Python for Language Processing

(3a) Collections

Dr. Jakob Prange

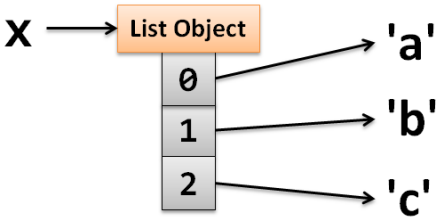
Fakultät für Angewandte Informatik - Universität Augsburg

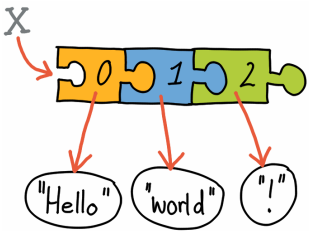
CL Fall School 24

Credit: This course is based on material developed by
Annemarie Friedrich, Stefan Thater, Michaela Regneri, and Marc Schulder at Saarland University

List Objects

```
1  x = ['a', 'b', 'c']
2  print("x[0] is: ", x[0])
3  print("x[1] is: ", x[1])
4  print("x[2] is: ", x[2])
```





List Indices

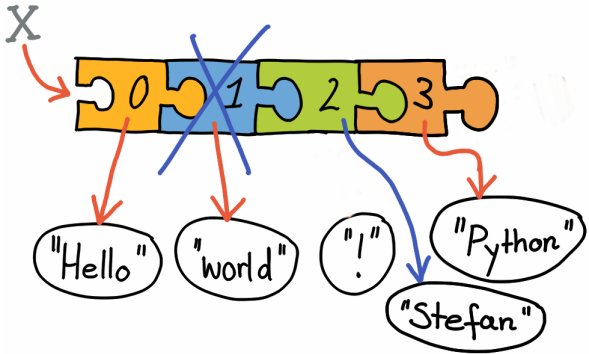
```
1    myList = ["a", "b", "c", "d", "hello"]
2    myList[3] = "world"
```

0	1	2	3	4
"a"	"b"	"c"	"d"	"hello"
-5	-4	-3	-2	-1

What does the following piece of code print out?

```
1    print(myList[1])
2    print(myList[4])
3    print(myList[-2])
4    print(myList[4][1])
```

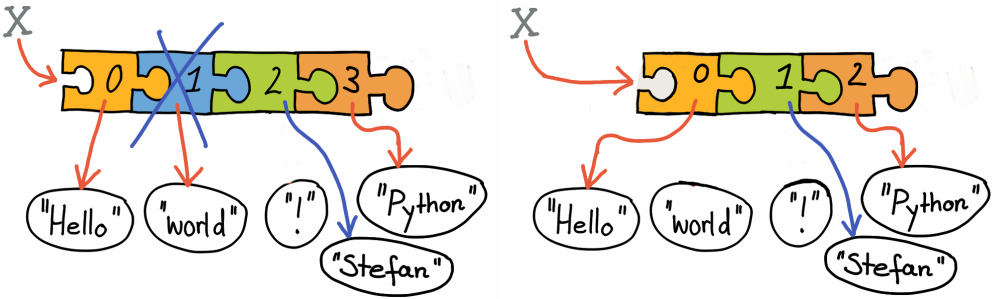
Appending and Replacing Items in Lists




```
1       x = ["Hello", "world", "!"]
2       x.append("Python")
3       x[2] = "Stefan"
```

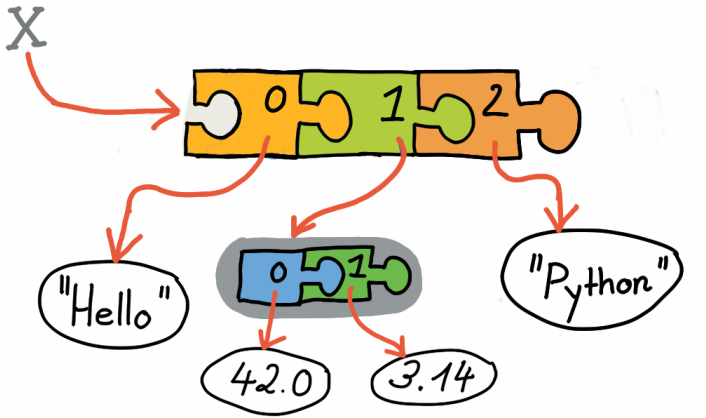
How can we delete a list item?

