University of Zurich[UZH]

# Python Basics

Philipp Gloor[1]

[1]University of Zurich

# Table of Contents

# General Introduction

# About me

## Education

- 2012 – Bachelor of Science UZH in Physics
- 2016 – Master of Science UZH in Computational Science

## Work

- 2014 – 2016: Software engineer CERN (remote)
- 2016 – 2021: PDF Tools AG
- 2021 – now: Zurich Instruments

## Programming experience

C++, C#, Java, TypeScript, JavaScript, Python

## Email

philipp.gloor@protonmail.com

# Learning targets

After this course...

- ... you will have an idea what programming is
- ... you will know how to write a basic computer program
- ... you are able to write a Python program based on a written out problem statement
- ... you know where you can find more information to improve your programming skills

But you only have started to scratch the surface.

# Introduction to Programming

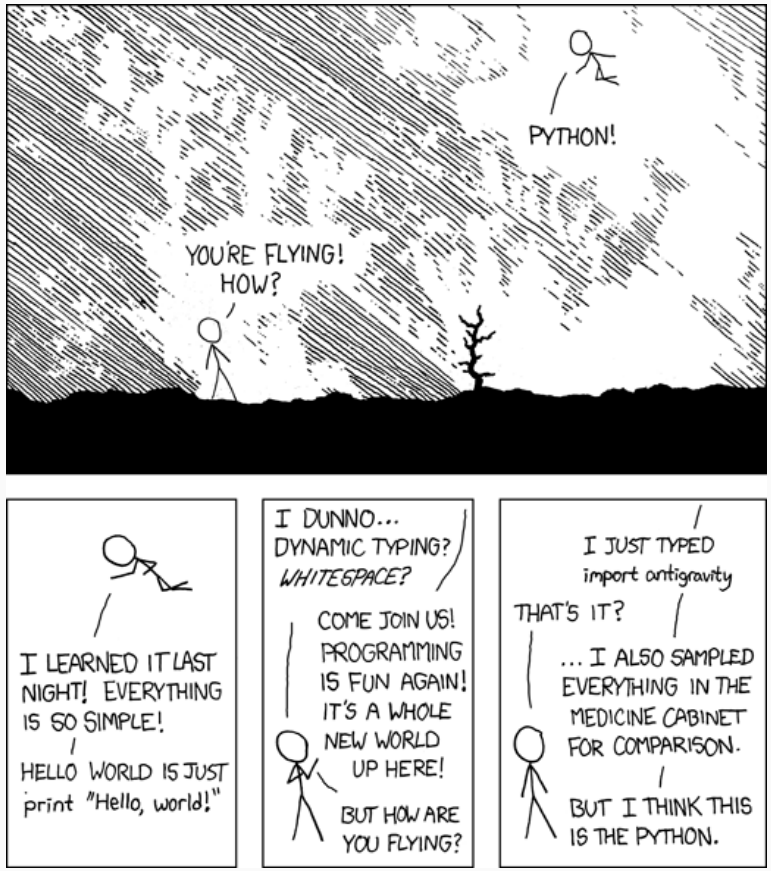
# What is a Computer Program

## Modular System

- **Input**: Data input from keyboard, files, internet, etc...
- **Output**: Processed data is displayed or saved to a file
- **Algorithms**: The computer's cooking recipes
- **Libraries**: Using existing implementations (can do anything of the above)

# Why Python?

- High-level programming language
- ”Simple” syntax
- Cross-platform - A script written on a Windows computer also runs on Linux & Mac
- Interpreted => Easy to run
- Many libraries available




Source: https://xkcd.com/353/

# Examples: Hello World i

High level languages but not trivial to learn:

## Java

```java
public class HelloWorld {
  public static void main(String args[]) {
    System.out.println("Hello World");
  }
}
```

## C++

```cpp
  #include <iostream>
  int main() {
    std::cout << "Hello World\n";
    return 0;
  }
```

# Examples: Hello World ii

Or even worse:

