Prof. Dr. Gemma Roig
M.Sc. Alperen Kantarcı
M.Sc. Gamze Akyol

Programmieren für Studierende der Naturwissenschaften

Lecture 3 – Aggregated Data Types

Contents

- L1: Basics of programming
  P1: Exercise 1 and help to installments.

- L2: Elementary data types and control structures
  P2: Exercises

- L3: Aggregated data types
  P3: Exercises

- L4: Aggregated data types and functions
  P4: Exercises

- L5:Testing,error messages and self-help
  P5: Exercises

Review of Loops

The head-driven or pre-test loop:

- first the termination condition is checked before the loop body is run through (usually indicated by the keyword WHILE (=so long-until).

The foot-driven or re-checking loop (implementable with a trick inPython):

- only after the loop body has been run through, the termination condition is checked e.g. by a construct REPEAT-UNTIL (=repeat-to).

Counting Loop

Python syntax:

```
for <variable name/run index> in <sequence>:
    <instructions>

<Instructions after the loop>
```

While Loop

```
while <logical expression>:
   <instructions>
else:
   <instructions>

<expression after loop>
```

The "while statement" is used for repeated execution of the code block (suite), as long as the "expression" evaluates to "True".
If the logical expression evaluates to "False", the else branch statement will be executed if present (VERY RARE!)
In the statement, the truth value of the logical expression must be changed at some point, otherwise a danger of an infinite loop!

Loop Example

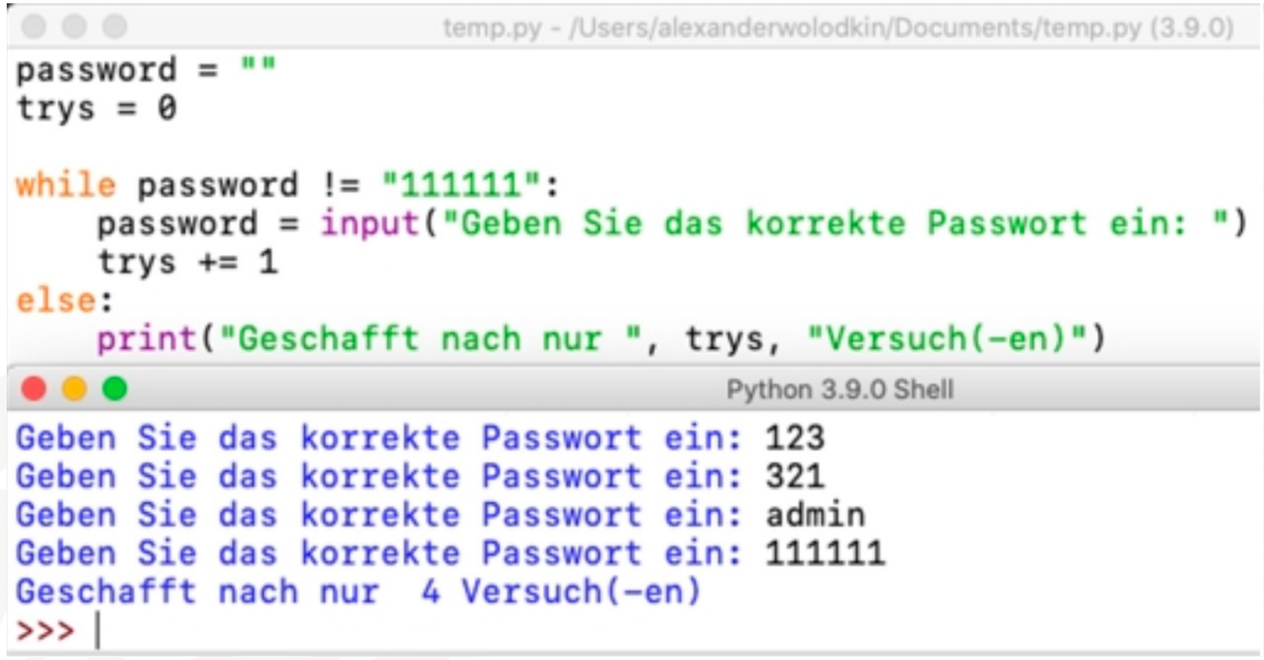
```
count = 0
while count < 9:
    print('The count is:', count)
    count = count + 1

print("Good bye!")
```