Removing Items from Lists




```
1    x = ["Hello", "world", "!"]
2    x.append("Python")
3    x[2] = "Stefan"
4    del x[1]
```

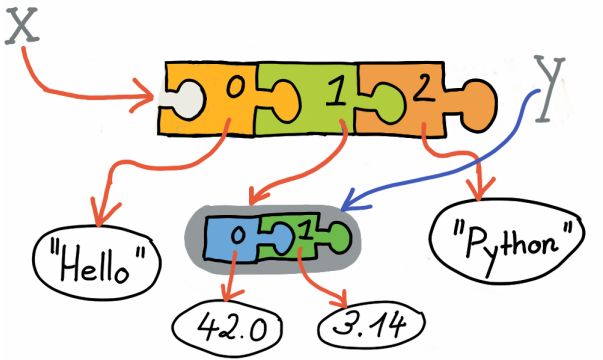
Multidimensional Lists



Lists can be **nested**. Why do we need to be aware of this?

```
1       x = ["Hello", [42.0, 3.14], "Python"]
```

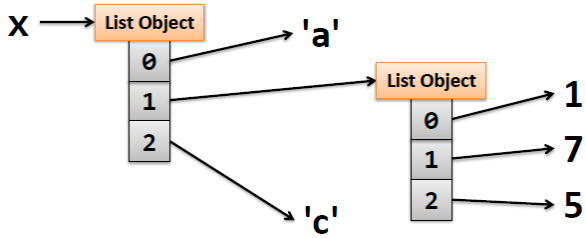
Multidimensional Lists




```
1    x = ["Hello", [42.0, 3.14], "Python"]
2    y = x[1]
3    y[1] = 0.0
4    print(x[1][1])
```

Multidimensional Lists

```
1    x = ['a', [1, 7, 5], 'c']
```




```
1    print("x[0] is: ", x[0])
2    print("x[1] is: ", x[1])
3    print("x[2] is: ", x[2])
4    print("x[1][0] is: ", x[1][0])
5    print("x[1][1] is: ", x[1][1])
6    print("x[1][2] is: ", x[1][2])
```

Multidimensional Lists

```
1    myList = ["a", "b", [1, 2, 3], "d", "e"]
2    myList[3] = [4, 5, 6]
```

0	1	2	3	4
"a"	"b"	[1,2,3]	[4,5,6]	"e"

a) What does the following piece of code print out?

```
1    print(myList[1])
2    print(myList[2][0])
```

b) How can you access the '5'?

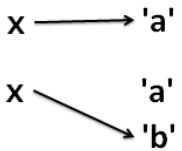
c) What happens in the following cases?

```
1    print(myList[5])
2    print(myList[2][3])
```

Mutable vs. immutable types

Mutable objects	**Immutable objects**
list	integer, float, string, boolean, ...
can be changed (items can be added, removed, modified)	*never changes, assignments result in new objects*

```
1    x = 'a'
2    x = 'b'
3    # the string object 'a' is NOT
4    # modified:
5    # 'b' is a new string object!
```

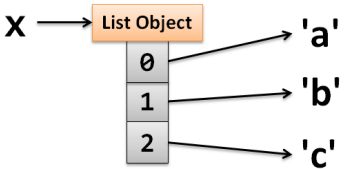


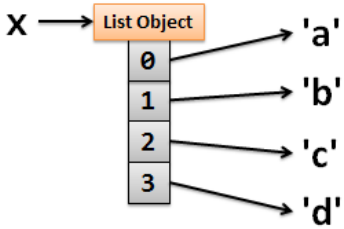
Mutable vs. immutable types

Mutable objects	Immutable objects
list	integer, float, string, boolean, ...
can be changed (items can be added, removed, modified)	never changes, assignments result in new objects

```
1    x = ['a', 'b', 'c']
2    x.append('d')
```

x still points to the same list object which has been modified!