**Machine Language**
Example of a low level language

```
.LC0:
  .string "Hello world!"
main:
  push rbp
  mov rbp, rsp
  mov edi, OFFSET FLAT:.LC0
  mov eax, 0
  call printf
  mov eax, 0
  pop rbp
  ret
```

# Examples: Hello World iii

Python

```python
print("Hello World")
```

# How to Run Python Code

**Options to run Python code:**

- Directly in the Python prompt (REPL - Read, Eval, Print, Loop)
- Write the code into a file and run python with the file
- Use IDE to run Python code

# Development Environment

- Integrated Development Environment (IDE)
- Collection of tools that are commonly used for software development (they make our life easier!)
- Popular IDEs
  - Visual Studio Code - `https://code.visualstudio.com`
  - JetBrains PyCharm - Community Edition available for free `http://jetbrains.com/pycharm/download`
- It takes time to get proficient using an IDE

# Fundamental Concepts

# Variables, Types, Expressions, Operators, Comments

# Variables i

Variables are used to store information to be referenced and manipulated in a computer program. They also provide a way of labeling data with a descriptive name, so our programs can be understood more clearly by the reader and ourselves. It is helpful to think of variables as containers that hold information. Their sole purpose is to label and store data in memory. This data can then be used throughout your program.

- Variables hold values
- Similar to mathematics
  - x = 2
  - y = x + 2
- Values assigned using the = operator

# Data Types

## Types

- Numbers (Integers, Floats)
  - 2
  - 1000000
  - -2
  - 3.2
  - 1.333333

# Data Types i

## Strings

- Strings (Text)
  - 'Hello World'
  - "Hello World"
- 'Single quotes' or "double quotes" can be used to create them
  - 'Hello World'
  - "Hello World"
  - "5"

## Boolean

Binary data type

- True
- False

# Data Types ii

## None Type

- `None` represents ”nothing” or ”no value”
- Think of it like an empty box - the box exists, but there’s nothing inside
- Common examples:
  - A person’s middle name when they don’t have one
  - The result when a function doesn’t return anything
  - An uninitialized variable

# Data Types iii

## Examples

```
middle_name = None  # Person has no middle name
print(middle_name)  # Prints: None

# Functions that don't return anything give None
result = print("Hello")  # print() returns None
print(result)  # Prints: None
```

# Variables i

## Examples
Use meaningful names

- Declaration

```
salutation = "Hello"
name = "Monty Python"
pi = 3.14159
```

- Usage

```
print(name)
```

# Variables ii

## Variables and values can be combined

```
a = 3
b = 8.3
c = a + b
print(c)

salutation = "Hello"
name = "Monty Python"
print(salutation + " " + name) # String concatenation with +
```

# Variables iii

## Keywords - reserved words

You cannot name a variable with these names as they are protected by the language.

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
```

# Operators

Order of precedence (kind of like PEMDAS)

- ()
- **
- unary + -
- * / % //
- binary + -
- <, >, <=, >=, !=, ==
- **not**
- **and**
- **or**

# Comments

- Comments have no impact on the program
- Should explain the code
- A comment starts with a # character

## Examples

```
# Declaring the name
name = "Philipp"
print(name) # Prints Philipp
```

# Input and Output

# Input and Type Conversions i

Assume you want to program a calculator. In order to do this the user needs to be able to input his numbers and the program should be able to read this. Here comes the `input` function into play.

