3 – Container data types

Bálint Aradi

Course: Scientific Programming / Wissenchaftliches Programmieren (Python)

https://www.bccms.uni-bremen.de/cms/people/b-aradi/wissen-progr/python/2022

Outline

- Comments in source code
- Tuples, lists, dictionaries
- In-place arithmetic operators
- Some string methods

Comments in source code

- Comments are indicated by a non-quoted hashmark (#)
- Anything between the comment mark and the end of the line is ignored by the interpreter
- Comments can be used to add **short explanation for non-trivial** / unexpected **operations** so that the code logics can be followed easily

```
# Shift index by one to ensure counting from one
ind += 1
```

- Comments should **not be used to explain trivialities**

```
# Run a loop over the range of all terms
for ii in range(nterm):
…
```

- **Your code should be clean and self documenting, and not requiring any comments** (or maximal a few ones) and still being easy to follow.

Tuples

- Contain sequences of objects of arbitrary data type
- Items within a tuple can have different data type
- Delimited by **(** and **)**, elements are separated by **,**

```
t1 = (1, 3.0, "Hello")
t1
(1, 3.0, 'Hello')
```

- If non-ambiguous, the delimiters can be omitted

```
t1 = 1, 3.0, "Hello"
t1
(1, 3.0, 'Hello')
```

- **Empty tuple** is specified with **()**:

```
t0 = ()
t0
()
```

Tuples

- For tuples with one element, a comma must be appended after last element to make it non-ambiguous:

```
t1bad = (1)
t1bad
1

t1good = (1,)
t1good
(1,)
```

- For tuples with more than one elements last comma may be added:

```
t1multi = (1, 2,)
t1multi
(1, 2)
```

Accessing elements of a tuple

- Tuple elements, tuple ranges can be accessed by the **[ ]** operator
- Works exactly as for substring/character selection in strings

Negative indices count elements backwards:
-1 = last element

```
t1
(1, 3.0, 'Hello')
t1[0]
1
t1[-1]
'Hello'
t1[1:3]
(3.0, 'Hello')
t1[::-1]
('Hello', 3.0, 1)
```

- Tuples are immutable, and can not be changed once they have been created

```
t1[0] = 24
… TypeError: …
```

Tuple operations

- Tuples can be appended with the **+** operator

```
t1 = (1, 2, 3)
t2 = (4, 5)
t3 = t1 + t2
t3
(1, 2, 3, 4, 5)
```

- Tuples can be repeated with the **+** operator

```
t4 = t2 * 3
t4
(4, 5, 4, 5, 4, 5)
```

- Number of items in a tuple can be queried by the **len()** function:

```
len(t4)
6
```

Tuple assignment

- Components of a tuple can be assigned to individual variables within an assignment

```
mytuple = (1, 2)
t1, t2 = mytuple
t1
1
t2
2
```

Assigning entire tuple to one variable

Assigning tuple components to individual variables

- The number of variables on the left hand side must be compatible with the tuple length:

```
mytuple = (1, 2, 3)
t1, t2 = mytuple
ValueError: too many values to unpack (expected 2)
```

Lists

- Lists are very similar to tuples, but they are **mutable**
- Lists are delimited by **[** and **]**, lists elements are separated by **,**
- Element and range selection, **len()** function, operators **+** and ***** work analogously to tuples

```
l1 = [1, 3.0, 'Hello']
l1
[1, 3.0, 'Hello']
l1[0]
1
l1[-1]
'Hello'
l1[1:3]
[3.0, 'Hello']
l1[::-1]
['Hello', 3.0, 1]
```

```
len(t1)
3
l2 = []
len(l2)
0
l3 = [1, 4,]
l4 = l1 + l3
l4
['Hello', 3.0, 1, 1, 4]
l5 = l3 * 2
l5
[1, 4, 1, 4]
```

Modifying lists

- Changing elements

```
l1 = [3, 2, "test", 1.5]
l1
[3, 2, 'test', 1.5]
l1[0] = 42
l1
[42, 2, 'test', 1.5]
```

