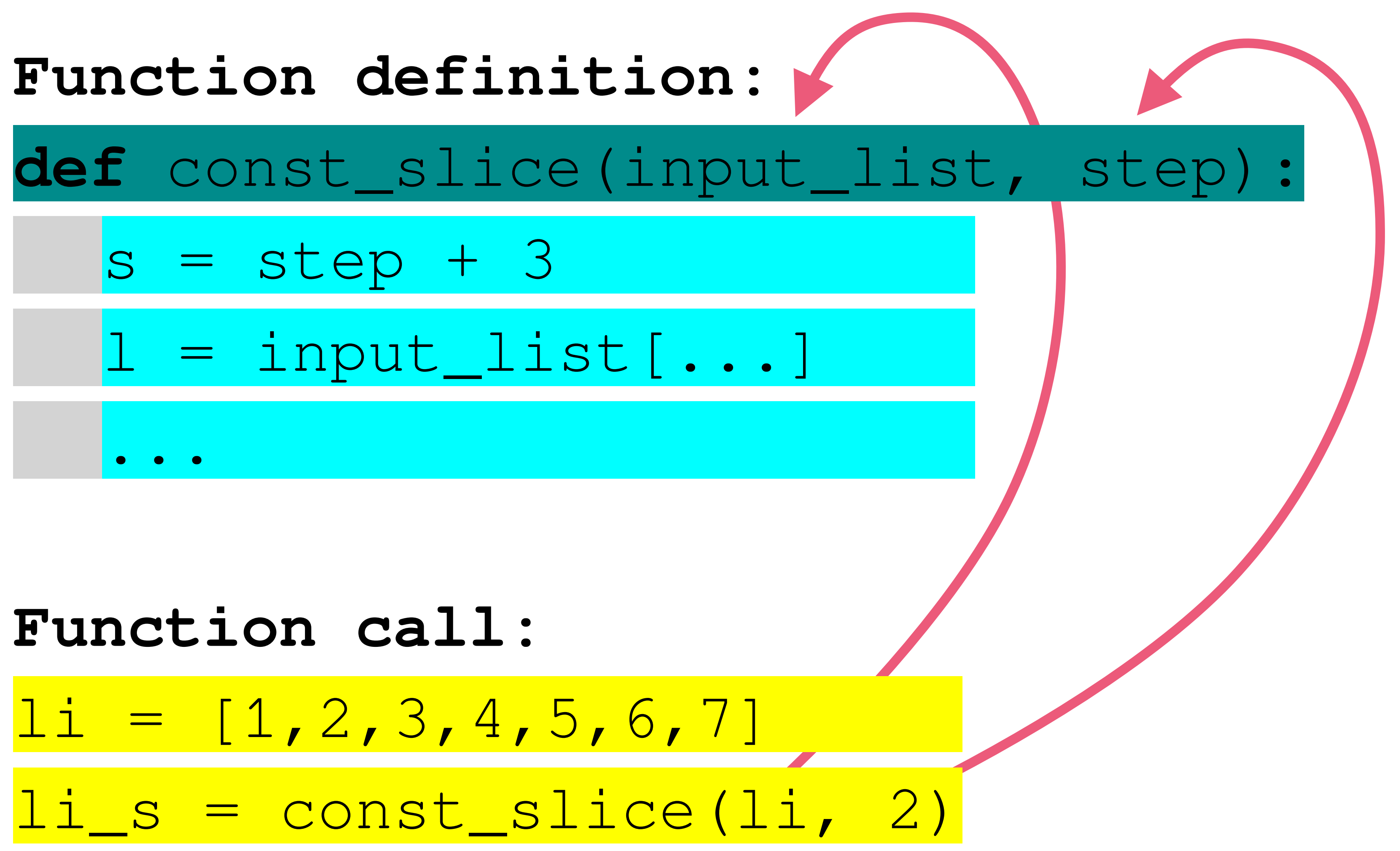


```
    a = 4
    b = a + 3
    return b
```

- Can consist of unlimited number of instructions.
- Function body can also call other functions.
- Function can have **nested** functions.

Python Functions: Call with (Keyword-)Parameters

Parameters are organized in a list and define the input for the function.
-> input values are assigned to local variables.



