3 – Sets & dictionaries

Bálint Aradi

Course: Scientific Programming / Wissenchaftliches Programmieren (Python)

Bremen Center for Computational Materials Science BCCMS

Universität Bremen

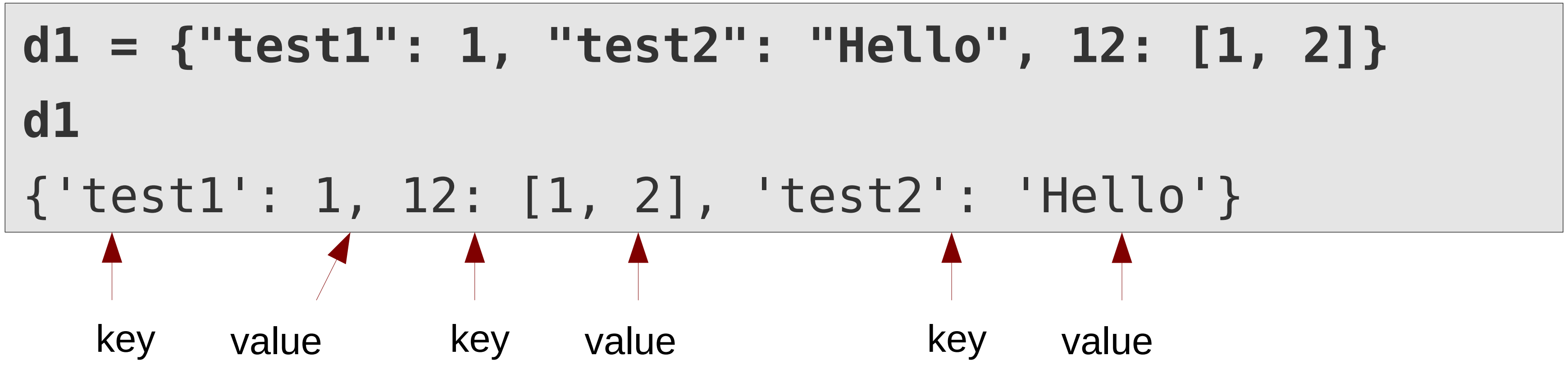
https://www.bccms.uni-bremen.de/people/b-aradi/wissen-progr/python/2023

Outline

- Dictionaries
- Sets
- Some string methods

Dictionaries

- Store items of arbitrary type
- Items identified by their unique key, not by their position
- Key must be of immutable data type
- Dictionary is delimited by **{** and **}**



```
d1 = {"test1": 1, "test2": "Hello", 12: [1, 2]}
d1
{'test1': 1, 12: [1, 2], 'test2': 'Hello'}
```

- Elements can be accessed as in lists, but by using their key

```
d1["test1"]
1
d1[12]
[1, 2]
```

Dictionaries

- Dictionaries are mutable
- If a key is used, which is already present, the item is overwritten

```
d1["test1"] = 3+4j
d1
{'test1': (3+4j), 12: [1, 2], 'test2': 'Hello'}
```

- If a key is used, which is not present yet, a new item is created

```
d1[(-1,)] = 12
d1
{'test1': (3+4j), 12: [1, 2], 'test2': 'Hello', (-1,): 12}
```

- Elements can be deleted by the **del** statement

```
del d1["test2"]
d1
{'test1': (3+4j), 12: [1, 2], (-1,): 12}
```

Dictionaries

- The **in** operator can be used to check the presence of a key

```
'test1' in d1
True
"missing" in d1
False
```

- Trying to access a non-existing key leads to an error

```
d0["missing"]
… KeyError: 'missing
```

- The **get()** method can be used to obtain an item or a default value if the key is not found

```
default = -1
key = "missing"
value = d0.get(key, default)
```

```
if key in d0:
    value = d0[key]
else:
    value = default
```

Dictionaries as iterators

- Dictionaries return their keys one by one:

```
dd = {12: [1, 2], 'test1': 3.2, (-1,): True}
for key in dd:
    print(f"key: {key}")
```

```
key: 12
key: (-1,)
key: test1
```

- An iterator over dictionary values can be obtained by the **values()** method

```
for val in dd.values():
    print(f"value: {val}")
```

```
value: [1, 2]
value: True
value: 3.2
```

- An iterator over key, value tuples can be obtained by the **items()** method:

```
for key, val in dd.items():
    print(f"{key}: {val}")
```

```
12: [1, 2]
(-1,): True
test1: 3.2
```

Creating dictionaries

- From a dict-literal

```
dd = {3.2: 'hello', 'a': 1}
```

```
{3.2: 'hello', 'a': 1}
```

- From an iterable containing (key, value) tuples

```
dict([('a', 1), (3.2, 'hello')])
```

```
{3.2: 'hello', 'a': 1}
```

- From a dictionary comprehension

filtering is optional

```
{keyexpr: valuexpr for itervar in iterator if condition}
```

```
nums = [1, 3, 2, 9, 8, 3]
oddsquares = {num: num**2 for num in nums if num % 2 == 1}
```

```
{1: 1, 3: 9, 9: 81}
```