- Changing ranges

```
l1[0:2] = [1, -1]
l1
[1, -1, 'test', 1.5]
l1[0:4:2] = [0, 0]
l1
[0, -1, 0, 1.5]
```

Modifying lists

- If the range is continuous, it can be replaced with a list (iterable) of arbitrary size. The size of the list will change accordingly

```
l1
[0, -1, 0, 1.5]
len(l1)
4
```

```
l1[0:3] = [9,]
l1
[9, 1.5]
len(l1)
2
```

- A given element or range can be deleted by the **del** statement

```
l2 = [1, 2, 3, 4]
del l2[0]
l2
[2, 3, 4]
del l2[0:2]
l2
[4]
```

```
l3 = [1, 2, 3, 4, 5, 6]
l3
[1, 2, 3, 4, 5, 6]
del l3[0::2]
l3
[2, 4, 6]
```

List methods

- The **append()** method can be used to append one element to the list

```
l5 = []
l5.append(1)
l5
[1]
```

```
l5.append(2)
l5
[1, 2]
```

- The **extend()** method can be used to extend the list by an other list (iterable)

```
l5.extend([4, 5, 6])
l5
[1, 2, 3, 4, 5, 6]
```

or

```
l5 += [4, 5, 6]
l5
[1, 2, 3, 4, 5, 6]
```

- Further methods for list manipulation
  - **insert()**, **index()**, **reverse()**, …
  - See **Python Standard Library documentation**: Sequence types

List methods

- Lists can be sorted by the **sort()** method:

```
ll =  [9, -1, 3, 8, 5]
ll.sort()
ll
[-1, 3, 5, 8, 9]
```

```
ll =  [9, -1, 3, 8, 5]
ll.sort(reverse=True)
ll
[-1, 3, 5, 8, 9]
```

- The **in** operator can be used to query for the presence of an element in the list
- It checks each list element individually, so do not use it for large structures (*O(N)*)

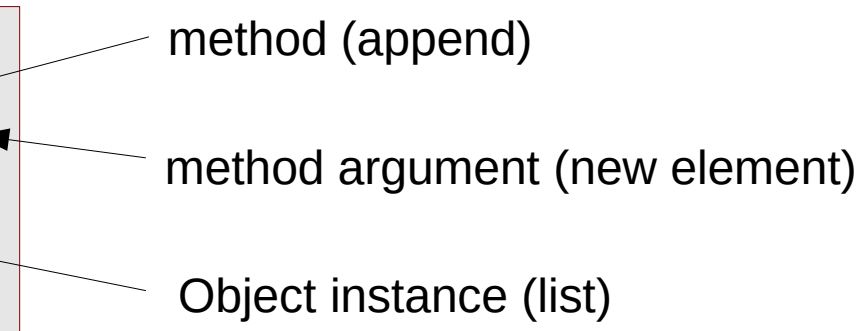
```
l5
[1, 2, 3, 4, 5, 6]
3 in l5
True
-1 in l5
False
```

Objects and methods in a nutshell

- In Python, every type is a class, every instance (variable) an object.
- An object contains:
  - **Data**
  - **Methods**: Functions which use/manipulate the contained data
- Methods are called as

```
objectname.methodname(eventual method arguments)
```

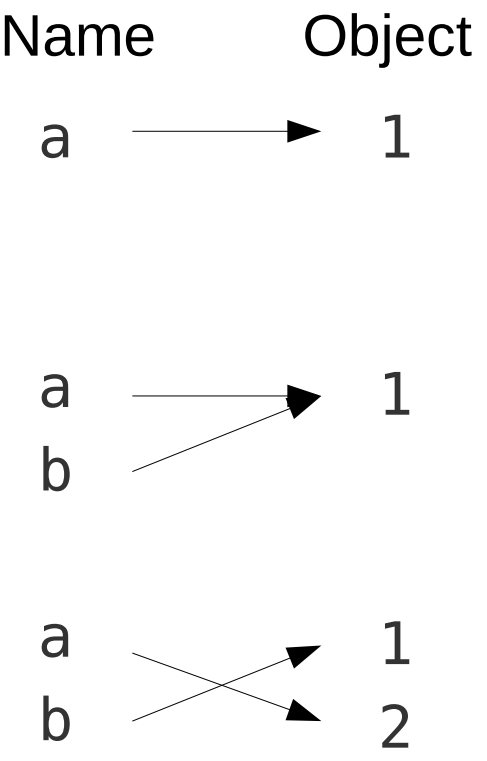
```
ll = [1, 2]
ll.append(3)
ll
[1, 2, 3]
```