```
print("Add two numbers")
a = input("Please enter the first number ")
b = input("Please enter the second number ")

print("The result of the two numbers is: ")
print(a + b) # Something is wrong here
```

# Input and Type Conversions ii

Sometimes it is necessary to convert a variable from one data type to another (if possible). If you read data from a file into python, at first all the data is interpreted as strings even if your file only contains numbers.

In order to do mathematical analysis on these numbers, they need to be converted to the appropriate number type first.

- `int("32")`: Converts a string that holds a number to an integer
- `int("Hello")`: This doesn't work and it will throw a ValueError exception
- `float("313.333")`: Converts a string that holds a number to a float
- `str(32)`: Converts a number to a string

# Input and Type Conversions iii

Thanks to the f-string there is at least one less need to use explicit conversion functions:

## Examples

```
a = 20
b = 10
res = a + b
print(f"The sum of {a} and {b} is {res}")
# used to look like this:
print("The sum of " + str(a) + " and " + str(b) + " is " + str(res))
```

# Naming Conventions & Debugging

# Naming Conventions i

## How to name your functions and variables (PEP8)

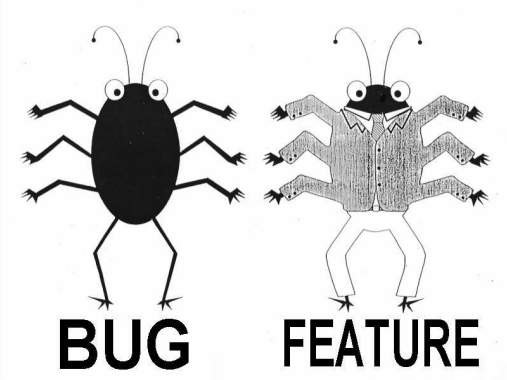
- Naming convention is a set of rules for choosing names of functions and variables
- Every programming language has different naming conventions
- Python
  - No spaces in variable and function names
  - Variable and function names are in lowercase and _ is used to separate words

```
length_in_cm = 15

def say_hello():
  print("Hello")
```

# Debugging i

## Finding and resolving ”bugs”




- Programming is a complex activity
- Mistakes happen all the time
- A mistake made in programming is called a bug
- The process of finding and resolving bugs is called debugging

# Debugging ii

## Errors

- Syntax error
  - Incorrect syntax of a statement: `print(Hello World)` instead of `print("Hello World")`
- Runtime error
  - Error that occurs during the execution of a program
  - e.g. division by 0
- Semantic errors
  - Program does not deliver correct results
  - No error messages (code is syntactically correct)
  - Fixing semantic errors can be extremely complicated (good software design is important)

# Debugging iii

## Techniques

- Reading code
- Print variables with `print()` to examine values (a poor man's debugger)
- Go through the program step by step -> **Debugger**!

# Conditionals: if/else/elif

# Conditionals i

- Boolean algebra is a part of mathematics
- Often used in programming
- A boolean expression is either true or false

```
5 == 5 # --> True
5 == 6 # --> False
6 > 4 # --> True
5 >= 8 # --> False
```

# Conditionals ii

## Examples

### if

- The expression if defines a condition
- If the condition is true, subsequent statements will be executed
- If the condition is false, subsequent statements will not be executed
- There has to be at least one statement after the condition

```
x = 10
if x > 0:
  print(f"{x} is positive")
if True:
  # This statement will always be executed
  print("Yes")
if False:
  # This statement will never be executed
  print("No")
```

# Conditionals iii

## else

- Expression else is executed if the if condition is false
- Can only be used in combination with an if expression

```
if x == 0:
  print("x is zero")
else:
  print("x is not zero")
```

# Conditionals iv

## Examples

### %-operator (remainder after division)

```
if x % 2 == 0:
  print("The number is even")
else:
  print("The number is odd")
```

# Conditionals v

## Chained conditionals

- elif is used to combine multiple conditions
- The else expression is executed when neither if nor any of the elifs is true.
- Any number of elif expressions can be used but only one if and one else

# Conditionals vi

## Examples

```
if x < y:
  print(f"{x} is less than {y}")
elif x > y:
  print(f"{x} is greater than {y}")
else:
  print(f"{x} and {y} are equal")
```

