PROGRAMMING IN PYTHON I

Unit 01: Tuples, lists, indices, dictionaries, slices

Michael Widrich & Sebastian Lehner
Institute for Machine Learning

JOHANNES KEPLER
UNIVERSITY LINZ

Outline

1. **Grouping values**

2. **Excursion: Linear array**

3. **Sequence and mapping types**

4. **Tuples**

5. **Dictionaries**

6. **Slicing, indexing details, and more examples**

GROUPING VALUES

Motivation

- Often we want to handle a group of values
  - E.g.: Group of names, measurements, etc.
- We could handle each value as separate variable but this would get tedious and complicated:

```
var1 = 1
var2 = 2
var3 = 3
var4 = 4
...
```

- Instead, we want to have one variable as reference to the group of values:

```
var = [1, 2, 3, 4, ...]
```

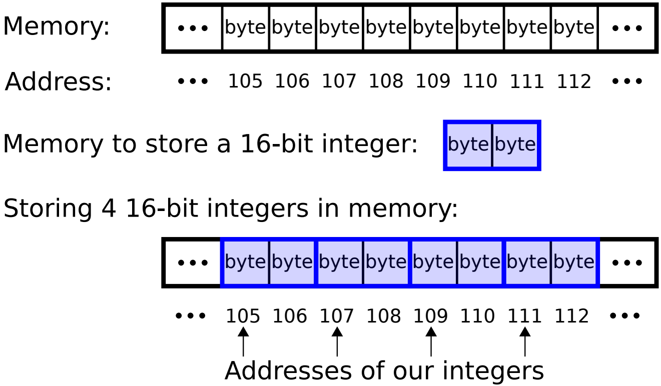
  - We can then retrieve individual values via `var`

EXCURSION: LINEAR ARRAY

Excursion: Linear array (1)

- Goal: We want to store a group of values of a fixed datatype
  - □ E.g. 4 16-bit integer values
- Assumption: Our storage (=memory) consists of 1-byte large blocks with contiguous addresses
  - □ Byte 1 has address 0, byte 2 has address 1, etc.
- We know we need $m = 4 \cdot 16/8 = 8$ bytes to store our values
  - □ We can allocate 8 bytes of memory with contiguous addresses, starting at address x

Excursion: Linear array (2)



Memory: ••• byte byte byte byte byte byte byte byte •••

Address: ••• 105 106 107 108 109 110 111 112 •••

Memory to store a 16-bit integer: byte byte

Storing 4 16-bit integers in memory:

••• byte byte byte byte byte byte byte byte •••

••• 105 106 107 108 109 110 111 112 •••

Addresses of our integers

Excursion: Linear array (3)

- Given the address of the first byte and the datatype bytes (m), we can access one stored value via its index i:
  - □ 1st integer will be at address $x + 0 * m$
  - □ 2nd integer will be at address $x + 1 * m$
  - □ 3rd integer will be at address $x + 2 * m$
  - □ (i+1)th integer will be at address $x + i * m$
  - □ Note: Indices here are integers, starting at 0
- In our example: $x = 105$, $m = 2$

→ 3rd integer will be at address $x + i * m = 105 + 2 * 2 = 109$

- This is the concept of a linear array

Excursion: Linear array (4)

- Properties of linear arrays:
  - □ The cost of the indexing operation is independent of the size of the array or the value of the index (in contrast to linked lists[1])
  - □ Allocation of memory takes time and is therefore costly
  - □ If we want to increase the size of our linear array, we might have to copy the whole array to allocate enough contiguous space
  - → Increasing the number of elements is costly if done naively

[1] Linked lists: `https://realpython.com/linked-lists-python/`

SEQUENCE AND MAPPING TYPES

Sequence and mapping types

- We consider two types of groups, depending on relationships of values:
- Sequence types
  - □ Ordered list (=sequence) of values
  - □ Position of values (=index) in sequence is used to access a value
  - □ Python: List, tuple, string, . . .
- Mapping types or associative arrays[2]
  - □ Group of (key, value) pairs
  - □ Keys, e.g. strings, are used to access values
  - □ Values in group might be unordered
  - □ Python: Dictionary, . . .

[2] https://en.wikipedia.org/wiki/Associative_array

Lists (1)

- In Python, a list is the most versatile sequence type
- It is created using square brackets containing comma-separated values (=items or elements):

```
my_list = ['some item', 'b', 5463, 5.24]
```

- It can contain items of variable datatypes
- The order of items is preserved
- The index of the items is used to access it:

```
my_list[1] # Returns value 'b'
```

  - □ Important: Indices in Python are integers and start at 0!