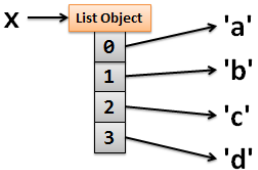


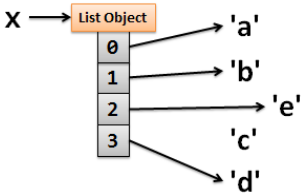


Mutable vs. immutable types

Mutable objects	Immutable objects
list	integer, float, string, boolean, ...
can be changed (items can be added, removed, modified)	never changes, assignments result in new objects

```
1   x = ['a', 'b', 'c']
2   x.append('d')
3   x[2] = 'e'
4   # x still points to the
5   # same list object,
6   # which has been modified!
7   # the string object 'c'
8   # has not been modified,
9   # 'e' is a new string
10  # object!
```



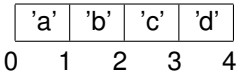


List Methods

- **Methods** = functions which are applied 'on an object'
- `someObject.methodName(parameters)`
- usually change the object 'on which they are called'

```
1    someList = [1, 2, 3]
2    someList.append(5)
3
4    # appends 5 to the list
5    >>> print(someList)
6    [1, 2, 3, 5]
```

Slicing



Slicing creates **copies** of the list objects!

```
>>> myList = ['a', 'b', 'c', 'd']
>>> myList[0:3]
['a', 'b', 'c']
>>> myList[2:3]
['c']
>>> myList = ['a', 'b', 'c', 'd']
>>> myList[0:3]
['a', 'b', 'c']
>>> myList[2:4]
['c', 'd']
>>> myList[1:]
['b', 'c', 'd']
>>> myList[:3]
['a', 'b', 'c']
>>> myList[:]
['a', 'b', 'c', 'd']
```

String Immutability

- Strings are also sequences (of strings: one character at a time)
- We can access the items of a string:

```
1    >>> myString = "telephone"
2    >>> print(myString[2])
3    l
4    >>> print(myString[4:]) # copy!
5    phone
```

- Strings are immutable sequences ⇒ They can't change.
  `myString[0] = "T"` ⇒ DOES NOT WORK!
- Concatenation creates new strings.
  `myString = "T" + myString[1:]`

Shared References

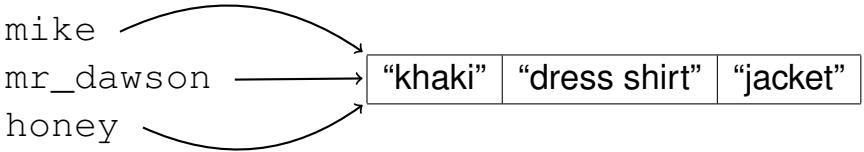
- Variables do not *contain* values (like a tupper box contains food)
- Variables *point* to positions in the memory, like a name points to a person (the name does not contain the person), and the memory positions contain the values.

```
>>> mike = ["khakis", "dress shirt", "jacket"]
>>> mr_dawson = mike
>>> honey = mike
>>> mike
['khakis', 'dress shirt', 'jacket']
>>> mr_dawson
['khakis', 'dress shirt', 'jacket']
>>> honey
['khakis', 'dress shirt', 'jacket']
```

Examples taken from M. Dawson, Python Programming for the Absolute Beginner, 3rd edition, Course Technology, p.138

Shared References

```
1    >>> mike = ["khakis", "dress shirt", "jacket"]
2    >>> mr_dawson = mike
3    >>> honey = mike
```

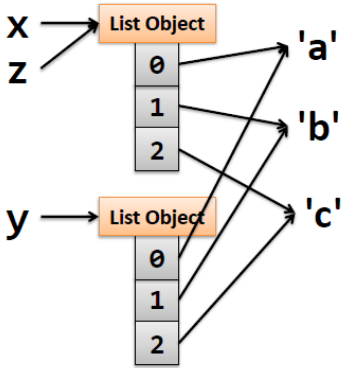



```
1    >>> honey[2] = "red sweater"
2    >>> honey
3    ['khakis', 'dress shirt', 'red sweater']
4    >>> mike
5    ['khakis', 'dress shirt', 'red sweater']
```

Copying a List (Shallow Copy)

- Slicing can be used to create a *shallow* copy of a list.
- A *shallow* copy constructs a new list object and then inserts references into it to the objects found in the original.
  (http://docs.python.org/3.2/library/copy.html)
- The **is** operator tells you whether two variables point to the same list object.

