In [ ]:
```
# get formating done automatically according to style `black`
#%load_ext lab_black
```

Python core language part 1

Python is an *Interpreted*, *high-level*, *general-purpose programming language*

- Interpreted: source code executes directly
- High-level: strong abstraction from the details of the computer
- General-purpose: widest variety of applications
- Programming language: language which holds a set of instructions that produce various kinds of output

Content

You will:

- Write your first line of code
- Understand the syntax
- Learn about variables ...
- ... and data types
- Work with strings
- Create and use lists, tuples and sets
- Dive into dictionaries

Your first Line of Code

Simply type the follwing in the empty cell below and execute the code. You execute the code by pressing the `Control + Enter` key or click the `Run` button above.

```
print("Hello world.")
```

In [1]:
```
# Exercise
print("Anything")
print("another level")
```

```
Anything
another level
```

Syntax

- Python is cASe SensItiVe.

```
Print("Hello world.")
```
will not work.

- You can comment a line of code with `#` .

```
# Commented lines will not be executed. They are important
features for the human reader of the code.
```

- In Python, blocks of code are expressed by their indentation. This keeps the code clean and tidy.

```
print("one indentation level")
print("this indentation level again")

    print("another indentation level")
```
will lead to `IndentationError: unexpected indent`

- If a line of code becomes to large, you can simply contiue in the following line

```
print("Python is an interpreted, high-level and general-purpose
programming language. "
      "Python's design philosophy emphasizes code readability
with its notable use of significant "
      "indentation. Its language constructs and object-oriented
approach aim to help programmers "
      "write clear, logical code for small and large-scale
projects.")
```

Exercise

Play around with different indentation levels and divide a line of code over two lines. Comment your code using `#` .

In [2]:
```
# Exercise
print("Python is an interpreted, high-level and general-purpose programmi
      "Python's design philosophy emphasizes code readability with its no
      "indentation. Its language constructs and object-oriented approach
      "write clear, logical code for small and large-scale projects.")
```

```
Python is an interpreted, high-level and general-purpose programming lang
uage. Python's design philosophy emphasizes code readability with its not
able use of significant indentation. Its language constructs and object-o
riented approach aim to help programmers write clear, logical code for sm
all and large-scale projects.
```

Variables

A **variable** is a symbolic name, which contains information referred to as a value. A variable is created with an "assignment" equal sign `=`, with the variable's name on the left and the value it should store on the right.

```
x = 5
```

In the exapmple above, the assignment `x = 5` sets `x` point to `5`.

The Python language reserves a small set of keywords that have a special language functionality. No object can have the same name as a reserved word. Let us see what happens in the following example:

In [3]:
```
class = 5
```

```
  Cell In[3], line 1
    class = 5
          ^
SyntaxError: invalid syntax
```

You can see the keyword list any time by typing `help("keywords")`

In [4]:
```
help("keywords")
```

```
Here is a list of the Python keywords.  Enter any keyword to get more hel
p.

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

Programmers are not mathematicians:

In [5]:
```
x = 5
y = x
print(y)
```

```
5
```

```
x = 0
print(y)
> ?
```

In [6]:
```
x = 0
print(y)
```

```
5
```

I mathematician would `y` expect to be `0` (the same value as `x`). However, in Python the `=` sign does not set up a permanent relationship between two variables. The assignment `y = x` sets `y` to point to the same thing as `x`. In the beginning, `x` and `y` still point to the same value (`5`). Later, `x` points to a new value (`0`), whereas `y` still points to `5`.

In Python `=` reads as **point to**.

Data Types

Basic **data types** are *numeric* (integer, floating point, complex), *string* and *Boolean*

- Integers are all whole numbers: e.g. -4, -6, 2, 5
- Floating point numbers are real numbers: e.g. -4.2321, -0.1, 5e-10, 10.01
- Complex numbers: e.g. 3+5j is written as `complex(3,5)`
- Strings contain text: e.g. "This is a String"
- The Boolean data type can either be True or False

In [7]:
```
integer_var = -3
type(integer_var)
```

Out[7]:
```
int
```

In [8]:
```
real_number_var = 3.
type(real_number_var)
```

Out[8]:
```
float
```

In [9]:
```
string_var = "I'm not a string."
type(string_var)
```

Out[9]:
```
str
```

In [10]:
```
boolean_var = True
type(boolean_var)
```

Out[10]:
```
bool
```

In [11]:
```
comp = complex(3,5)
type(comp)
```

Out[11]:
```
complex
```

One data type can also be converted into another:

In [12]:
```
str_num = "123"
int_num = int(str_num)
type(int_num)
```

Out[12]:
```
int
```

Exercise

1. Create a variable of each type you have learned.
2. Douple check the type of your created varibale by using `type(<var>)` .
3. What happens if you add 2 variables of different types? In which cases does it work and in which does it not?

In [13]:
```
# 1.
```

repeat example with your own variable names

In [14]:
```
# 2.
```

repeat example with own variabel names

In [15]:
```
# 3.