Function definition:

```
def const_slice(input_list, step):
    s = step + 3
    l = input_list[...]
    ...
```

Function call:

```
li = [1,2,3,4,5,6,7]
li_s = const_slice(li, 2)
```

Parameter list…

- can be or arbitrary length (also empty)
- has by default ‘fixed’ order
- can contain optional parameters with default value

  `def const_slice(input_list, step = 3):`
- support keywords to allowing changed order

  `def const_slice(input_list, step = 3):`
- does not proof data type
- accepts only **pointer** to storage location
  (can also be a pointer to a function)

  `def apply_fnc(input_list, fnc):`

Python Functions: Return value

All functions (including purely functional) have one return value; at least **`None`**.

- return value is a pointer to a storage location; or `None`.
- if function does not explicitly return a value, it implicitly returns `None`.
- function can return any data type including pointer to other functions.
- multiple returns are collected into a list: `return list(‘hello’, 7, ‘computer’)`

Returns can be assigned to a variable:

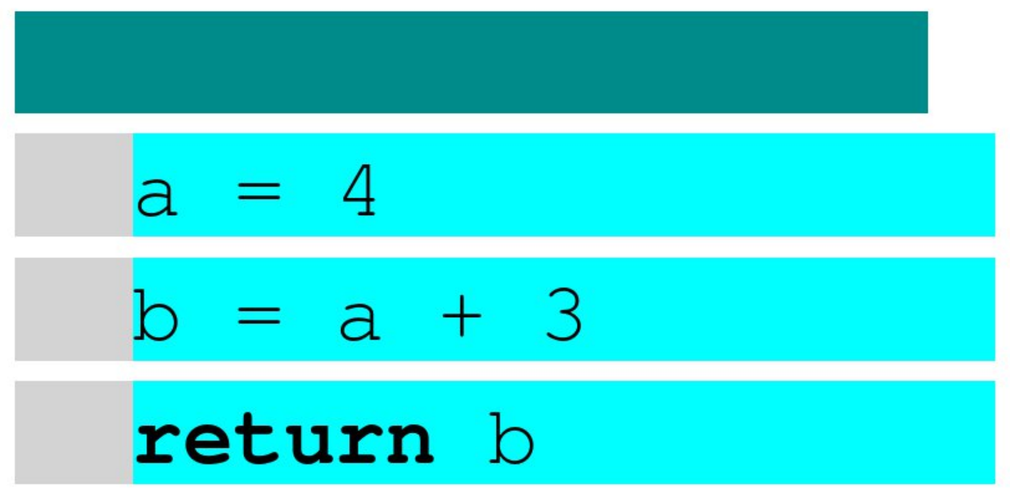
```
var1 = my_function(my_list)
```

Returns can be used as reference and passed to another function:

```
my_wrapper(my_function(88))
```

Returns can be implicitly used as reference for chaining (object oriented programming):

```
my_function(88).my_wrapper()
```



```
a = 4
b = a + 3
return b
```

HSLU Lucerne University of Applied Sciences and Arts

String Formatting

Information Technology

March 13, 2025

String Formatting

Python treats a string as a **sequence** (list) of characters (single letters, symbols or escape characters).

Strings…

• can be sliced like regular sequence data types.	`li[:10:2]`
• can be concatenated.	`li[0:5] + li[6:11]`
• allow index based access to particular elements.	`li[5]`
• allow formatting based on character positions.	`‘•••lo’  -> (right aligned with 5 characters)`

`li :`	`'H'`	`'e'`	`'l'`	`'l'`	`'o'`	`' '`	`'w'`	`'o'`	`'r'`	`'l'`	`'d'`	`'\n'`
`idx:`	0	1	2	3	4	5	6	7	8	9	10	11

String Formatting

Formatting strings means defining the representation and treatment of particular sequences.

Example problem: a dictionary with street names and house numbers should be printed in a structured way
-> addresses left aligned, house numbers right aligned.