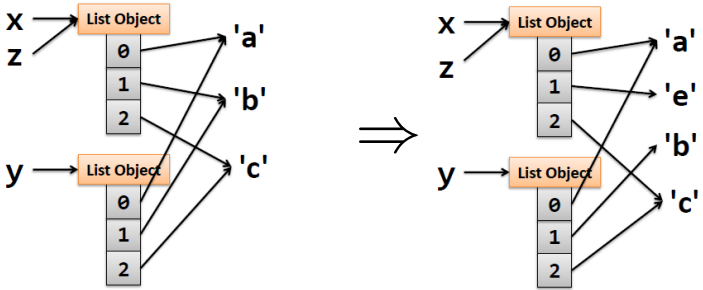
```
1    >>> x = ['a', 'b', 'c']
2    >>> z = x
3    >>> y = x[:]
4    >>> y
5    ['a', 'b', 'c']
6    >>> z is x
7    True
8    >>> y is x
9    False
```



Copying a List (Shallow Copy)

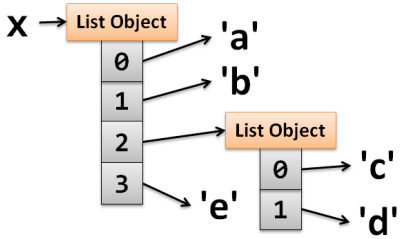
```
1    >>> x = ['a', 'b', 'c']
2    >>> z = x
3    >>> y = x[:]
4    >>> y
5    ['a', 'b', 'c']
```

```
1    >>> x[1] = 'e'
2    >>> x
3    ['a', 'e', 'c']
4    >>> y
5    ['a', 'b', 'c']
6    >>> z
7    # what is printed?
```



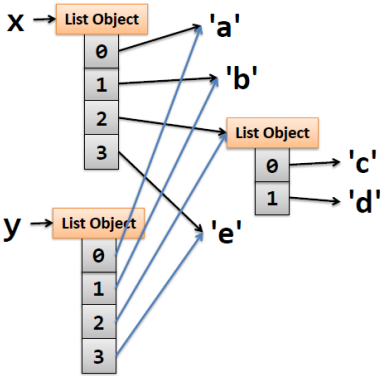
Shallow Copy of a Nested List

```
1    >>> x = ['a', 'b',
2           ['c', 'd'], 'e']
```



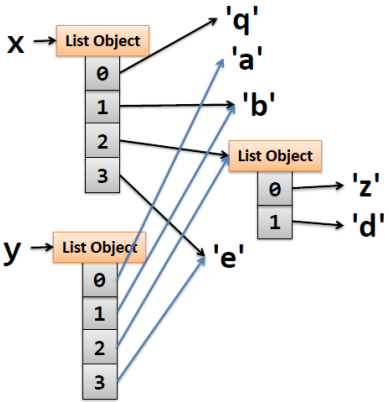
Shallow Copy of a Nested List

```
1    >>> x = ['a', 'b',
2           ['c', 'd'], 'e']
3    >>> y = x[:]
4    >>> y
5    ['a', 'b', ['c', 'd'], 'e']
```



Shallow Copy of a Nested List

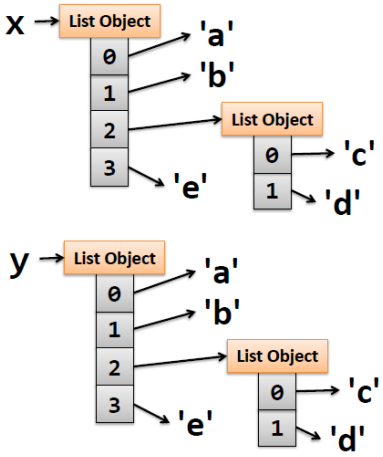
```
>>> x[2][0] = 'z'
>>> x
['a', 'b', ['z', 'd'], 'e']
>>> y
['a', 'b', ['z', 'd'], 'e']
>>> x[0] = 'q'
# a new string object 'q' is
# created, 'a' is not changed
>>> x
['q', 'b', ['z', 'd'], 'e']
>>> y
['a', 'b', ['z', 'd'], 'e']
```