# Float + Integer works
print(3.2 + 3)

# String + String works
print("hello " + "world")

# Boolean + Boolean works
print(True + True)
```

```
6.2
hello world
2
```

In [16]:
```
# String + Int does not work!

print("Hello" + 3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
Cell In[16], line 3
      1 # String + Int does not work!
----> 3 print("Hello" + 3)

TypeError: can only concatenate str (not "int") to str
```

String Methods

Python has a set of built-in methods that you can use on strings. A good overview can be found on W3School.

In the following a few examples are presented:

In [17]:
```
# Define a test string
test_string = "This is 'my' test string."
```

In [18]:
```
test_string
```

Out[18]:
```
"This is 'my' test string."
```

In [19]:
```
# Converts a string into upper case
test_string.upper()
```

Out[19]:
```
"THIS IS 'MY' TEST STRING."
```

In [20]:
```
# Splits the string at the specified separator, and returns a list
test_string.split(" ")
```

Out[20]:
```
['This', 'is', "'my'", 'test', 'string.']
```

In [21]:
```
# Returns a string where a specified value is replaced with a specified v
test_string.replace("i", "!")
```

Out[21]:
```
"Th!s !s 'my' test str!ng."
```

Exercise

Create various string variables and apply three different **string methods**.

In [22]:
```
# Exercise
```

repeat example with different functions

Lists, Tuples and Sets

A **list** can contain any type of variable and as many variables as you wish.

It is initialized by square brackets `my_list = [item1, item2, item3, ...]` .

If you are unsure about the content you can also declare an empty list with `empty_list = []` .

In [23]:
```
my_list = [3, 8.01, 'beer']
print(my_list)
```

```
[3, 8.01, 'beer']
```

You can append items with `my_list.append(var)`

In [24]:
```
my_list.append('another beer')
```

You can access a list item with `my_list[<item index>]`

In [25]:
```
my_list[2]
```

Out[25]:
```
'beer'
```

With **slicing** you can create a new list as a subset of the old list.

In [26]:
```
old_list = [0, 1, 2, 3, 4]
new_list = old_list[1:3]
print(new_list)
```

```
[1, 2]
```

In [27]:
```
another_list = ["Apple", "Cherry", "Banana", "Orange", "Pineapple", "Mang
smaller_list = another_list[1:3]
print(smaller_list)
```

```
['Cherry', 'Banana']
```

Exercise

1. Create your own list.
2. Add another item to your list.
3. Access an item in your list via its index.

In [28]:
```
# 1.
```

repeat example

In [29]:
```
# 2.
```

repeat example

In [30]:
```
# 3.
```

repeat example

A **tuple** is a collection which is ordered and unchangeable.

It is initialized with round brackets `my_tuple = (var1, var2, var3, ...)`

In [31]:
```
my_tuple = (1, "test", 3.)
type(my_tuple)
```

Out[31]:
```
tuple
```

In [32]:
```
print(my_tuple)
```

```
(1, 'test', 3.0)
```

Exercise

1. Create a tuple in which one item is a subset of your list.
2. Access an item in your tuple via its index.
3. Can you add an item to your tuple?

In [33]:
```
# 1.
my_tuple = (1, "test", 3.)
```

In [34]:
```
# 2.
my_tuple[2]
```

Out[34]: 3.0

In [35]:
```
# 3.
my_tuple.append("new item")
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call las
Cell In[35], line 2
      1 # 3.
----> 2 my_tuple.append("new item")

AttributeError: 'tuple' object has no attribute 'append'
```

A **set** is a collection which is unordered and unindexed. It does not allow duplicate members. A set is initialized with curly brackets `my_set = {item1, item2, item3, ...}`

In [36]:
```
my_set = {"water", "beer", "wine"}
type(my_set)
```

Out[36]: set

Exercise

1. Create a set.
2. Access an item in your set via its index.
3. Can you add an item to your set?

In [37]:
```
# 1.
my_set = {"water", "beer", "wine", "water"}
my_set
```

Out[37]: {'beer', 'water', 'wine'}

In [38]:
```
# 2.
my_set[2]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
Cell In[38], line 2
      1 # 2.
----> 2 my_set[2]

