Bioinformatical problem solving with Python

Wednesdays 17:30-19:00, M801

alexander.nater@uni-konstanz.de

Not covered topics so far

- Error handling (exceptions)
- **Numerical and scientific libraries (SciPy, NumPy, pandas, etc.)**
- Plotting libraries (Matplotlib)
- Software design

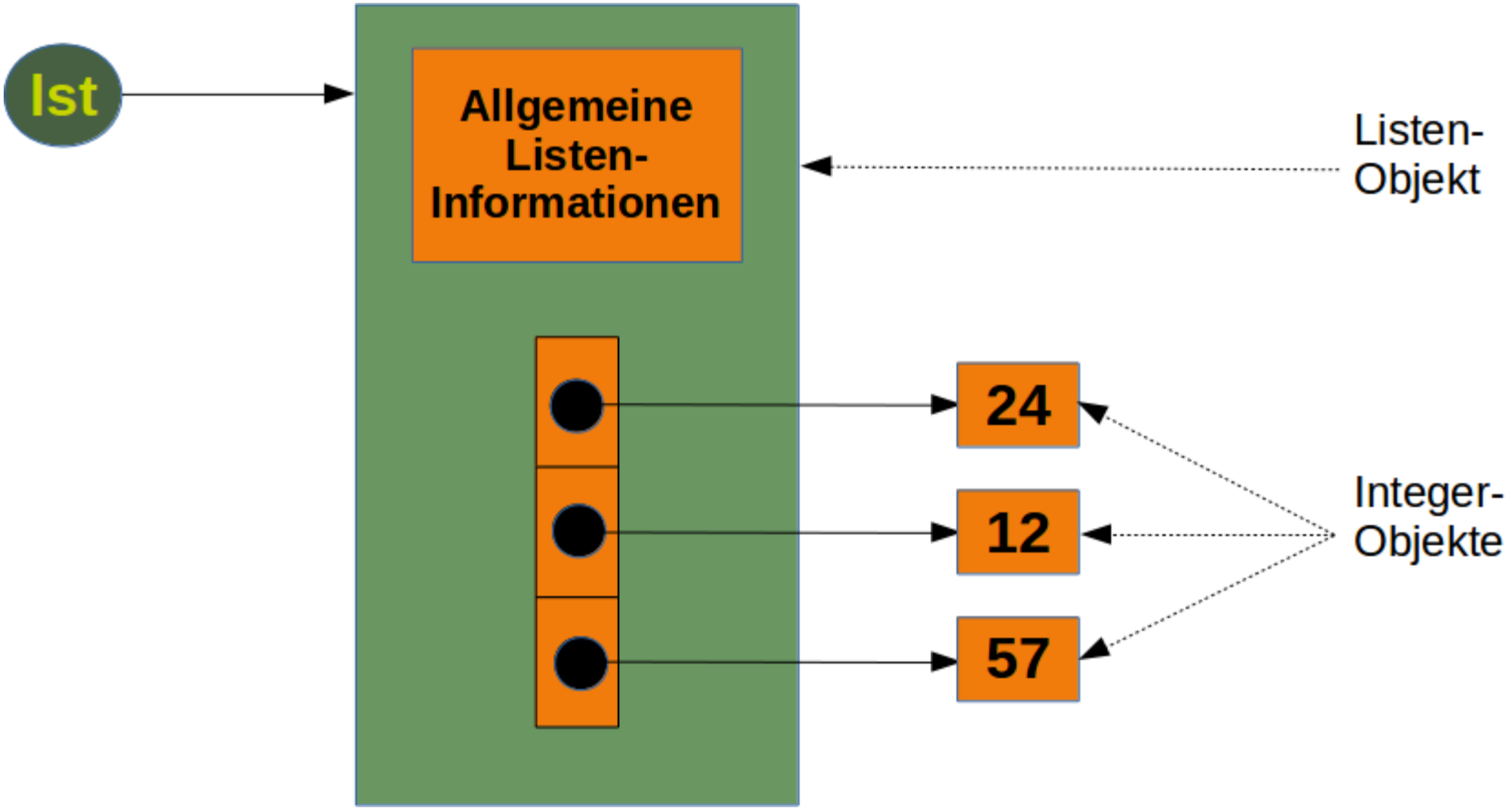
Multi-dimensional containers

- Built-in container classes in Python support any kind of object types (e.g. integers, floats, lists, dicts).