Python supports 4 different ways:

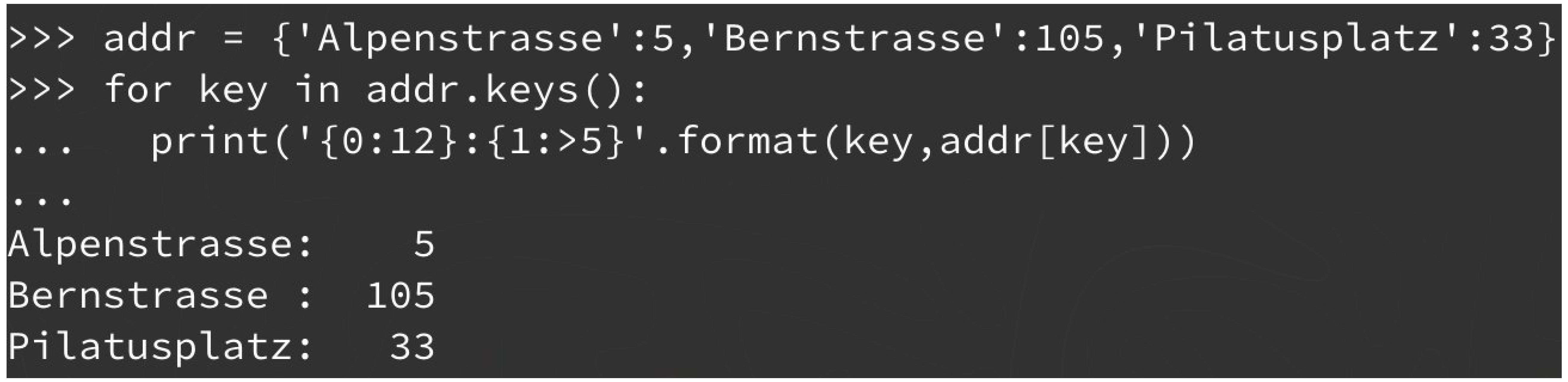
- Formatted String Literals

  `f”hello {my_str}”`
- The String format() Method

  `“hello {}”.format(my_str)`
- Manual String Formatting
- String interpolation

  `“hello %5.3” % (pi)`

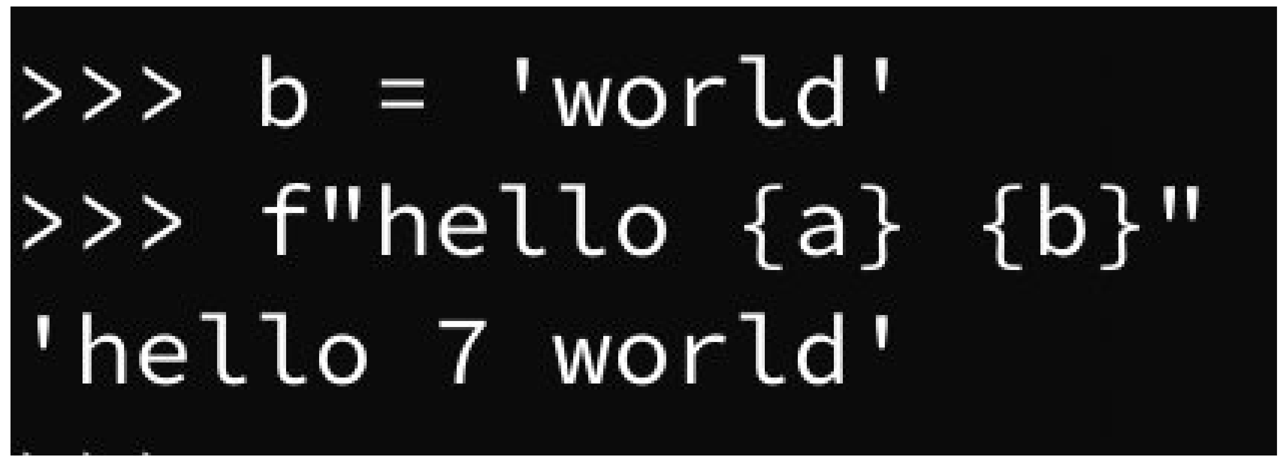
Not discussed in this course.



```
>>> addr = {'Alpenstrasse':5,'Bernstrasse':105,'Pilatusplatz':33}
>>> for key in addr.keys():
...   print('{0:12}:{1:>5}'.format(key,addr[key]))
...
Alpenstrasse:    5
Bernstrasse :  105
Pilatusplatz:   33
```

String Formatting: Formatted String Literals

Formatted string literals allow to include values in strings by variable names.
e.g. for `var1 = 7, var2 = ‘world’`



```
>>> b = 'world'
>>> f"hello {a} {b}"
'hello 7 world'
```

String literals...