TypeError: 'set' object is not subscriptable
```

In [39]:
```
# 3.
my_set.add("test")
```

Dictionaries

A **dictionary** stores data in *key* and *value* pairs. It allows access of values via an unique key. It is a collection which is ordered, changeable and does not allow duplicates.

Creation

There are several methods to initialise a dictionary; use whatever fits your needs best.

In [40]:
```
a = {"k":1000, "hello":"world", 42:["hello", "world"]}
b = dict([("k", 1000), ("hello","world"), (42,["hello", "world"])])
c = dict({"k":1000, "hello":"world", 42:["hello", "world"]})
d = dict(zip(["k", "hello", 42], [1000, "world", ["hello", "world"]]))
```

In [41]:
```
a
```

Out[41]:
```
{'k': 1000, 'hello': 'world', 42: ['hello', 'world']}
```

All four dictionaries are equal.

In [42]:
```
a == b == c == d
```

Out[42]:
```
True
```

In [43]:
```
a
```

Out[43]:
```
{'k': 1000, 'hello': 'world', 42: ['hello', 'world']}
```

Accessing elements

Access a value of the dictionary via its key

In [44]:
```
a["k"]
```

Out[44]:
```
1000
```

In [45]:
```
a[42]
```

Out[45]:
```
['hello', 'world']
```

You can get a list of all keys or values

In [46]:
```
a.keys()
```

Out[46]:
```
dict_keys(['k', 'hello', 42])
```

In [47]:
```
a.values()
```

Out[47]:
```
dict_values([1000, 'world', ['hello', 'world']])
```

Modify dictionary

Dictionaries are by default mutable, therefore you can add new keys, overwrite existing ones or delete them.

In [48]:
```
a["New key"] = 23
a["hello"] = "welt!"
a.pop("k")
```

Out[48]:
```
1000
```

In [49]:
```
a
```

Out[49]:
```
{'hello': 'welt!', 42: ['hello', 'world'], 'New key': 23}
```

Watch out, if a key does not exist. Use *get* or *pop* method to be safe (pop removes key).

In [50]:
```
a["key does not exist"]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call las
Cell In[50], line 1
----> 1 a["key does not exist"]

KeyError: 'key does not exist'
```

In [51]:
```
a.get("key does not exist", "This key does not exist")
```

Out[51]:
```
'This key does not exist'
```

In [52]:
```
del a["hellofdhdf"]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call las
Cell In[52], line 1
----> 1 del a["hellofdhdf"]

KeyError: 'hellofdhdf'
```

In [53]:
```
a.pop("key does not exist", "This key can't be deletet, because it does n
```

Out[53]:
```
"This key can't be deletet, because it does not exist"
```

Dictionaries can also be nested

In [54]:
```
a["Nested"] = {1: "world!", 2: "Hello "}
print(a["Nested"][2] + a["Nested"][1])
```

```
Hello world!
```

Exercise

1. Create a new dictionary with the key-value pairs 1: "eins", "M": "Mega", "liste": [1,2,3]
2. Add a new entry, where either the key or value is a tuple
3. Remove the key 1 from the dictionary
4. Append number 4 to the list, which is associated with key "liste". Print the content of your dictionary afterwards

In [55]:
```
# 1.
exercise_dict = {1: "eins", "M": "Mega", "liste": [1,2,3]}
exercise_dict
```

Out[55]:
```
{1: 'eins', 'M': 'Mega', 'liste': [1, 2, 3]}
```

In [56]:
```
# 2.
exercise_dict["tuple"] = (1,2,3)
exercise_dict
```

Out[56]:
```
{1: 'eins', 'M': 'Mega', 'liste': [1, 2, 3], 'tuple': (1, 2, 3)}
```

In [57]:
```
# 3.
exercise_dict.pop(1)
exercise_dict
```

Out[57]:
```
{'M': 'Mega', 'liste': [1, 2, 3], 'tuple': (1, 2, 3)}
```

In [58]:
```
# 4.
exercise_dict['liste'].append(4)
exercise_dict
```

Out[58]:
```
{'M': 'Mega', 'liste': [1, 2, 3, 4], 'tuple': (1, 2, 3)}
```