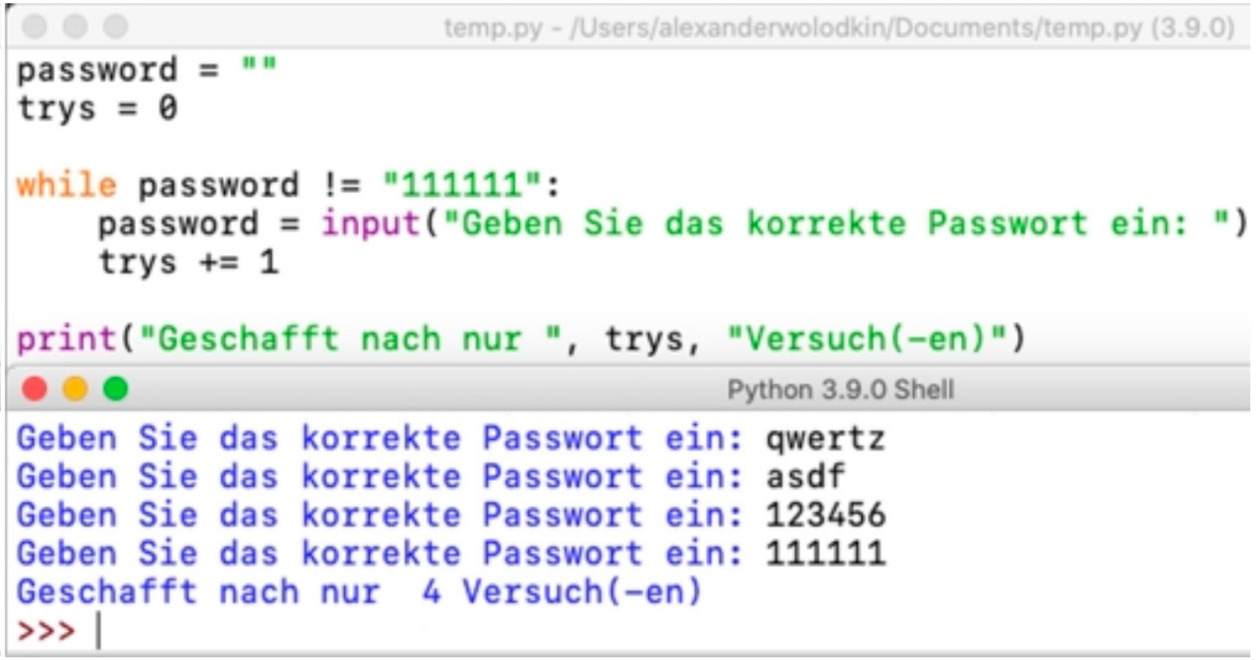
- A bad example for while loop. You can use for loop easily and more readable

```
var = 1
while var == 1 :
    num = raw_input("Enter a number  :")
    print("You entered: ", num)
        # This constructs an infinite loop
print("Good bye!")
```

- Infinite loop

Loop Example





String

- "String” is a datatype. A string is a character string and is written in quotes.
- Comparison operations on strings are possible, but not necessarily intuitive.

Indexing (strings and lists)

- Conceptual model of the string: characters are in "cells".
- These cells are numbered
- Each character is accessible under its own index `i` in square brackets: `s[i]`.

Slicing (strings and lists)

- Extracts contiguous (cell) ranges via indices. Result is again a string.
- Rule: `s[a:b]` extracts subrange from index a to b-1
  - `s[a]` is the first element in the subrange
  - `s[b-1]` is the last element in the subrange
- Example: `s[1:5]` goes from 1 to 5 **only**
- Rule: `s[a:]` goes from index a to the last element
- Example: `s[2:]` goes from index 2 to the last element

- Rule: `s[:]` creates a real (!) copy of `s`

Index ranges

Indexwerte außerhalb des Wertebereichs [0,11] oder [-12,-1] führen zu einem "IndexError".

```
>>> S[-12]
'E'
>>> S[-13]
Traceback (most recent call last):
File "<pyshell#204>", line 1, in …
IndexError: string index out of range
```

Anders **beim Slicing**, dort werden die mit roten Pfeilen markierten Indexwerte auf ± len(S) „gecasted"

```
>>> S[9:14]
'oll'
>>> S[-9:-12]
''
>>> S[-14:3]
'EPR'
```

Functions on string

- **len():** Outputs the number of characters in a string.
- Upper and lowercase: **.lower()** and **.upper()**
- Replace substring : **s.replace(<toReplace>,<replacestring>)**
- Count substring : **s.count(<tocount>)**

```
>>> my_string = "Hello World"
>>> len(my_string)
11
>>> my_string.upper()
'HELLO WORLD'
>>> my_string.lower()
'hello world'
>>> my_string.replace("World","Universe")
'Hello Universe'
>>> print(my_string)
Hello World
>>> my_new_string = my_string.replace("World","Universe")
>>> print(my_string, my_new_string, sep="\n")
Hello World
Hello Universe
>>> my_string.count('l')
3
>>>
```

Aggregated data types

- A data type (in computer science) is the combination of sets of objects with the operations defined on them

- Composite (aggregated) data types are data constructs that are composed of simpler (elementary) data types or composite data types.
  The focus here is on the operations.