Deep Copy of a Nested List

- A *deep* copy constructs a new list object and then inserts copies into it of the objects found in the original. Careful: can be 'expensive'!

```
>>> x = ['a', 'b',
         ['c', 'd'], 'e']
>>> from copy import deepcopy
>>> y = deepcopy(x)
>>> y
['a', 'b', ['c', 'd'], 'e']
>>> x[0] = 'q'
>>> y[2][1] = 'z'
>>> x
['q', 'b', ['c', 'd'], 'e']
>>> y
['a', 'b', ['c', 'z'], 'e']
```



Comparing Values

```
>>> x = [1, [2, 3], 4]
>>> y = x[:] # make a shallow copy of x
>>> # Do x and y contain the same values?
>>> x == y
True
```

- **var1 == var2** tells you whether the values of **var1** and **var2** are equal (no matter whether or not they are in the same memory location).

Excursus: Checking References

```
>>> x = [1, [2, 3], 4]
>>> y = x[:] # make a shallow copy of x
>>> # Do x and y point to the same memory location?
>>> # = Are x and y the same list object?
>>> x is y
False
>>> # Do x and y contain the same sublist?
>>> x[1] is y[1]
True
```

- **`var1 is var2`** tells you whether **`var1`** and **`var2`** point to the same memory location.
- *Careful: If you compare values of immutable types using the `is` operator, you might see some unexpected behavior. We will get back to this later. For now, you can use the `is` operator to check whether two variables point to the same mutable values (e.g., lists, sets).*

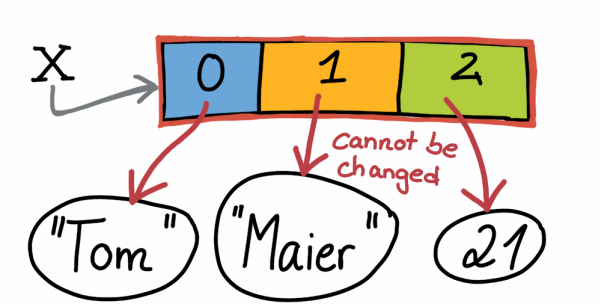
No Value: `None`

- `None` is Python's way of representing 'nothing'.
- Placeholder for any value.
- Variable does not point anywhere.
- Can be used for initialization as follows:

```
1    x = None
2    while x != "":
3      print("Press Enter to exit.")
4      x = input("Type something to be printed.")
5      if x:
6        print(x)
```

Tuples

- Basically like lists, but cannot be changed (**immutable**).

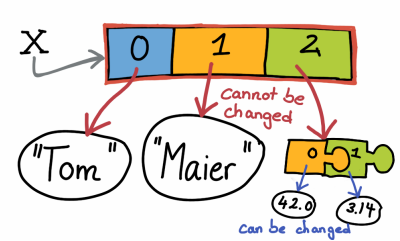



```
1    tom = ("Tom", "Maier", 21)
```

Tuples

```
1   tom = ("Tom", "Maier", 21, "German")
2   carlos = ("Carlos", "Sanchez", 23, "Spanish")
3
4   # trying to assign new value to tuple index:
5   >>> tom[2] = 22
6   Traceback (most recent call last):
7     File "<stdin>", line 1, in <module>
8   TypeError: 'tuple' object does not support
9   item assignment
10
11  # trying to append value to tuple:
12  >>> carlos.append("Barcelona")
13  Traceback (most recent call last):
14    File "<stdin>", line 1, in <module>
15  AttributeError: 'tuple' object has no attribute
16  'append'
```

Tuples



Items within tuples may be mutable

```
1    >>> x = ("Tom", "Maier", [42.0, 3.14])
2    >>> x
3    ('Tom', 'Maier', [42.0, 3.14])
4    >>> x[2][1] = 0.0
5    >>> x
6    ('Tom', 'Maier', [42.0, 0.0])
```