- List of lists, dictionaries of tuples, etc. are already implemented:
  lol = [[1,2,3], [4,5,6]]
  print(lol[1][2]) → 6

  lol[1].append(7)
  print(lol) → [[1,2,3], [4,5,6,7]]

Memory representation



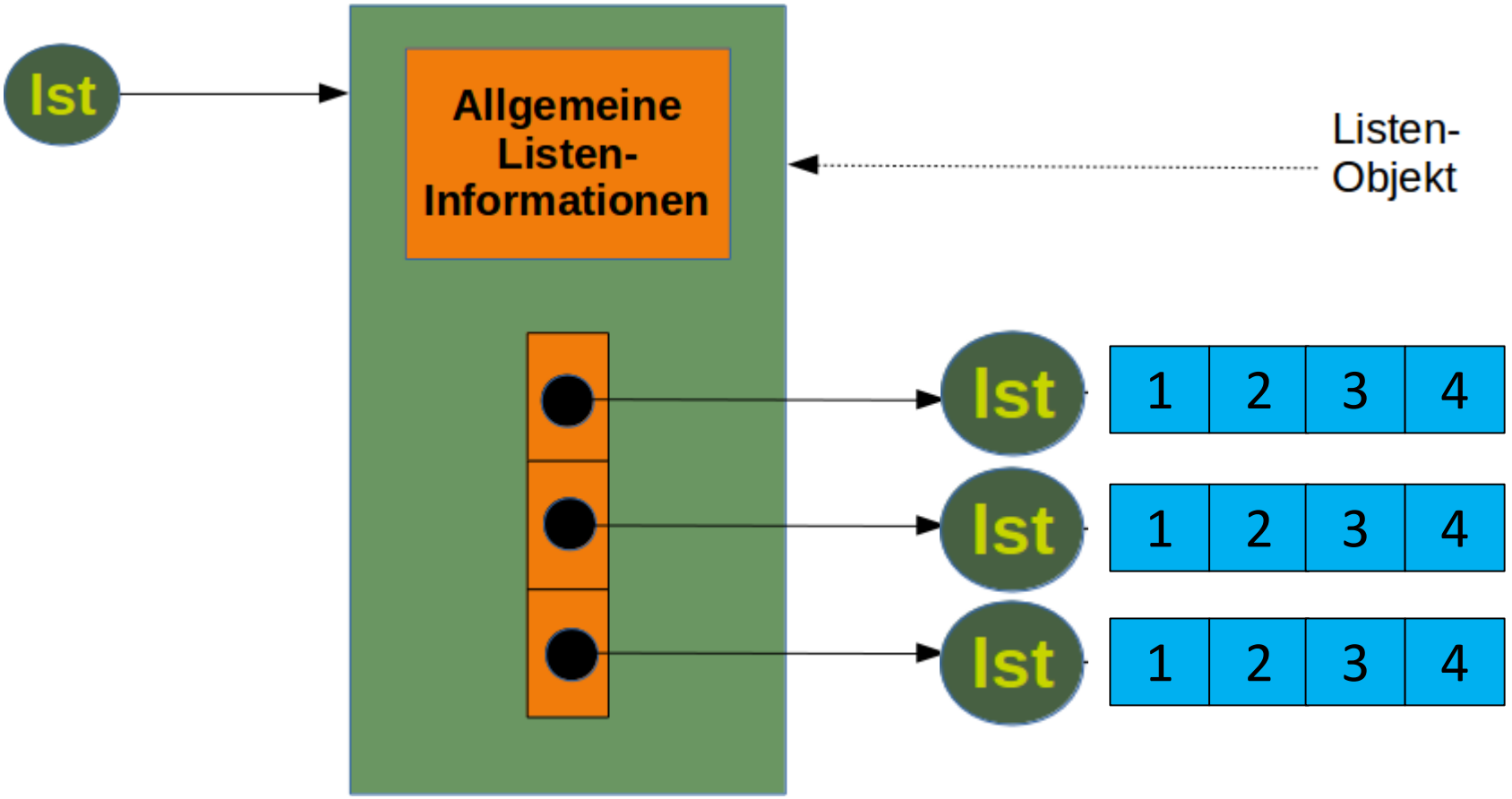
Memory representation

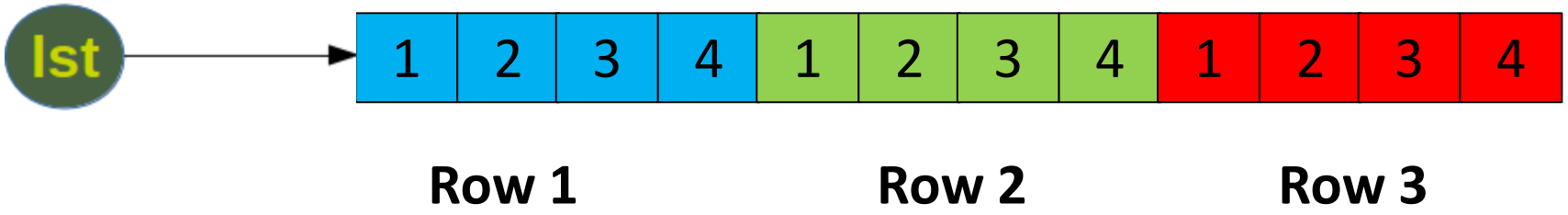
- Built-in containers are not very memory-efficient:
  from sys import getsizeof as size
  la = [24, 12, 57]
  print(size(la)) → 88
  print(size(la[0]) * len(la)) → 84

- Built-in containers have substantial memory overheads:
  lb = []
  print(size(lb)) → 64

- A list of three integer values consumes 64 + 3*8 + 3*28 bytes of memory!

List of lists




lst
1
2
3
4
1
2
3
4
1
2
3
4
Row 1
Row 2
Row 3

Write a class to represent a fixed-size two-dimensional array. The constructor should take three arguments: number of rows, number of columns, and initial value for each cell (default 0). Implement a getter method taking a row and column index as arguments and returning the corresponding value, as well as a setter function taking the value to be set as third argument. Both methods should throw an exception if the user tries to access or set a cell value with invalid indices.

Create a list with integer values from -20 to 50, representing values in degrees Celsius. Create a second array with the corresponding values in degrees Fahrenheit (C * 9/5 + 32).

- The NumPy library offers a memory-efficient multi-dimensional container for values of the same predefined type.

- Every array has a ndim (number of dimensions), shape (tuple indicating the size of each dimension), and dtype (data type) attribute:
  import numpy as np
  my_array = np.array([1,2,3,4,5])
  print(type(my_array)) → class 'numpy.ndarray'
  print(my_array.dim) → 1
  print(my_array.shape) → (5,)
  print(my_array.dtype) → dtype('int64')

- NumPy arrays support arithmetic operations directly on the array object (vectorization):

```
import numpy as np
cvalues = list(range(-20, 101))
carray = np.array(cvalues)