Lists (2)

- Python lists are mutable
  - → We can add, modify, and delete the items in the list
- Python lists can contain all kinds of objects
- Python lists can be nested, i.e. contain other lists:

```
my_list = [23, '367', ['trh', 5], 6.35]
my_list[2] # Returns ['trh', 5]
```

- See `01_code.py` for more operations

More information: `https://docs.python.org/3/tutorial/introduction.html#lists`

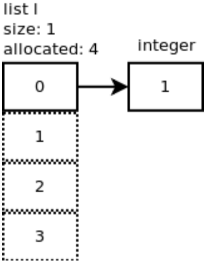
Lists: Realization (1)

- CPython lists are variable-length arrays
  - □ Similar to linear array but holds references to (addresses of) objects instead of objects themselves
  - → Can store values with different datatypes but additional overhead for decoding values
  - □ Keeps track of start of array and length of array (=number of items)
  - □ Cost of indexing still independent of index value/array size
  - □ Pre-allocates memory slots in pairs of 4 even if currently not needed (for speed-up)

More information: `https://docs.python.org/3/faq/design.html#how-are-lists-implemented-in-cpython`

Lists: Realization (2)

- We initialize a list called `l` with one element via:
  `l = [1]`
- `l` now holds the location of the integer object with value 1
- The number of memory slots allocated by the list now is not one but four!



Lists: Realization (3)

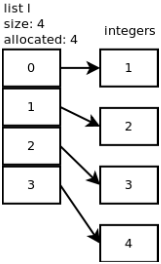
- Why? Chances are that you are going to add more than one element, but memory allocation is an expensive operation.
- More efficient to pre-allocate memory slots in steps of 0, 4, 8, 16, 25, 35, 46, 58, 72, 88, . . .
- An empty array has 0 slots,
  If it has 1-4 elements it has 4 slots,
  If it has 5-8 elements it has 8 slots . . .
- This behaviour is characteristic for Dynamic Arrays[3]

[3] https://en.wikipedia.org/wiki/Dynamic_array

Lists: Realization (4)

- Let's see what this behaviour means in practice
- We add three elements at the end of the list via:

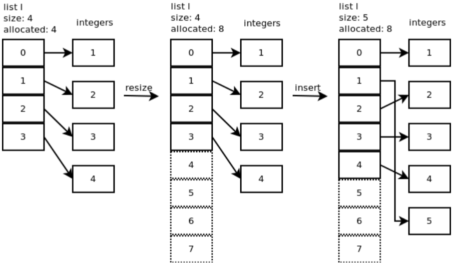
```
l.append(2)
l.append(3)
l.append(4)
```



Lists: Realization (5)

- Now the memory allocated to our list is exhausted
- Python lists allow not only to add elements at the end (append) but to insert them at any position:

  `l.insert(1,5)`



- The allocated memory had to be expanded by one step from 4 to 8 slots
- The second pointer now points to a different object (5)

TUPLES

Tuples

- Another example of a sequential datatype in python
- Tuples are created via a number of values, separated by commas

  ```
  my_tuple = 42, 'a string', 346.345
  ```

  or

  ```
  my_tuple = (42, 'a string', 346.345)
  ```
- Tuples are similar to lists but immutable
  - □ Once a tuple is created it can not be changed anymore!

    ```
    my_tuple[1] = 5 # This would fail
    ```
  - □ It is possible to create tuples with mutable objects, e.g. lists

More information: `https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences`

DICTIONARIES

Dictionaries: Motivation

- Imagine you want to implement a phone book, i.e. associate a name with a phone number
- You could store the phone number in a list
- You have to remember whose number is at which position
- Could use a second list of names that with same order
  - → tedious to use and maintain!
- It would be better to use the names as indices to the phone number (=(key, value) pairs)
  - → this is a map/associative array
- Python dictionaries are mappings/associative arrays

Dictionaries in Python (1)

- Python dictionaries are mappings/associative arrays
  - □ Consist of (key, value) pairs, e.g. (name, phone number)
  - □ Any immutable/hashable object can be used as key
  - □ Any object can be used as values
- Mutable and not ordered
- Created with syntax

```
my_dict = dict(key1=value1, key2=value2)
```

or

```
my_dict = {key1:value1, key2:value2}
```

More information: https://docs.python.org/3/tutorial/datastructures.html#dictionaries,

https://docs.python.org/3/faq/design.html#how-are-dictionaries-implemented-in-cpython

Dictionaries in Python (2)

- Phone book example:

  ```
  phone_book = {'sam': '01234', 'alex': '98765'}
  ```

  or alternatively:

  ```
  phone_book = dict(sam='01234', alex='98765')
  ```

- Now `phone_book` contains two entries, let's use it to get sam's number:

  ```
  phone_book['sam'] # returns '01234'
  ```

- The key of dictionary entries has to be unique
- The following will overwrite the previous number for sam:

  ```
  phone_book['sam'] = '13579'
  ```

SLICING, INDEXING DETAILS, AND MORE EXAMPLES

Slicing and indexing details

- Python allows to select a range of items in a sequence type, e.g. a list, via slicing

  ```
  my_list[2:5] # View on list at indices 2, 3, 4
  ```

- It also allows for indexing in the reverse order

  ```
  my_list[-2] # Returns second-last element in list
  ```

- More information and examples in `01_code.py`