```
# Python 3
answer = input("Do you like Python?")
if answer == "yes":
  print("That is great!")
else:
  print("That is disappointing!")
```

# Conditionals i

## Nested conditionals

- Conditionals can be nested

```
if x > 0:
  if x < 10:
    print("x is a positive single digit")
```

## and

- Deep nesting can be difficult to read
- Use and to combine conditionals

# Conditionals ii

```
if x > 0:
  if x < 10:
    print("x is a positive single digit")
# is the same as
if x > 0 and x < 10:
  print("x is a positive single digit")
```

or

- At least one statement must be true for the condition to be true

```
if x > 0 or x < 0:
  print("x is not zero")
```

# Conditionals iii

## not

- Negation, inverts the boolean.
- not True -> becomes False
- not False -> becomes True

```
if not (y == 0):
  print(x/y)
else:
  print("Cannot divide by zero")
```

| X | Y | X and Y | X or Y |
|---|---|---|---|
| False | False | False | False |
| False | True | False | True |
| True | False | False | True |
| True | True | True | True |

# Exercise 1

Solve exercise 1

# Exercise 2

Solve exercise 2

# Lists: []

# Lists i

- Lists are a data type
- Lists are used in most programming languages (arrays)
- Lists are a set of values

```
list_a = [1, 2, 4]
list_b = ["Monty", "Python"]
```

# Lists ii

## Creating lists

- The easiest way to create a list is using []

```
numbers = [10, 12, 14, 19]
words = ["spam", "bungee", "swallow"]
```

- Data types can be mixed

```
my_list = ["music", 2000, 3.5, True]
```

# Lists iii

## Creating lists

- Since numbers are often stored in a list, there is a special method for doing so
- With only one argument, range returns a number series starting at 0

```
list(range(4))
# returns [0, 1, 2, 3]
```

- When using two arguments it is possible to define the start and end of the range [start, end) (end is not included in the list)

```
list(range(1,5))
# returns [1, 2, 3, 4]
```

## Creating lists

- The step size can be defined with a third argument

```
list(range(1, 10, 2))
# return [1, 3, 5, 7, 9]
```

- An empty list can also be created

```
empty_list = []
```

- This is often done when the values to be inserted in the list are not yet known.

# Lists v

## Creating lists

- Accessing elements can be done with the [] operator

```
names = ["Anna", "Tom", "Ralph", "Peter"]
print(names[1])
# prints Tom
```

**Important**
Array indices start at 0!

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| Anna | Tom | Ralph | Peter |

## Accessing lists

- A negative index is used to access the list from the end

```
names = ["Anna", "Tom", "Ralph", "Peter"]
print(names[-1])
# prints Peter
```

## Length

- The number of elements in a list can be obtained using the `len()` function

```
names = ["Anna", "Tom", "Ralph", "Peter"]
print(len(names))
# prints 4
```

## Out of range

- If there is no item in the list at the desired index, Python will print an error message

```
names = ["Anna", "Tom", "Ralph", "Peter"]
n_names = len(names)
print(names[n_names])
# IndexError: list index out of range
```

# Lists viii

## Changing elements in a list

- An element can be changed using [INDEX]

```
names = ["Anna", "Tom", "Ralph", "Peter"]
names[0] = "Alice"
# ["Alice", "Tom", "Ralph", "Peter"]
```

## Adding elements

- The `append()` method can be used to add an element at the end of the list

```
numbers = list(range(5))
# [0, 1, 2, 3, 4]
numbers.append(5)
# [0, 1, 2, 3, 4, 5]
```

# Lists x

## Concatenate lists

- The + operator can be used to join lists

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
# [1, 2, 3, 4, 5, 6]
```

## Slices

- Lists can be cut into slices
- The operator [n:m] returns a list of the elements that start at index n and stop before m

```
my_list = ["a", "b", "c", "d", "e", "f"]
my_list[1:3]
# ["b", "c"]
```