Tuples

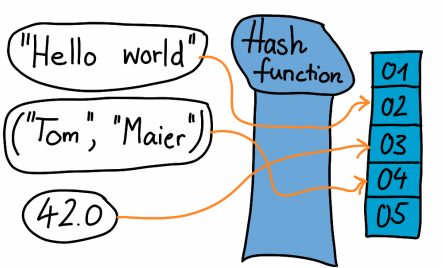
- Tuples are immutable: You cannot change them.
  - But (as with strings) you can create new tuples from existing ones (e.g. concatenation, slicing)

```
>>> carlos = ('Carlos', 'Sanchez', 23, 'Spanish')
>>> tom = ("Tom", "Maier", 21, "German")
>>> carlos2 = carlos[0:2] + tom[2:]
>>> print(carlos2)
('Carlos', 'Sanchez', 21, 'German')
```

- Empty tuples $\Rightarrow$ False.
- Advantage VS. lists
  - faster
  - perfect for creating constants
  - if they contain only immutable values: **hashable** (next slide)
- Tuple with one element: `myConstant = (value,)`

Hashable Objects

- Map immutable objects of arbitrary size to fixed size objects (e.g. an integer): used internally in Python (more on this later)
- All **immutable** objects are hashable.
- test using `hash()`




```
1  >>> hash(42.0)
2  42
3  >>> hash("Hello world")
4  -8673689256988262017
5  >>> hash(("Tom", "Maier"))
6  7027136242295325986
7  >>> hash([1,2,3])
8  TypeError:
9  unhashable type:
10 'list'
```

Hashable Objects

- Tuples are hashable if they don't contain any mutable objects.

```
>>> x = ("Carlos", "Sanchez", 21)
>>> hash(x)
-2458947529095456877
>>> y = ("Carlos", "Sanchez", [21, 42])
>>> hash(y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Set

- unordered, can contain each element at most once
- may only contain **hashable** types (numbers, strings, booleans,...)

```
>>> mySet = {5, 3, 21}
>>> mySet = set([3, 5, 3, 21, 4])
>>> mySet
{3, 5, 21, 4}
>>> mySet.add(7)
>>> mySet
{3, 5, 21, 4, 7}
>>> mySet.remove(5)
>>> mySet
{3, 21, 4, 7}
```

- What is the effect of `set()`, how is it applied above?
- Empty set: `set()`
- Careful: `{}` creates an empty dictionary, not an empty set!

Set

- One **cannot** access items of a set via indices

```
1  x = set([5, 2, 6, 1])
2
3  print(x[1])
4  >> Traceback (most recent call last):
5  >> File "<stdin>", line 1, in <module>
6  >> TypeError: 'set' object does not support indexing
7
8  # iterate over set
9  for i in x:
10   print(i)
11
12 # if order matters:
13 for i in sorted(x):
14   print(i)
```

frozenset

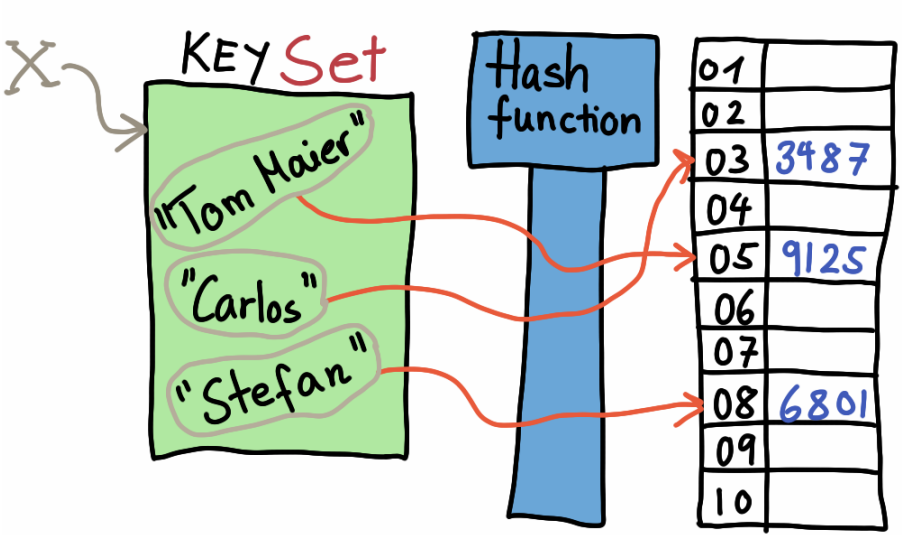
- `set`s are mutable
  - can only put immutable / hashable values in set
  - set itself is mutable: can add / remove items
- `frozenset` works like a set, but all methods that alter the set (inserting, deleting, changing) are prohibited
- Instantiation: `freezing = frozenset(['snow', 'hail'])`

Type Conversion of Collections

- All collections (except for dictionaries) can be converted into each other.

