PROGRAMMING IN PYTHON I

Data Structures

Andreas Schörgenhumer

Institute for Machine Learning

JOHANNES KEPLER
UNIVERSITY LINZ

Institute for
Machine Learning

Copyright Statement

Contact

Andreas Schörgenhumer

Institute for Machine Learning
Johannes Kepler University
Altenberger Str. 69
A-4040 Linz

E-Mail: `schoergenhumer@ml.jku.at`

Write mails only for personal questions

Institute ML Homepage

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`
- A group of values is a **data structure**

Data Structures

- **Sequence types**
  - **Ordered list** (=sequence) of values
  - Position of values (=index) in sequence is used to access a value
  - Python: List, tuple, string, range
- **Unordered collections**
  - **Unordered set** of unique elements
  - Python: Set
- **Mapping types**
  - Group of **key-value pairs**
  - Unique keys are used to access values
  - Key-value pairs **might be unordered**
  - Python: Dictionary
- Python takes care of growing data structures for you

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 data types**
- The **order of items is preserved**
- The index of the items is used to access them:

  ```
  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 (also mixed)
- Python lists **can be nested**, i.e., contain other lists:

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

Tuples

- Another example of a **sequence data type** in Python
- Tuples are created via a number of values, separated by commas

```
my_tuple = 42, "a string", 346.345
my_tuple = (42, "a string", 346.345)
```

- Tuples are **similar to lists but immutable**
  - □ Once a tuple is created, it cannot be changed anymore!

    ```
    my_tuple[1] = 5  # This would fail
    ```

  - □ It is possible to create tuples with mutable objects, e.g., lists

Sets

- Sets are **unordered** collections of **unique** elements
- In Python, a set is created using `set()` (required for an empty set) or with curly braces that include at least one element:

```
my_set = set()  # Creates an empty set
my_set = {"hi", 12, 123}
```

- Common set operations are supported:
  - □ Union: `set1 | set2`
  - □ Intersection: `set1 & set2`
  - □ Difference: `set1 - set2`
  - □ ...

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 with same order

  → Tedious to use and maintain!
- It would be better to use the names as indices to the phone number, i.e., to have key-value pairs

  → This is a map/mapping

Dictionaries (1)

- Python dictionaries are **mappings**
  - □ Consist of **key-value pairs**, e.g., name → phone number
  - □ Any **hashable** object can be used as **key**
  - □ Any object can be used as **value**
- **Mutable** and **ordered**[1] (insertion order is preserved)
- Created with syntax (keys can be anything)

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

  or syntax (keys must be identifiers and are automatically converted to type string)

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

[1] Since Python version 3.7

Dictionaries (2)

- Phone book example (mapping string keys to string values):

  ```
  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 keys of dictionary entries have to be **unique**
- The following will overwrite the previous number for sam:

  ```
  phone_book["sam"] = "13579"
  ```

List Comprehensions

- Compact way to loop over an iterable (strings, lists, sets, ranges, etc.), perform (optional) actions on the elements and store the results in a list:

```
lst = [code-to-be-executed for element in iterable]
```

- Can optionally include conditions during iteration for filtering elements:

```
lst = [code-to-be-executed for element in iterable
       if condition]
```

- Can also store results in sets and dictionaries, in which case they are called set and dictionary comprehensions

Code Example

```python
my_list = ["a", "b", "c"]
uppercase_list = [item.upper() for item in my_list]
```

- Very compact way instead of a normal for loop (and typically also faster in terms of run-time performance)

Slicing and Indexing Details

- Python allows to select a range of items in a sequence type, e.g., a list or a string, via **slicing**
- Syntax for slicing: `sequence[start:end:step]` with default values if not specified explicitly (0 for `start`, length of sequence for `end`, 1 for `step`); `start` is inclusive, `end` is exclusive

```
my_list[2:5]  # View on list at indices 2, 3, 4
my_list[2:5:1]  # Same with explicit step size
my_list[2:5:2]  # View on list at indices 2, 4
my_list[::-1]  # Entire list in reverse order
```

- Negative numbers can also be used in indexing and slicing (-1 for the last element, -2 for the second-last, etc.)

```
my_list[-2]  # Second-last element in list
```

Unpacking

- If you have a sequence of values, you cannot only assign this sequence to some variable but also its contents. This is called **unpacking**

```
abc = [1, 2, 3]  # Regular assignment
a, b, c = [1, 2, 3]  # a -> 1, b -> 2, c -> 3
```

- Can improve code readability (see the accompanying code file for examples)

Objects and (Im)Mutability

- Recall: Everything in Python is an **object**
- Some are **immutable** (integers, tuples, etc.) and some are **mutable** (e.g., lists, dicts, etc.)
- Immutability means that objects cannot be changed, e.g.:

```
x = 3        # Integer object with constant 3
y = ("a", 3) # 2-tuple object with fixed references
z = "hi"     # String object with fixed "hi"
```

- Mutability means that objects can be changed, e.g.:

```
x = [1, 2, 3]  # List object with items 1, 2, 3
x[0] = 7       # x -> [7, 2, 3]
```

Assignments + Immutability

- An **assignment to an existing object** does not copy the object, it will simply **reference the same object**, e.g.:

```
x = 3  # Integer object
y = x  # The same integer object
```

- `x` and `y` reference the same object, the object exists only once in memory
- Since integers are immutable, there cannot be any side effects in this case

Assignments + Mutability

- Now, consider another example with mutable objects:

  ```
  x = [1, 2, 3]  # List object with items 1, 2, 3
  y = x          # The same list object
  ```

- x and y reference the same object, the object exists only once in memory
- Since lists are mutable, there can be side effects in this case if you change the object, e.g., via `x[0] = 7`
- With this, the list changed; its content is now `[7, 2, 3]`
- Since x and y still reference the same object, this change is visible through both variables (alias effect):

  ```
  x  # List object with items 7, 2, 3
  y  # The same list object, which means the same
      items 7, 2, 3!
  ```

Consideration of Side Effects

- Side effects are not bad per se, you just have to consider them while programming
- Often, you actually explicitly want this behavior, e.g., a sorting function that sorts a list **in-place**, i.e., makes changes directly within this list object
- If you do not want this behavior, you can make a copy of your object and perform the changes on the copy, e.g., a sorting function that sorts a list by first copying it, sorting the copy and then returning this copy, leaving the original list unchanged