## Slices

- If the first index is empty, the slice starts at the beginning

```
my_list = ["a", "b", "c", "d", "e", "f"]
my_list[:4]
# ["a", "b", "c", "d"]
```

- If the second index is empty, the slice will include elements until the end of the list

```
my_list = ["a", "b", "c", "d", "e", "f"]
my_list[3:]
# ["d", "e", "f"]
```

## Deleting elements

- The `del()` method deletes items from the list

```
list_a = ["one", "two", "three"]
del(list_a[1])
# ["one", "three"]
list_b = ["a", "b", "c", "d", "e", "f"]
del(list_b[1:5])
# ["a", "f"]
```

# Immutables: Tuples () and Strings

# Tuples i

## Tuples is an immutable sequence data type

- It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.
- Tuples are declared using () instead of []

```
tuple = ("a", "b", "c", "d", "e")
```

- Tuples containing only one element (singleton) must have a comma at the end of the definition

```
tuple = ("a", )
```

# Strings i

## Strings are immutable

- Unlike lists, strings cannot be changed
- Operations on strings always return a modified copy of the string
- The original string remains unchanged

```
greeting = "Hello, world!"
greeting[0] = "J"
# TypeError: 'str' object does not support item assignment
```

# Iteration: for/while

# Iterations i

- Iterations are used to repeat statements
- There are two expressions for iterations
  - while
  - for

## while

- As long as the condition of the while loop is True, the body of the loop gets executed

## Example

```
while n > 0:
  print(n)
  n = n - 1
print('Lift off!')
```

# Iterations ii

## while

- If the condition is False at the beginning, the body of the loop is never executed
- If the variable that is used to check the condition of the while loop does not change, the loop will never terminate -> infinite loop
- Whether a while loop terminates can be hard to determine

```
while n != 1:
  print(n)
  if n % 2 == 0:
    n = n / 2
  else:
    n = n * 3 + 1
```

# Iterations iii

## while

- A while loop can be used to iterate through a list

```
names = ["Tom", "Anna", "Christopher"]
index = 0
while index < len(names):
  name = names[index]
  print(name)
  index = index + 1
```

# Exercise 3

Solve exercise 3

# Exercise 4

Solve exercise 4

# Iterations

## for

- Since it is often necessary to operate through lists and other data types, there is a special expression for this

```
for element in element_list:
  print(element)
```

# Exercise 5

Solve exercise 5

# Functions

# Built-in Functions i

Function are self contained modules of code that accomplish a specific task. They usually ”accept” input data and ”return” a result.

```
name = "Some name"
print(name) # Some name is used inside the print function -> the print
      function accepts the input and prints it to the console
```

- Functions can (and often do) also return a result (but the print function does not)

## Examples

```
text = "Python programming language"
print(text) # Prints: Python programming language
text_length = len(text) # This function returns the length of the text
print(text_length) # Prints length of the string
```

# Built-In Functions

https://docs.python.org/3/library/functions.html

# Functions, First Library

## Library import

```
import math
log_res = math.log(17.0)
sin_res = math.sin(45) # ??
```

- http://docs.python.org/library/math.html

# Functions, First Library

## Library import

```
import math
log_res = math.log(17.0)
sin_res = math.sin(45) # WRONG (well, not really, but not what we want)

sin_res = math.sin(math.radians(45))  # cos/sin etc take radians as
    arguments -> conversion from degree to radians necessary
```

- http://docs.python.org/library/math.html

# User-defined Functions i

## User-defined functions

- A function encapsulates some functionality
- Reduces complexity

```
def print_two_values(param1, param2):
  print(param1)
  print(param2)
```

- Syntax is important
  - Indentation
  - The colon

# User-defined Functions ii

## Examples

```
def line_separator():
  print("")

print("First Line")
line_separator()
print("Second Line")
line_separator()
print("Third Line")
line_separator()
print("Fourth Line")
```