```
>>> a = set([1,2])
>>> b = list(a)
>>> c = tuple(a)
>>> d = tuple(b)
>>> e = set(b *5)
...
```

Dictionaries (= Hashmaps)



Dictionaries

- Access a value using a key (but not vice versa!)
- Each key may occur only once per dictionary.
- Keys have to be **hashable** (strings, numbers, ...)
- Values don't have to be unique (or hashable).
- Empty dictionary: `phoneBook = dict()` or `phoneBook = {}`

PHONE BOOK

KEY	VALUE
Smith	3253
Johnson	3938
Brown	1443
Miller	9388

```
>>> phoneBook = {"Smith" : 3253,
                 "Johnson" : 3938,
                 "Brown" : 1443}
>>> print(phoneBook["Smith"])
3253
```

Dictionaries

PHONE BOOK

KEY	VALUE
Smith	3253
Johnson	3938
Brown	1443
Miller	9388

```
1 phoneBook = {}
2 phoneBook["Smith"] = 3253
3 phoneBook["Johnson"] = 3938
4 phoneBook["Brown"] = 1443
5 phoneBook["Miller"] = 9388
```

Dictionaries - Check existence of key I

- Good style: Check for existence of a key using the `in` operator.
- `phoneBook` stands for the set of keys of this dictionary here

```
1  person = input("Name: ")
2  if person in phoneBook:
3    print(person, "=", phoneBook[person])
4  else:
5    print("Sorry, I don't know this person.")
```

Dictionaries - Views I

- Views show the current state of a dictionary.
- dict.keys() - returns a view of all the keys of a dictionary.
- dict.values() - returns a view of all the values of a dictionary.
- Views are like lists, except we can't change them.
  Usage: Iterate over views or convert to list.

```
1    phoneBook = {'Brown': 1443, 'Smith': 3253}
2
3    persons = phoneBook.keys()
4    for person in persons:
5        print(person, ">>", phoneBook[person])
```

Dictionaries - Views II

- dict.items() - returns a view of all the items in a dictionary. Each item is a tuple: `(key, value)`.
- In each iteration, `items()` returns a `(key, value)` tuple.

```
1    phoneBook = {"Brown" : 1443, "Smith" : 3253}
2
3    entries = phoneBook.items()
4    for entry in entries:
5        print(entry[0], ">>", entry[1])
6
7    # Iterating over key-value pairs using tuples
8    for (name, number) in phoneBook.items():
9        print("Name:", name, "\tNumber:", number)
```

Dictionaries - Changing Entries

```
>>> phoneBook
{'Brown': 1443, 'Smith': 3253, 'Johnson': 3938}
>>> phoneBook['Smith'] = 7777
>>> phoneBook
{'Brown': 1443, 'Smith': 7777, 'Johnson': 3938}
>>> phoneBook["Miller"] = 9388
>>> phoneBook
{'Brown': 1443, 'Smith': 7777, 'Johnson': 3938,
   'Miller': 9388}
```

⇒ **Assignment of value to key**

- If key exists, value is overwritten.
- If key does not exists, new entry is created.

Dictionaries - Comparison of Keys

- Keys are compared using the == operator.
- Does it return `True`?
- 1 and 1.0 are the same key!

Dictionaries - Deleting Entries