Assignment

- An object (e.g. result of an operation) gets a name assigned (variable name)
- **Name = Objec**t
  Name should point to Object
- **Name1 = Name2**
  Name1 should point to the same object to which Name2 points
- When using a variable name in an expresssion, it will be substituted with the object it points to.
- There are **no “classic” variables** in Python, just **pointers/aliases**!

```
>>> a = 1
>>> a
1
>>> b = a
>>> b
1
>>> a = 2
>>> a
2
>>> b
1
>>> a + b
3
```

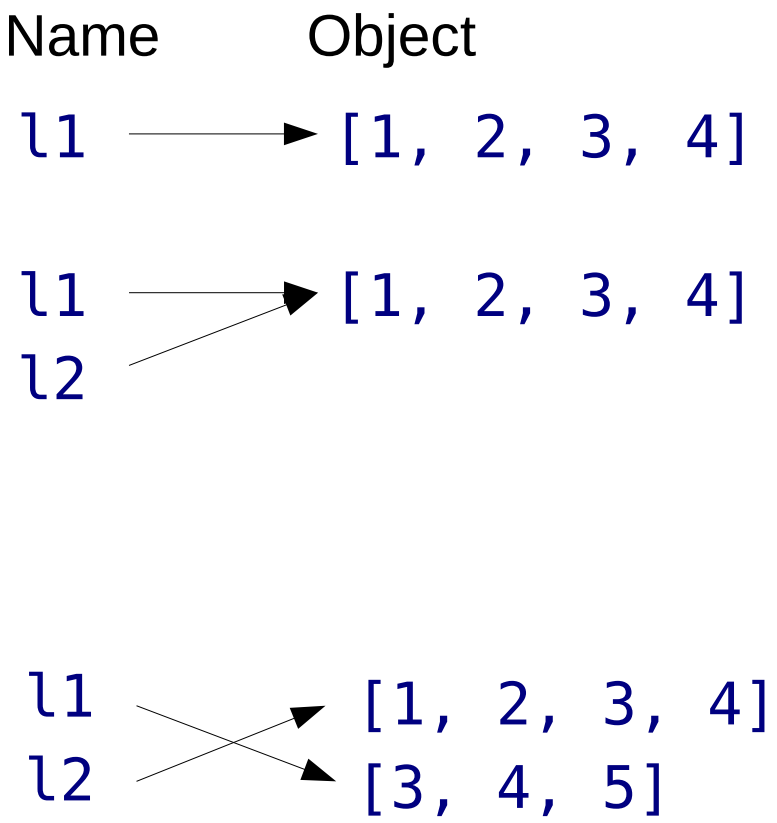


Assignment of mutable types

- Analogous to immutable types

```
l1 = [1, 2, 3, 4]
l2 = l1
l1
[1, 2, 3, 4]
l2
[1, 2, 3, 4]

l1 = [3, 4, 5]
l1
[3, 4, 5]
l2
[1, 2, 3, 4]
```