- If we want to change the line separator to a dashed line we only need to change a single line of code

```
def line_separator():
  print("------------------------------")
```

# User-defined Functions iii

## Examples

- If the line separator should output two lines we can define a new function that calls the `line_separator()` function twice

```
def two_lines():
  line_separator()
  line_separator()

print ("First Line")
two_lines()
print("Second Line")
```

# User-defined Functions iv

## Parameters and arguments

- Arguments are passed when calling a function
- Value of arguments is assigned to parameters

```
def print_sum(number_1, number_2):
  result = number_1 + number_2
  print(result)

print_sum(1, 3)
print_sum(10, 5)
```

# User-defined Functions v

## Parameters and arguments

- Parameters are variables valid within the scope of the function
- Variables that are defined in a function can only be seen inside that function
- Scope can be identified by indentation

```
def concatenation(param1, param2):
  concat = param1 + param2
  print(concat)

concatenation("Hello", "World")
print(concat) # NameError: name 'concat' is not defined
```

# User-defined Functions vi

## Conclusion

- A function can be called multiple times
- If some code can be reused, put it in a function so you need to write less code
  - Higher factorization
  - Less redundancy
  - Better maintenance
- Functions can also call other functions

# Functions with return value i

- Some functions will return a value

```
# Python 3
answer = input("Do you like Python?")
```

- Our previously defined functions have never returned anything, but only printed something out

# Functions with return value  ii

## return

- Functions that return a value use the return keyword

```
import math
def area(radius):
  result = math.pi * radius ** 2
  return result

print(area(10))
my_circle_area = area(8)
```

- Functions can return any valid data type

# Exercise 6

Solve exercise 6

# Dictionaries: {}

# Dictionaries i

## Key-Value pair

- Dictionaries are similar to lists but have a key and value for each entry

# Dictionaries ii

## Creating dictionaries

- Dictionaries are created using {}

```
eng2de = {}
eng2de["one"] = "eins"
eng2de["two"] = "zwei"
```

- Values can be added directly

```
inventory = {
"apples": 430,
"bananas": 312,
}
```

# Dictionaries iii

## Accessing entries

- Values can be accessed directly using dictionary[”key”]

```
inventory = {
"apples": 430,
"bananas": 312,
}
print(inventory["apples"])
# 430
```

# Dictionaries iv

## Assigning and modifying values

- The key is assigned a value
- If the key already exists the existing value is overwritten

```
inventory = {
"apples": 430,
"bananas": 312,
}
inventory["oranges"] = 530
inventory["bananas"] = 250
print(inventory["bananas"])
# 250
```

## Deleting entries

- Key-Value pairs can be deleted using the `del()` function

```
inventory = {
"apples": 430,
"bananas": 312,
}
del(inventory["bananas"])
```

# Dictionaries vi

## Number of entries

- The `len()` function returns the number of entries

```
inventory = {
  "apples": 430,
  "bananas": 312,
}
len(inventory)
# 2
```

# Dictionaries vii

## Checking if an entry exists

- The in keyword can be used to check if a key exists in a dictionary

```
inventory = {
  "apples": 430,
  "bananas": 312,
}
if "apples" in inventory:
  inventory["apples"] += 100
else:
  inventory["apples"] = 100
```

# Dictionaries viii

## Iterating over entries

- The `items()` function combined with the for statement can be used to iterate through every key-value pair

```
for (my_key, my_value) in my_dict.items():
  print(my_key + ' : ' + my_value)
```

# Exercise 7

Solve exercise 7

# Functions with return value i

## Boolean return values

- The functions can return a boolean value (True, False)
- The function name should be formulated as a yes/no question

```
def is_divisible(x, y):
  return x % y == 0
```

# Functions with return value  ii

## Boolean return values

- The return value can be used in a condition

```
if is_divisible(x, y):
    print(f"{x} is divisible by {y}")
else:
    print(f"{x} is not divisible by {y}")
```