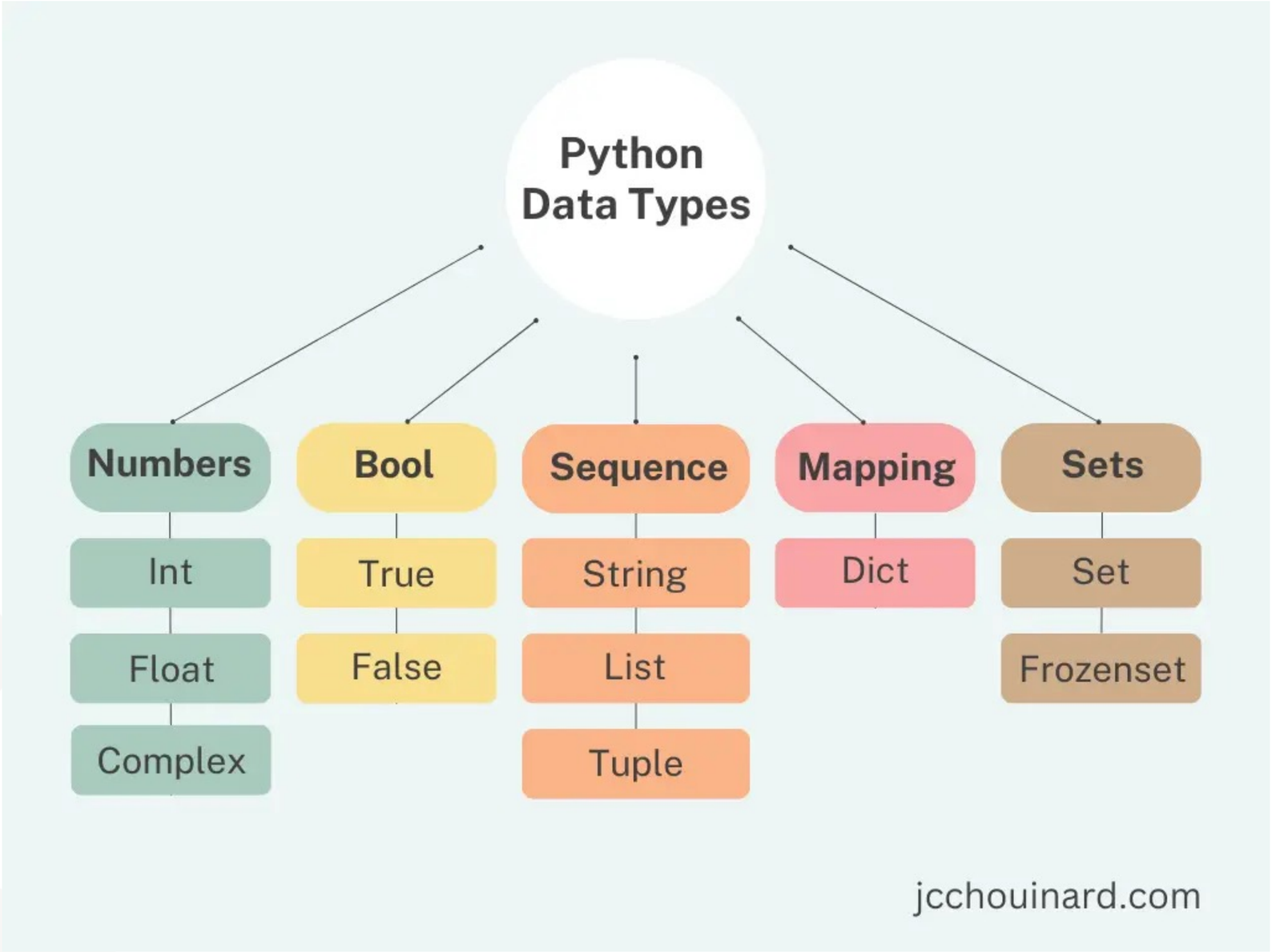
- Aggregated data types are available in most programming languages

- Programming languages can support aggregated data types at two levels:

  - It provides constructors that can be used to describe the structure of an aggregated data type

  - You have predefined aggregated data types (built-in in Python)