print(carray * 9/5 + 32)
```

Compare the size of a NumPy array with the size of a Python list containing the same elements.

- The array function converts any sequence object into an NumPy array:
  la = list(range(0, 1000))
  my_array = np.array(la)

- The arange and linspace functions directly create arrays with evenly spaced numbers:
  my_array = np.arange(0, 1000)

- Array indexing and slicing works exactly as with the usual Python containers:
  print(my_array[:5]) → [0, 1, 2, 3, 4]

- The array function converts a list of equal-length lists into a two-dimensional array:
  la = [[1,2,3,4,5], [6,7,8,9,10]]
  my_array = np.array(la)
  print(my_array.dim) → 2
  print(my_array.shape) → (2, 5)
  print(my_array.dtype) → dtype('int64')

- The functions ones, zeros, and empty generate arrays of a given shape with each cell filled with 1 / 0, or leaving the cells uninitalized:

```
my_ones = np.ones(10)
my_zeros = np.zeros((8, 4))
my_empty = np.empty((4, 10, 8))
```

- Elements of multi-dimensional arrays can be accessed by providing a comma-separated list of indices:
  my_array = np.array([[1,2,3,4], [5,6,7,8]])
  print(my_array[1, 2]) → 7
  print(my_array[1, 1:3]) → [6, 7]

- Slices are array views, not copies! Changes to slices affect the original array:
  my_slice = my_array[1, 1:3]
  my_slice[1] = 10
  print(my_array) → [[1,2,3,4], [5,6,10,8]]
  my_slice = my_array[1, 1:3].copy()

Boolean indexing

- Like in R, arrays can be indexed and sliced by providing an equal-length array of Boolean (True, False) values:
  my_array = np.array([1, 2, 3, 4])
  my_bool = np.array([False, True, False, True])
  my_subset = my_array[my_bool]
  print(my_subset) → [2, 4]

- Boolean arrays can be generated by using the vectorized comparison operators:
  my_array = np.arange(0, 100)
  my_bool = (my_array > 60) & (my_array < 80)
  my_subset = my_array[my_bool]

Exercise

Create an array with integer values from 0 to 10000. Extract all numbers that are divisible by 5 or 7, but not by 3.