Sets

- Sets contain only keys (like dictionaries), but no values
- Sets are mutable
- All members must be of inmutable type
- Every key (element) is unique and occurs only once

```
s1 = {"test", 12, -3.6, (1,2)}
s1
{(1, 2), 12, -3.6, 'test'}
```

- Elements can be added by the **add()** method

```
s1.add(True)
s1
{(1, 2), True, 12, -3.6, 'test'}
```

- Adding an already existing element to the set leaves it unchanged:

```
s1.add("test")
s1
{(1, 2), True, 12, -3.6, 'test'}
```

Sets

- Elements can removed by the **remove()** method

```
s1.remove(-3.6)
s1
{(1, 2), True, 12, 'test'}
```

- The **in** operator can be used to check the presence of an element

```
s1
{(1, 2), True, 12, 'test'}
12 in s1
True
13 in s1
False
```

Sets as iterators

- Sets return their elements one by one, but the order is undetermined:

```
s1 = {True, 12, 'test', (1, 2)}
for item in s1:
    print('Item:', item)
```

```
Item: (1, 2)
Item: True
Item: 12
Item: test
```

Creating sets

- From a set-literal

```
st = {1, 9, (3, 4), False, 8.2}
```

```
{1, 9, (3, 4), False, 8.2}
```

- From an iterable containing (key, value) tuples

```
set([1, 9, (3, 4), False, 8.2])
```

```
{1, 9, (3, 4), False, 8.2}
```

- From a set-comprehension

filtering is optional

```
{expr for itervar in iterator if condition}
```

```
nums = [1, 3, 2, 9, 8, 3]
oddsquares = {num**2 for num in nums if num % 2 == 1}
```

```
{1: 1, 3: 9, 9: 81}
```

Containers – overview

Lists

- Ordered
- Elements indexed by sequential integer (position)
- Index of a given element might change when other elements are inserted/deleted
- Fast *O(1)* access by index
- Slow O(N) access by value

Dictionaries

- Unordered (ordered for Python > 3.7)
- Elements indexed by key (arbitrary inmutable object)
- Index of given element remains unchanged when other elements are inserted/deleted
- Fast O(1) by key
- Slow O(N) access by value

Sets

- Unordered
- Elements are unique
- Fast O(1) access for checking element presence

Containers – access times

Note: choice of the container type might seriously affect performance

```
import random
MAX_NUM = 10000000
```

```
random_list = [random.randint(0, MAX_NUM – 1)
               for _ in range(MAX_NUM)]
random_set = set(random_list)
```

```
%%timeit
MAX_NUM in random_list
```

compare execution times!

```
%%timeit
MAX_NUM in random_set
```

Comparing containers

- Equality of containers can be checked with **==** and **!=** operators
- Two containers are equal, if all elements and their keys/indices are equal

```
{'key1': 1, 'key2': 2} == {'key2': 2, 'key1': 1}
{'key1': 9, 'key2': 2} == {'key2': 2, 'key1': 1}
```

```
True
False
```

- Ordered (sequence) types (lists, tuples, but not dicts) can also be compared by **>**, **>=**, **<**, **<=**
- The comparison is done component-wise
- The first non-matching component determines the relation

```
(1, 2, 3) > (1, 2, 4)
(9, "ahoi") > (6, "hello")
```

```
False
True
```

- The same ordering rules are applied in internal routines, like sorting:

```
ll = [(9, "ahoi"), (6, "hello")]
ll.sort()
ll
```

```
[(6, 'hello'), (9, 'ahoi')]
```

Some string methods

split(*separator*)

- Splits a string into pieces using a given delimiter

```
"a,b,c,d".split(",")
['a', 'b', 'c', 'd']
```

- If no delimiter is specified, the string is split by any whitespace characters (space, tab, newline)

```
"One short line.\nOne more.".split()
['One', 'short', 'line.', 'One',
'more.']
```

join(*iterator*)

- Joins the elements of the iterator into a string using the string as delimiter
- All elements returned by the iterator must be strings

```
", ".join(["word1", "word2", "word3"])
'word1, word2, word3'
```

Some string methods

lower(), upper()

- Converts all characters in a string to lower/upper case

```
"Word".lower()
'word'
words = ["Apfel", "Birne"]
[word.lower() for word in words]
['apfel', 'birne']
```

lstrip(), rstrip(), strip()

- Removes whitespace characters from left, right and both sides of a string

```
" word ".lstrip()"
'word '
" word ".rstrip()
' word'
" word ".strip()
'word'
```

Some string methods

replace()

- Replaces all occurances of a substring with a given replacement

```
txt = "However, the sky was dark."
txt.replace("was", "is")
'However, the sky is dark.'
txt.replace(",", "")
'However the sky was dark.'
```

- The result of all string methods is always a new string (strings are immutable)
- If the result should be manipulated further by a string method, the methods can be “chained”

```
txt2 = txt.replace("was", "is")
txt_new = txt2.replace(",", "")
```

```
txt_new = txt.replace("was", "is").replace(",", "")
```

For further sting methods, see the Python Library Docs (String methods)

For non-trivial replacements regular expressions might be more suitable

Have fun!