- start with a prefixing f or F before the string: `f”hello {var1} {var2}.”`
- enclose variables with curly brackets: `f”{var1}”`
- allow formatting after : in brackets: `f”{var1:>5}`

property	code	output
defined numbers (x=5) of characters	`f"hello {var1:>5} {var2}"`	`'hello ····7 world'`
self documentation and debugging	`f"hello {var1=} {var2}"`	`'hello var1=7 world'`

- format strings according to: https://docs.python.org/3/library/string.html#formatstrings

String Formatting: String format() Method

The format() method provides a similar but more **flexible way** for variable alignment than string literals. In contrast, with the format() method the variables are passed to the format-method and the string comprises placeholders {} only.

```
“hello {} {}.”.format(var1, var2)
```

The alignment of the variables to the placeholders can be done by:

property	code	output
strict order:	`"hello {} {}.".format(var1, var2)`	`'hello 7 world'`
numbering:	`"hello {1} {0}.".format(var1, var2)`	`'hello world 7'`
keywords:	`"hello {a} {b}.".format(b=var1, a=var2)`	`'hello world 7'`
dictionary*:	`"hello {0[num]} {0[txt]}.".format(dict)`	`'hello 7 world'`

```
*dict = {‘num’:7, ‘txt’:’world’}
```

HSLU Lucerne University of Applied Sciences and Arts

File Handling

Information Technology

March 13, 2025

File Handling

Files enable storing data outside of the application and hence to:

1. keep the information for next execution or
2. share data between different application.

- Python accesses files through a file object that is created with:

```
file_object = open(file_name, mode)
```

- The programmer who **opens** a file is also responsible to **close** the file!

```
file_object.close()
```

- Python takes over this responsibility,
if file access is handled with the keyword: **with**

```
with open(file_name, mode) as f:
    ...
```

mode	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	open for exclusive creation, failing if the file already exists
'a'	open for writing, appending to the end of file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open for updating (reading and writing)

Docu: https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files

File Handling: Read

Reading from a file relates mostly one the following objectives:

1. Reading certain quantity of data:

```
content = file.read(size)
```

2. Reading file line by line in a while or for loop and performing operations on each line:

```
while content not condition:                  for line in file:
    content = file.readline()                     ... line
```

`file.readlines()` or `list(file)` return all lines in a list.

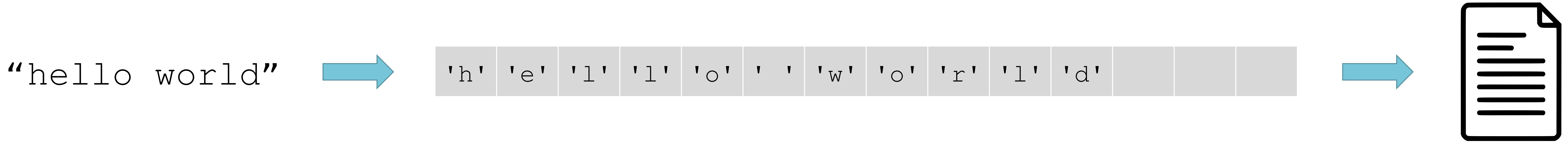
3. Reading all file at once:

```
content = file.read()
```

- `read(size)` returns the entire content of the file when size is: **empty** or **negative**
- if `read` returns the entire content, Python does **not** care about the **memory** size!
- if `read` reached end of file, it will return an empty string ‘ ‘.

File Handling: Write

Writing content to a file is more critical compared to reading. Data written to a file go through a buffer that writes packages to the file.



The buffer results in unfinished writing process as long as the file stream (`file_object`) is not flushed by:

1. `file_object.flush()` or
2. `file_object.close()`

   -> note: `with open(file_name) as f:` closes the file_object and flushes the stream when finished

```
Example writing to a file:

with open(file_name, ‘w’) as file:
    file.write(‘hello world’)
```

Docu: https://docs.python.org/3/library/io.html

HSLU Lucerne University of Applied Sciences and Arts

School of Computer Science and Information Technology
Research
Ramón Christen
Research Associate Doctoral Student

Phone direct +41 41 757 68 96
ramon.christen@hslu.ch

HSLU T&A, Competence Center Thermal Energy Storage
Research
Andreas Melillo
Lecturer

Phone direct +41 41 349 35 91
andreas.melillo@hslu.ch