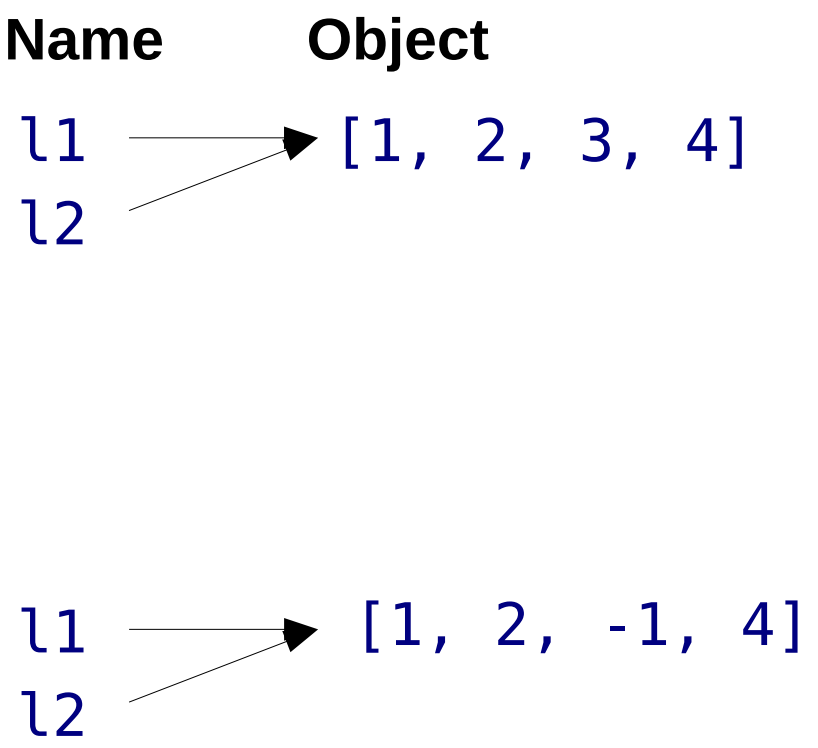


Assignment of mutable types

- If the content of a mutable variable is changed, the change is apparent in all variables, which are associated with that instance

```
l1 = [1, 2, 3, 4]
l2 = l1
l1
[1, 2, 3, 4]
l2
[1, 2, 3, 4]

l1[2] = -1
l1
[1, 2, -1, 4]
l2
[1, 2, -1, 4]
```



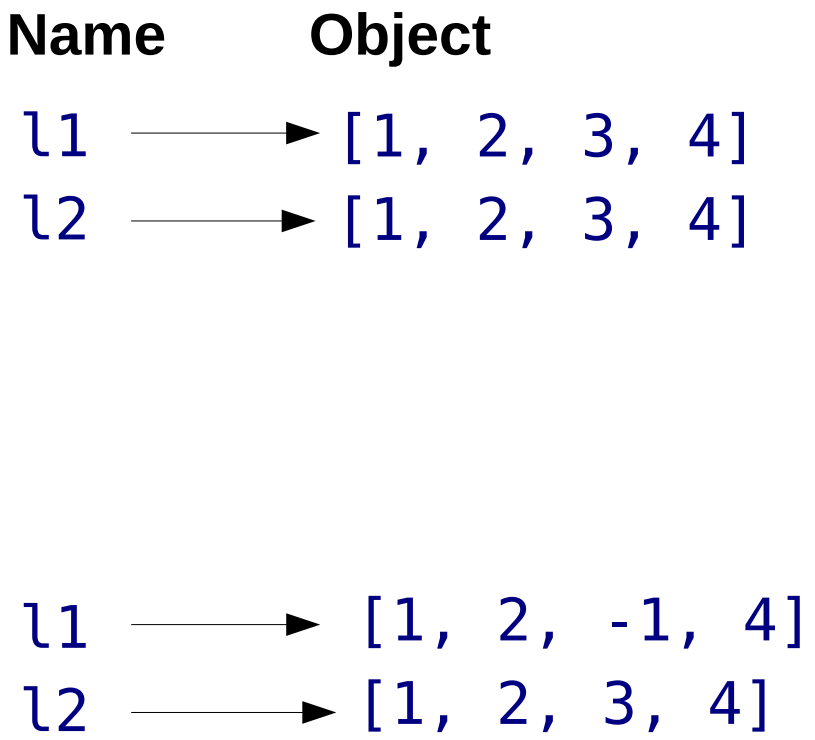
- Efficient, no copy is made
- Watch out for unwanted side effects with mutable types

Assignment of mutable types

- If a copy is needed, it must be explicetly created
- Try to avoid making copies, unless really necessary

```
l1 = [1, 2, 3, 4]
l2 = list(l1)
l1
[1, 2, 3, 4]
l2
[1, 2, 3, 4]

l1[2] = -1
l1
[1, 2, -1, 4]
l2
[1, 2, 3, 4]
```

