

# Introduction to Programming in Python

Lucerne University of  
Applied Sciences and Arts

**HOCHSCHULE  
LUZERN**

# A few odds and ends

## More on strings

### String interpolation

- String interpolation is used to "paste" values (often numbers) into strings, e.g., for printing.
- In principle, the same result can be achieved using string concatenation. String interpolation just makes this easier.
- There are different methods in Python for doing this; the most common is `str.format`.

## Examples:

In [ ]:

```
mystr = "{} is {} years old.".format("Simon", 45)  
print(mystr)
```

In [ ]:

```
mystr = "{0} is {1} years old.".format("Simon", 45)  
print(mystr)
```

In [ ]:

```
mystr = "{name} is {age} years old.".format(name="Simon", age=45)  
print(mystr)
```

In [ ]:

```
mystr = "{name:} is {age:2.2f} years old.".format(name="Simon", age=45)  
print(mystr)
```

## str.join

- This is used for composing a single string out of several, given as, e.g., a list.
- Example:

In [ ]:

```
city_list = ["Zurich", "Lucerne", "Bern"]  
" and ".join(city_list)
```

This may be a bit counterintuitive as first: the separator goes into the string, and the list into the `join` method.

# Dictionaries

- Dictionaries, or `dict`s, are another built-in data type.
- They consist of key-value pairs, and are constructed with curly braces:

In [ ]:

```
population= {"Germany": 83, "Switzerland": 8, "Netherlands": 16}  
print(population)
```

Indexing:

In [ ]:

```
population["Germany"]
```

- In a way, you can think of them as a poor man's dataframe.
- The keys have to be of immutable type ( `string` as above, `int` , etc), and unique.
- The values can be anything.
- Looping loops the keys:

In [ ]:

```
for c in population:  
    print(c)
```

In [ ]:

```
for c in population:  
    print(population[c])
```

(The order in which elements appear is the order in which they were inserted)

Deleting elements:

In [ ]:

```
del[population["Germany"]]
```

Methods:

In [ ]:

```
print(', '.join(filter(lambda m: callable(getattr(population, m)) and not m.startswith("_"), dir(population))))
```

## Exercise

The following dictionaries are given:

In [ ]:

```
en_de = {"red" : "rot", "green" : "grün", "blue" : "blau", "yellow": "gelb"}  
de_fr = {"rot" : "rouge", "grün" : "vert", "blau" : "bleu", "gelb": "jaune"}
```

(example taken from <https://python-course.eu/python-tutorial/dictionaries.php>).

How can you translate "red" from English to French?

In [ ]:



# Comprehensions and generator expressions

## List comprehensions

- A "comprehension" in Python is a convenient way of constructing container types such as lists, dictionaries, etc.
- Lists are the most common use case.
- They look a bit like a for loop:

In [ ]:

```
[a**2 for a in range(0, 10)]
```

In [ ]:

```
[a**2 for a in range(0, 10) if a%2!=0]
```

# Generator expressions

- These look like list comprehensions, but use round brackets.
- The main difference is that they don't "materialize" a list; rather, they produce their elements on demand. This saves memory.
- Example

In [ ]:

```
(a**2 for a in range(0, 10))
```

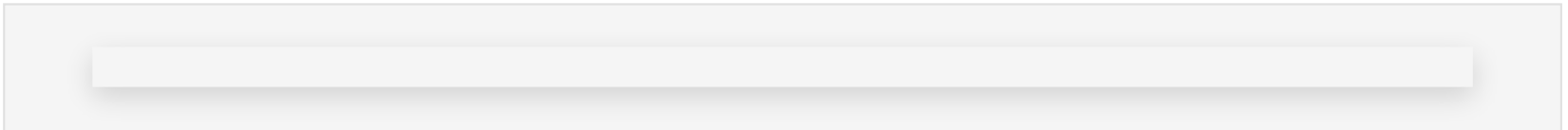
In [ ]:

```
mygen = (a**2 for a in range(0, 10))  
for elem in mygen:  
    print(elem)
```

## Exercise

- Create a list containing the odd numbers between zero and 50, using
  - a for loop
  - a list comprehension

In [ ]:



# Recommended reading

- [https://www.w3schools.com/python/ref\\_string\\_format.asp](https://www.w3schools.com/python/ref_string_format.asp)
- <https://python-course.eu/python-tutorial/dictionaries.php>
- <https://www.geeksforgeeks.org/python-list-comprehensions-vs-generator-expressions/>

# Homework

- <https://holypython.com/intermediate-python-exercises/exercise-1-python-format-method/>
- <https://holypython.com/intermediate-python-exercises/exercise-2-join-method/>
- <https://towardsdatascience.com/beginner-to-advanced-list-comprehension-practice-problems-a89604851313> 1-3