# Exercise 8

Solve exercise 8

# Persistence

# Persistence

- So far no data has been saved in any of our examples
- All data was deleted from the memory as soon as our examples quit
- There are several ways to permanently store data on the hard disk
  - Database
  - Simple text files

# Files i

## Common procedure

- Open file
- Do something with the file
- Close the file - depending on syntax done automatically.

```
from pathlib import Path

file_path = Path('my_file.txt')
with file_path.open("mode") as file:
  # do some stuff
```

# Files ii

## Why pathlib?

- Modern, object-oriented approach to file handling (Python 3.4+)
- Cross-platform path handling (works on Windows, Mac, Linux)
- More intuitive methods for common file operations
- Better path manipulation with / operator

```
from pathlib import Path

# Create paths easily
data_dir = Path('data')
file_path = data_dir / 'my_file.txt'

# Check if file exists
if file_path.exists():
  print(f"File exists: {file_path}")
```

# Files iii

## Different modes

- The mode defines how the content of the file should be treated
- Modes
- ”r”: read only
- ”w”: write only
- ”r+”: read and write
- ”a”: append

```
from pathlib import Path

file_path = Path('my_file.txt')
# open a file in read mode
with file_path.open("r") as file:
  # do some stuff
```

# Files iv

## Write

- The write_text() method is a convenient way to write strings
- The open() method with ”a” mode can be used for appending
- ’\n’ is used to insert a line break

```
from pathlib import Path

file_path = Path('my_file.txt')

# Simple write (overwrites file)
file_path.write_text('Hello World\n')

# Append mode
with file_path.open("a") as file:
  file.write('First line of the write operation\n')
  file.write('This is a line with a new-line character at the end\n')
```

# Files v

## Read

- Path.read_text() reads the entire file as a string
- A for loop can be used to read a file line by line
- `line.strip()` removes the trailing '\n'

```
from pathlib import Path

file_path = Path('my_file.txt')

# Read entire file
content = file_path.read_text()
print(content)

# Read line by line
with file_path.open("r") as file:
  for line in file:
    line = line.strip()
```

# Files  vi

```
    print(line)
```

# JSON i

## Dictionaries/list in JSON

- file.write() only accepts strings as arguments
- If complex structures such as dictionaries or lists should be stored in a file, it's necessary to convert these structures into strings first
- An example of a standard used for this purpose is JSON (Javascript Object Notation)

```
import json
my_dict = {"one": "eins", "two": "zwei"}
my_dict_as_string = json.dumps(my_dict)
print(my_dict_as_string)
```

# JSON ii

## Convert JSON to dictionaries/lists

- Example of a string in JSON that is converted into a dictionary

```
import json
my_dict_as_string = '{"two": "zwei", "one": "eins"}'
my_dict = json.loads(my_dict_as_string)
print(my_dict)
```

# JSON iii

## Combining pathlib with JSON

- Path makes it easy to save and load JSON files
- write_text() and read_text() work perfectly with JSON

```python
import json
from pathlib import Path

# Save dictionary to JSON file
my_dict = {"one": "eins", "two": "zwei", "three": "drei"}
file_path = Path('data.json')
file_path.write_text(json.dumps(my_dict, indent=2))

# Load dictionary from JSON file
loaded_dict = json.loads(file_path.read_text())
print(loaded_dict)
```

# Exercise 9

Solve exercise 9

# Additional Resources

- How to Think Like a Computer Scientist from Allen Downey, Jeffrey Elkner, and Chris Meyers
- Learning with Python: Interactive Edition 2.0
  - `http://interactivepython.org/courselib/static/thinkcspy/index.html`
- Official Python Documentation
  - `http://www.python.org/doc/`
- Project Euler: Mathematical problems that can be solved programmatically
  - `http://projecteuler.net/`
- Platforms to prepare for coding interviews
  - `https://leetcode.com`
  - `https://www.interviewbit.com/`