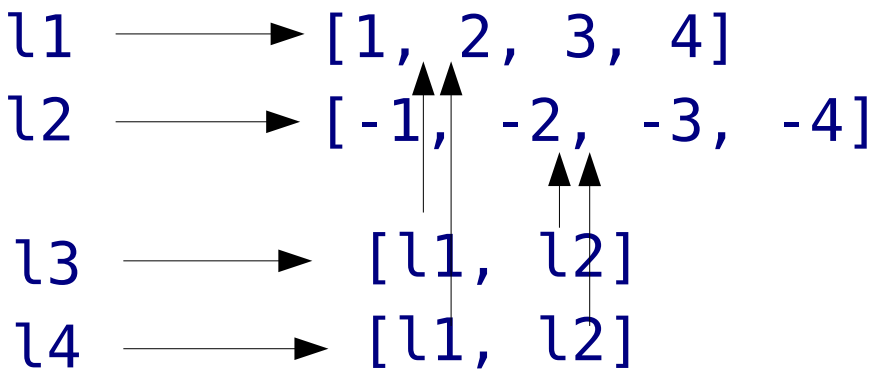


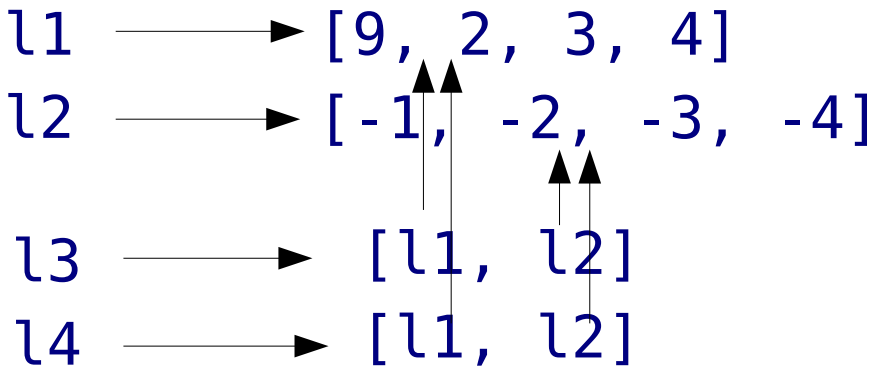
Assignment of mutable types

- If you copy a nested mutable object, only top layer is copied (**shallow copy**)

```
l1 = [1, 2, 3, 4]
l2 = [-1, -2, -3, -4]
l3 = [l1, l2]
l4 = list(l3)
l3
[[1, 2, 3, 4], [-1, -2, -3, -4]]
l4
[[1, 2, 3, 4], [-1, -2, -3, -4]]

l3[0][0] = 9
l3
[[9, 2, 3, 4], [-1, -2, -3, -4]]
l4
[[9, 2, 3, 4], [-1, -2, -3, -4]]
l1
[9, 2, 3, 4]
```





- Function **deepcopy()** in module copy can be used, if true nested copy is needed

Tuple/List operations

- The **+** operator creates a new list by concatenation:

```
l1 = [1, 2, 3]
l2 = [4, 5, 6]
l1 + l2
[1, 2, 3, 4, 5, 6]
```

- The ***** operator creates a new list by repetition

```
l1 = [1, 2, 3]
l1 * 2
[1, 2, 3, 1, 2, 3]
```

In-place operations

- In-place operations store the result of an arithmetic operation in the first operand:

```
aa += bb
aa -= bb
aa *= bb
aa /= bb
aa //= bb
```

equiv.

```
aa = aa + bb
aa = aa - bb
aa = aa * bb
aa = aa / bb
aa = aa // bb
```

- For mutable objects it can help to avoid creating unnecessary copies

```
long = [1, 2, … ]
short = [-1, -2]
```

```
long = long + short
```

Creates a temporary copy of long, extends it with short and replaces long with the result

```
long += short
```