Aggregated data types



Mutable - Immutable

- Entries in a frozen set are immutable in contrast to a set.

```
a = frozenset([4,3,2,1])
print(a)

a[1] = 0
print(a)
```

Python 3.9.0 Shell

```
temp.py
frozenset({1, 2, 3, 4})
Traceback (most recent
call last):
  File "/Users/alexande
rwolodkin/Documents/tem
p.py", line 12, in <mod
ule>
    a[1] = 0
TypeError: 'frozenset'
object does not support
item assignment
```

```
my_text = "Text123456789"
my_text[2]="E"
print(my_text)
```

Python 3.9.0 Shell

```
Traceback (most recent
call last):
  File "/Users/alexande
rwolodkin/Documents/tem
p.py", line 2, in <modu
le>
    my_text[2]="E"
TypeError: 'str' object
does not support item a
ssignment
>>>
```

Ln: 837 Col: 4

Lists

•A list is a **composite** (not elementary) data structure, which in Python is one of the sequence types (like string), i.e. each element has exactly one (or no) successor and is thus iterable

•The content can be addressed via **indices** (as with strings). **Slicing** also works!

•Lists are indicated by **square brackets**, list items are separated by commas.

Ex: [1, 15, 11, 23] or [3.141, ‘Hello', 1, 2]

•You can change list elements "in place" (this is not possible with many other data types, including strings). Therefore lists are called: **mutable**

•**ATTENTION, data types!**

In Python **different data types** may be in one list. In other programming languages this is not always so.

Ex: my_list = [1, 2, 3, “Hello World!”, -2, 4.6]

Lists

•A list is a **composite** (not elementary) data structure, which in Python is one of the sequence types (like string), i.e. each element has exactly one (or no) successor and is thus iterable

•The content can be addressed via **indices** (as with strings). **Slicing** also works!

•Lists are indicated by **square brackets**, list items are separated by commas.

Ex: [1, 15, 11, 23] or [3.141, ‘Hello', 1, 2]

•You can change list elements "in place" (this is not possible with many other data types, including strings). Therefore lists are called: **mutable**

•**ATTENTION, data types!**

In Python **different data types** may be in one list. In other programming languages this is not always so.

Ex: my_list = [1, 2, 3, “Hello World!”, -2, 4.6]

Lists

• List elements can have any non-uniform data type, especially also lists (lists of lists)

```
>>> a = [[1, 4, 9], 15, [3, [11,5]], 23]
>>> a[0] = [1, 4, 9]
>>> a[0][2] = 9
>>> a[2][1][1] = 5
```

• **Attention:** You can NOT access elements with comma-separated indices as usual in other programming languages or mathematics, so: a [0,2] leads to an error

List operations

`L.append(x)`	Das Element x hinten an L anfügen
`L.clear()`	Leert die Liste L
`L.count(x)`	Zählt das Auftreten von x in der Liste L
`L.index(x)`	Gibt den Index des ersten Auftretens von x in L zurück
`L.extend(x)`	Erweitert L um die Liste x
`L.insert(i, x)`	Fügt x am Index i in Liste L ein (und verschiebt den Rest von L nach hinten)
`L.remove(x)`	Entfernt alle x aus Liste L
`L.reverse()`	Dreht die Liste L um
`L.sort()`	Sortiert die Elemente der Liste

Sets

Creating a set:

```
my_set = set([1, 2, 3]) my_set_1 = set("test set")
```

- Elements that occur more than once are deleted

- {1,2,3,3} = {1,2,3} = {3,1,2}

```
>>> my_set = set([1,2,3,4,5,5,6,1,1,1,1])
>>> print(my_set)
{1, 2, 3, 4, 5, 6}
>>> my_set2 = set([20,15,10,10,20,17,3])
>>> print(my_set)
{1, 2, 3, 4, 5, 6}
>>> print(my_set2)
{3, 10, 15, 17, 20}
>>> my_set3 = set("Hello World!")
>>> print(my_set3)
{'H', 'l', 'd', 'o', 'e', 'r', ' ', 'W', '!'}
>>>
```

Tuples

- Tuples are written as comma-separated values in round brackets: (x,y,z).
- They are not changeable "inplace” (they are immutable).
- Reading entries works the same way as with lists.

Tuples

```
>>> tup = (1,2,3)
>>> tup(1)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    tup(1)
TypeError: 'tuple' object is not callable
>>> tup[1]
2
>>> tup[1] = 5
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    tup[1] = 5
TypeError: 'tuple' object does not support item assignment
>>> for i in tup:
        print(i)


1
2
3
```