[Table of contents (../toc.ipynb)](../toc.ipynb)

Built in data types and structures

Python ships with

- Bool (can be `True` or `False`)
- String
- Number
    - Integer (signed integers)
    - Float (floating point numbers in double precision)
    - Complex (complex numbers with real and imaginary part)
- List
- Tuple
- Set
- Dictionary

data types.

Operations with strings

You already know that strings are created with `""`.

You can add, slice, and repeat strings with simple operators.

If you slice, note that Python is zero based and the last element can be assessed with `-1`. Furthermore, note that the start is included while the end is excluded.

In [1]:

```
my_string = "Say something "

print(my_string[0:3])  # prints the first three letters
print(my_string[4:])  # prints the fifth to the last letter
print(my_string[-2])  # prints the second last letter
print(my_string + "- right now")  # appends "- right now"
print(my_string * 2)  # prints the string twice
```

```
Say
something
g
Say something - right now
Say something Say something
```

Numbers

Integers

- Integers are created with assignments like `a = 1` - the number has no decimal point.
- From Python 3 onwards, divisions with 2 integers are automatically converted to floats.
- Python integers are variable precision. Hence, overflow is not such a big issue as in C language.

In [2]:
```
6 / 2
```

Out[2]:
```
3.0
```

In [3]:
```
# if you want an integer, use floor
6 // 2
```

Out[3]:
```
3
```

Floats

- Floats are assigned with decimal point like `a = 3.14`.
- Floats can also be defined with exponential notation `a = 1e-3`, which means $1 \cdot 10^{-3}$.
- You can convert integers into floats with

In [4]:
```
a = 3
float(a)
```

Out[4]:
```
3.0
```

Fixed precision of floating point numbers

- Remember that numbers are actually stored in binary form in computers.
- The precision of floating point numbers is always limited, due to fixed number of bits.
- This drawback of floating point numbers is not Python specific, it will also occur in any computer.

In [5]:
```
# let's look a bit deeper in floats
a = 0.1
b = 0.2
# print a with 21 digits
print("{0:.21f}".format(a))
print("{0:.21f}".format(b))
```

```
0.100000000000000005551
0.200000000000000011102
```

In [6]:
```
# Hence, tests like this will fail
0.1 + 0.2 == 0.3
```

Out[6]:
```
False
```

Floats take aways

- Therefore, **never use exact equality tests for floats!**
- If you want to know more about floating point precision in Python, consult the [Python docu (https://docs.python.org/3/tutorial/floatingpoint.html)](https://docs.python.org/3/tutorial/floatingpoint.html).

In [25]:

```
# Approximate comparison of floats was added in Python 3.5
from math import isclose as close

a = 5.0
b = 4.99998
close(a, b, abs_tol=0.00003)
```

Out[25]:

```
True
```

In [26]:

```
# Tune the tollerance
close(a, b, abs_tol=0.00001)
```

Out[26]:

```
False
```

In [27]:

```
# And now this works with default settings
close(0.1 + 0.2, 0.3)
```

Out[27]:

```
True
```

Complex numbers

Complex numbers have real and imaginary part and can be assigned with

In [28]:
```
a = complex(1, 3)
```

In [29]:
```
# or
a = 1 + 3j
```

You can compute real, imaginary part as well as magnitude and conjugate with

In [31]:
```
a.real
```
Out[31]:
```
1.0
```

In [32]:
```
a.imag
```
Out[32]:
```
3.0
```

In [33]:
```
abs(a)
```
Out[33]:
```
3.1622776601683795
```

In [35]:
```
a.conjugate()
```
Out[35]:
```
(1-3j)
```

Lists

- Lists are created with brackets `[]` and may contain entries on different type.
- The entries as well as the shape (length) of lists may change (are *mutable*).
- Lists are *ordered*.

In [118]:
```
x = [0, 1, 2, 3, 4, 5]

# slice from index zero to 2
print(x[0:3])

# append one element
x.append(6)
print(x)

# remove first occurance of the the value
x.remove(2)
print(x)

# remove and returns item for given index
x.pop(0)
print(x)

# This is just a snapshot, there are many more methods for lists.
```

```
[0, 1, 2]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 3, 4, 5, 6]
[1, 3, 4, 5, 6]
```

List comprehensions

- A nice special feature in python are list comprehensions.
- You can use them to create lists with loops.
- Usually `for` loops are used, which are covered in a later section.