Makes an in-place addition (usually without temporary copy)

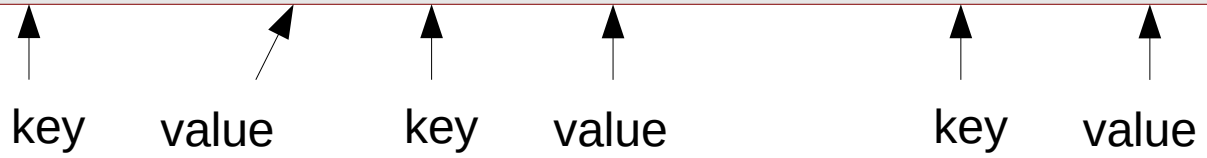
```
long.extend(short)
```

Extends list directly without temporary copy

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

- Empty dictionary can be created by **{}**

```
d0 = {}
d0
{}
```

- Number of key/value pairs can be queried by the **len()** function

```
len(d0)
0
```

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
```

Sets

- Sets contain only keys (like dictionaries), but no values
- 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'}
```

Set

- 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
```

Lists, sets, dictionaries – summary

Lists

- Ordered, elements are identified by their unique position (index)
- Fast *O(1)* access, if index of the element is known
- Slow O(N) access, if index is not known (e.g. looking for an element with given value)

Dictionary

- Unordered, elements identified by their unique key
- Fast O(1) access, if key of an element is known
- Slow O(N) access, if key is not known (e.g. looking for an element with given value)

Sets

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

Containers as iterators

- All containers can be used as iterators (e.g. in for-loops)
- Lists and tuples return their elements ordered by their index (position)

```
ll = [1, "test", 12.6, -1+3j]
for item in ll:
    print("Next item: ", item)
```

```
Next item:  1
Next item:  test
Next item:  12.6
Next item:  (-1+3j)
```

- Sets return their element 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
```

Containers as iterators

- Dictionaries return their keys one by one, but the order is undetermined:

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

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

Enumerate

- If within an iteration you need both, the iterator value and the current iteration number, you can use the **enumerate()** iterator
- **enumerate()** returns a new iterator over tuples containing the current iteration number and the value from the passed iterator

```
ll = [1, 'test', 12.6, (-1+3j)]
for ind, item in enumerate(ll):
    print("Item {:d}: {}".format(ind, item))
```

equivalent

```
Item 0: 1
Item 1: test
Item 2: 12.6
Item 3: (-1+3j)
```

```
for ind in range(len(ll)):
    print("Item {:d}: {}".format(ind, ll[ind]))
```

Initializing containers with iterators

- Most containers can be created from arbitary iterators
- The container will be filled up with the elements of the iterators as if they had been added one by one

```
list('test')
['t', 'e', 's', 't']

set('test')
{'e', 's', 't'}

set([1, 2, 4, 2, 1])
{1, 2, 4}

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

Every string can be used as an iterator over the charaters in it

If the container does not support multiple entries, they will become unique

Dictionary needs an iterator over (key, value) tuples

Comprehensions

- A comprehension can be used to create containers with a (slightly) modified or filtered content of an iterator

List comprehension

filtering is optional

```
[expr for itervar in iterator if condition]
```

```
words = ["Wort", "Word", "WORT", "word"]
loweredwords = [word.lower() for word in words]
loweredwords
['wort', 'word', 'wort', 'word']
```

Converts every character in a string to lowercase

```
nums = [1, 3, 2, 9, 8, 3]
oddsquares = [num**2 for num in nums if num % 2]
oddsquares
[1, 9, 81, 9]
```

Comprehensions

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}
oddsquares
{1, 9, 81}
```

Dictionary comprehension

filtering is optional

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

```
oddsquares = {num: num**2 for num in nums if num % 2}
oddsquares
{1: 1, 3: 9, 9: 81}
```

Comparison

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

- Ordered (sequence) types 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)          False
(9, "ahoi") > (6, "hello")     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'
"Word".upper()
'WORD'
```

lstrip(), rstrip(), strip()

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

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

See Python Standard Library docs: String methods