Python

Lists, tuples

Prof. Dr. Thomas Kopinski

Python - tuple

Tuple:

- ordered sequence of elements
- can hold various data types
- immutable (cf. strings)
- examples:

```
#tuples, lists
t = ()

t1 = (1,2,3)
t2 = ('a', 'b')
t3 = (10, 'name', 2.0)

print(t3[2])

print('id of t2' + str(id(t2)))

t2 += ('c', 'd')

print('id of t2' + str(id(t2)))

print('slicing a tuple: ' + str(t2[2:3]))
```

Python - tuple

Tuple:

- use, if you want to return more than one value from a function

```
def returnAllFromModulo(i, j):

    res = i//j
    rem = i%j

    return (res, rem)

print(returnAllFromModulo(10,7))
```

```
(1, 3)
```

- use, if you want to swap values easily

```
a = 100
b = 200

(b,a) = (a,b)

print(a)
print(b)
```

```
200
100
```

Python - list

List:

- ordered sequence of elements
- access via index as in tuples
- mutable type!
- can contain different types

```
#lists
                empty list
l = []
l = [1, 2, 3, 'this', ('a', 'b'), [1, 2]]

s = l[4]
print(id(s))          can assign parts to other vars
print(id(l[4]))

l1 = [1, 2, 3, 4]
print(id(l1))

l1[0] = 10
print(id(l1))

                use dot notation for operations
l1.append(5)
print(id(l1))
```

container for different types

this mutates the list

Python - list

List:

- ordered sequence of elements
- access via index as in tuples
- mutable type!
- can contain different types

```
#lists

l = []
l = [1, 2, 3, 'this', ('a', 'b'), [1, 2]]

s = l[4]
print(id(s))
print(id(l[4]))

l1 = [1, 2, 3, 4]
print(id(l1))

l1[0] = 10
print(id(l1))

l1.append(5)
print(id(l1))
```

```
140463417073824
140463417073824
```

```
140463416246752
140463416246752
140463416246752
```

Python - list

```
#operations on lists

l2 = [5, 6, 7, 8]

l3 = l1 + l2 #concatenation via +

l1.extend([5, 6, 7, 8])

print(l3)
print(l1)
```

```
[10, 2, 3, 4, 5, 5, 6, 7, 8]
[10, 2, 3, 4, 5, 5, 6, 7, 8]
```

```
print(l1)
del(l1[2]) #remove with index
print(l1)

l1.pop() #remove and return last element
print(l1)

r = l1.pop(2) #remove and return indexed element
print(l1)
print(r)
```

```
[10, 2, 3, 4, 5, 5, 6, 7, 8]
[10, 2, 4, 5, 5, 6, 7, 8]
[10, 2, 4, 5, 5, 6, 7]
[10, 2, 5, 5, 6, 7]
4
```

Python - list

List:

- other list operations

https://docs.python.org/3/tutorial/datastructures.html

```
print(list('splitIt!'))
print('this, is a string'.split(','))

l4 = ['more', 'words', 'here']
print('_'.join(l4))
```

```
['s', 'p', 'l', 'i', 't', 'I', 't', '!']
['this', ' is a string']
more_words_here
```

```
l5 = [3, 6, 1, 9, 5, 3]
rL = sorted(l5)
print(rL)
l5.sort()
print(l5)
l5.reverse()
print(l5)
```

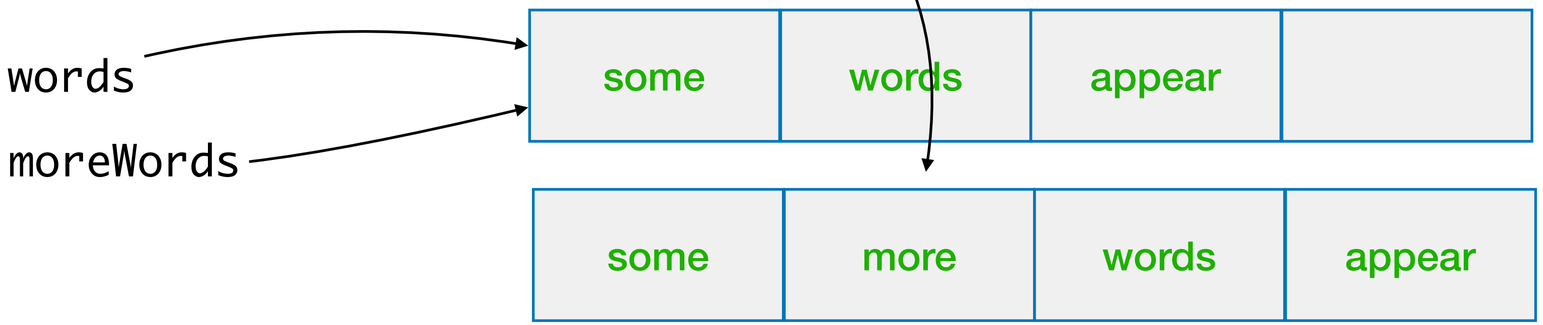
```
[1, 3, 3, 5, 6, 9]
[1, 3, 3, 5, 6, 9]
[9, 6, 5, 3, 3, 1]
```

Python - list

List:

- mutable objects
- is an object in memory
- variables point to this object
- variables pointing to this object are affected when it is changed

```
words = ['some', 'words', 'appear']
print (words)
moreWords = words
moreWords.insert(1, 'more')

print (words)
```



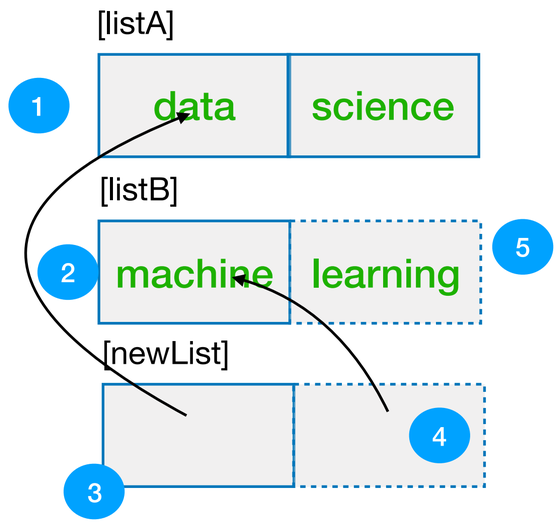
- cloning a list with [:] does not have the same effect!
  It leads to two separate objects!