- The syntax is `[expr for variable in iterable]`

  or if you require a condition `[expr for variable in iterable if condition]`

In [2]:
```
# Create a list from 0 to 9, where each entry is the negative of the index
[-i for i in range(10)]
```

Out[2]:
```
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

In [4]:
```
# Here with a condition
[-i for i in range(10) if i % 2 == 0]
```

Out[4]:
```
[0, -2, -4, -6, -8]
```

Tuple

- Tuples are created with parenthesis `()` or without anything like `a = 1, 2, 3`.
- They are *ordered* and *immutable*.
- Methods are very similar to lists.

In [59]:
```python
a = (0, 1, 2, 3)
a[0]
```

Out[59]:
```
0
```

In [70]:
```python
# Changing values is not allowed for tuples
try:
    a[3] = 15  # This does not work
except Exception as error:
    print(error)
```

```
'tuple' object does not support item assignment
```

Because tuples are *immutable*, they are very often used for multiple returns of functions.

In [39]:
```
# Here a function which returns a tuple
def my_func(a, b):
    return a**2, b**2


my_func(a=1, b=10)
```

Out[39]:
```
(1, 100)
```

Set

- Sets are collections of unique values.
- Sets are unordered.
- They are defined with curly brackets `{}`.
- You can do anything like with mathematical sets, such as union, intersection...

In [41]:
```python
set_one = {0, 1, 2, 3, 4}
set_two = {3, 4, 5, 6}

# union
set_one | set_two
# there is also a better readable way
set_one.union(set_two)
```

Out[41]:
```
{0, 1, 2, 3, 4, 5, 6}
```

In [42]:
```python
# intersection
set_one & set_two
# or with method
set_one.intersection(set_two)
```

Out[42]:
```
{3, 4}
```

Dictionaries

- Dictionaries are very flexible and well designed for data representation.
- They are unordered and mutable.
- Dictionaries have key and value field `a = {"key_one": value_one, "key_two": value_two}`.
- There are many methods for dictionaries, please read the [Python docu (https://docs.python.org/3/library/stdtypes.html)](https://docs.python.org/3/library/stdtypes.html).

In [43]:
```
a = {"temp": 300, "pressure": 15.78}
a["pressure"]
```

Out[43]:
```
15.78
```

In [44]:
```
# Read the key names
a.keys()
```

Out[44]:
```
dict_keys(['temp', 'pressure'])
```

In [45]:
```
# Add a new field
a["speed"] = 125
```

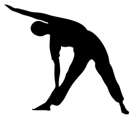
In [47]:
```
a.keys()
```

Out[47]:
```
dict_keys(['temp', 'pressure', 'speed'])
```

In [48]:
```
# Print all values
a.values()
```

Out[48]:
```
dict_values([300, 15.78, 125])
```

Exercise: Try dictionaries (15 minutes)



Please try out what you have seen by yourself.

To solve this task you have (at least) three options to run you code:

- Write a Python file and call it through ipython with `%run yourfile.py`.
- Use the interactive Jupyter Notebooks of this course on [Binder (https://mybinder.org/v2/gh/StephanRhode/py-algorithms-4-automotive-engineering/master)](https://mybinder.org/v2/gh/StephanRhode/py-algorithms-4-automotive-engineering/master).
- Use IPython cloud service [Python anywhere (https://www.pythonanywhere.com/try-ipython/)](https://www.pythonanywhere.com/try-ipython/) Note that you need the magic command `%cpaste` to be able to type multiple lines in IPython.

Here the task:

- Define a dictionary `a = {'scale': 3.6, 'speed': [0., 15., 22., 30.], 'label': "Scaled speed"}`
- Write a script which uses a for loop and prints the product of each scale and speed.

Hint

- Try at first to read the scale and the speed list. Print them.
- Then try to define a for loop which iterates over the list. (Google is your friend).

Solution

Please find one possible solution in `solution_data-types.py` (solution_data-types.py) file.

In [55]:
```
%run solution_data-types.py
```

```
Scaled speed 0.0
Scaled speed 54.0
Scaled speed 79.2
Scaled speed 108.0
```