Python - list

List:

- lists can be nested —> nested lists

```
# nested lists

listA = ['data', 'science']
listB = ['machine']

newList = [listA]

print (newList)

newList.append(listB)

print (newList)

listB.append('learning')
print (listB)
print (newList)
```




```
[['data', 'science']]
[['data', 'science'], ['machine']]
['machine', 'learning']
[['data', 'science'], ['machine', 'learning']]
```

Python - list

Task:

- write a function which remove duplicates from a list, using another list
- lists:
  - `listA = [10, 20, 50, 70]`
  - `listB = [10, 20, 60, 80]`
- function signature:
  - `removeDuplicates(l1, l2)`

```
listA = [10, 20, 50, 70]
listB = [10, 20, 60, 80]

def removeDuplicates1(l1, l2):
    for elem in l1:
        if elem in l2:
            l1.remove(elem)
    return(l1)


def removeDuplicates2(l1, l2):
    l3 = l1[:]
    for elem in l3:
        if elem in l2:
            l1.remove(elem)

    return(l1)
```

- Python uses an internal counter to keep track of the list index
- mutating lists while iterating over them does not update the counter
- rule: never update the list you are itering over!

Python - itemgetter

- `operator` module provides a set of operators
- `operator.itemgetter(n)` assumes some iterable (list, set, tuple) and gets the `n-th` element
-

```
from operator import itemgetter

a = [[20,30], [30,40],[10,10]]

f = itemgetter(1)
for e in a:
    print(f(e))
```

```
30
40
10
```

- `itemgetter (n)` constructs a callable which can then be applied to elements and returns their n-th element
-

```
print(sorted(a, key=f))
```

```
[[10, 10], [20, 30], [30, 40]]
```

```
print(sorted(a, key = lambda x: x[1]))
```

```
[[10, 10], [20, 30], [30, 40]]
```

Python - sorting

```
class Place:
    def __init__(self, name, population, state):
        self.name = name
        self.population = population
        self.state = state
    def __repr__(self):
        return repr((self.name, self.population, self.state))


places = [
    Place('Berlin', 3600000, 'Germany'),
    Place('Hamburg', 1800000, 'Germany'),
    Place('Helsinki', 648000, 'Finland')
]

print(sorted(places, key= lambda place: place.population))
```

```
[('Helsinki', 648000, 'Finland'), ('Hamburg', 1800000, 'Germany'),
 ('Berlin', 3600000, 'Germany')]
```

Python - sorting

```
places = [
    Place('Berlin', 3600000, 'Germany'),
    Place('Hamburg', 1800000, 'Germany'),
    Place('Helsinki', 648000, 'Finland')
]

places_tuples = [
                ('Berlin', 3600000, 'Germany'),
                ('Hamburg', 1800000, 'Germany'),
                ('Helsinki', 648000, 'Finland')
]

#print(sorted(places, key= lambda place: place.population))
print(sorted(places_tuples, key= itemgetter(1)))
print(sorted(places, key= attrgetter('population')))
```

```
[('Helsinki', 648000, 'Finland'), ('Hamburg', 1800000, 'Germany'),
 ('Berlin', 3600000, 'Germany')]
[('Helsinki', 648000, 'Finland'), ('Hamburg', 1800000, 'Germany'),
 ('Berlin', 3600000, 'Germany')]
```

Python - sorting

- `operator.itemgetter(n)` can take multiple arguments

```
#first by value at index 1, then by 2
print(sorted(places_tuples, key= itemgetter(1,2)))
#same as above, with attributes
print(sorted(places, key= attrgetter('population', 'state')))
```

```
print([(Place.name,Place.avg_density()) for Place in places])
```

```
[('Berlin', 4035.8744394618834), ('Hamburg', 2513.9664804469276),
('Helsinki', 858.2781456953643)]
```

- `operator.methodcaller(n)` calls a function on each object to sort by its result

```
print([(Place.name,Place.avg_density()) for Place in places])
print(sorted(places, key= methodcaller('avg_density')))
```

```
[('Helsinki', 648000, 'Finland', 755.0),
 ('Hamburg', 1800000, 'Germany', 716.0),
 ('Berlin', 3600000, 'Germany', 892.0)]
```

Python | List tasks

Task:

- **Task 1**: Write a program which extends a list (of integers) without using *.append()*
- **Task 2**: Write a routine which sorts a list of 3-tuples (of ints) with regard to the middle item; use *itemgetter()*
- **Task 3**: Recreate the following list of values with a list comprehension
  `[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]`
- **Task 4** : `m = [[1,2], [3,4], [5,6], [7,8]]` — switch the elements in this list using list comprehensions
- **Task 5:** Using list comprehensions - change the list
  `m = [[1,2,3,4], [3,4,5,6], [5,6,7,8], [7,8,9,10]]`
  to
  `m = [[4,1,2,3], [6,3,4,5], [8,5,6,7], [10,7,8,9]]`