```
>>> phoneBook = {'Brown': 1443, 'Smith': 3253,
    'Johnson': 3938}
>>> del phoneBook["Smith"]
>>> phoneBook
{'Brown': 1443, 'Johnson': 3938}
>>> number = phoneBook.pop("Johnson")
>>> phoneBook
{'Brown': 1443}
>>> number
3938
```

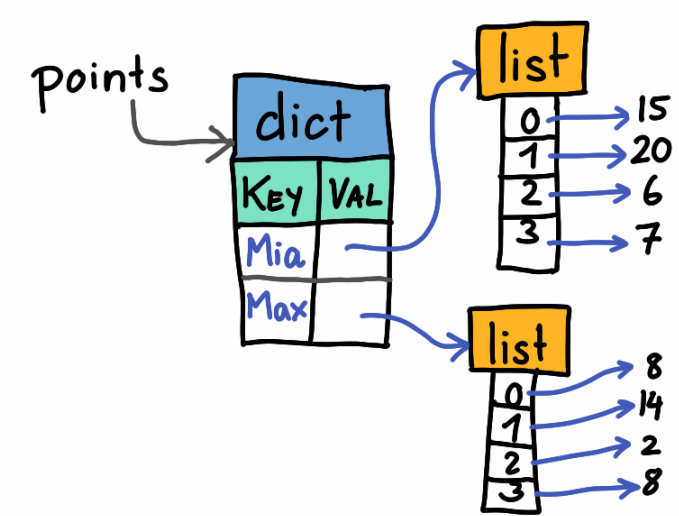
- Check whether a key exists before deleting!
- `del dict[key]` does not return anyting.
- `dict.pop(key)` returns the value.

Dictionaries - Values

- **Keys** ⇒ ONLY hashable values.
  Which types can we use as dictionary keys?
- **Values** ⇒ any types possible.

```
1    # number of points achieved in exercise sheets
2    points = {}
3    points["Mia"] = [15, 20, 6, 7]
4    points["Max"] = [8, 14, 2, 8]
5    print(points)
6    # {'Max': [8, 14, 2, 8], 'Mia': [15, 20, 6, 7]}
7
8    # points in 2nd exercise sheet
9    for key in points:
10       print(key, ':', points[key][1])
11   # Mia : 20
12   # Max : 14
```

Dictionaries – Values

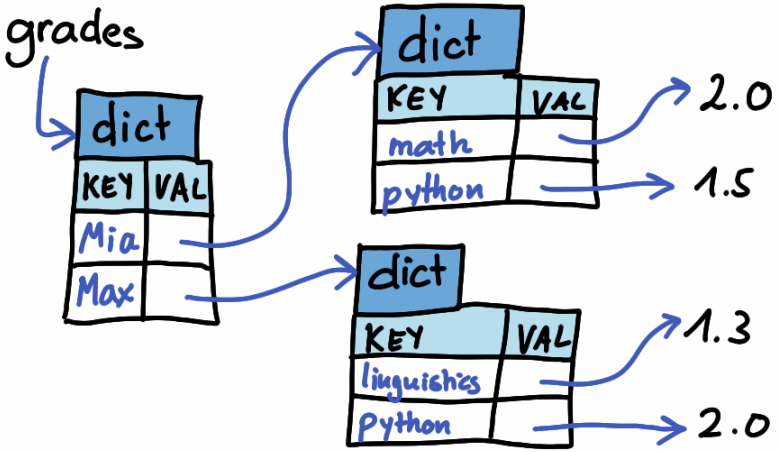


Dictionaries - Values

What is the type of the values in this example?

```
1 # cities visited
2 cities = {}
3 cities["Marc"] = {"New York", "San Francisco", "Paris"}
4 cities["Stefan"] = {"London", "Chiang Mai", "Paris"}
5 commonCities = cities["Marc"].intersection(cities["Stefan"])
6 print(commonCities)
```

Dictionaries – Values



Dictionaries - Values

- Especially when using complicated dictionaries, make sure you are not overwriting keys, and check keys before accessing values.
- You need to create the 'inner' dictionaries for each key of the 'outer' dictionaries.

```
1  grades = {}
2  grades["Mia"] = {"math" : 2.0, "python" : 1.5}
3  grades["Max"] = {"linguistics" : 1.3, "python" : 2.0}
4  # ask for a grade
5  grade = grades["Mia"]["python"]
6  print("In Python, Mia got a", grade, "!")
7  # In Python, Mia got a 1.5 !
8
9  # add a grade
10 grades["Max"]["math"] = 3.3
```

Defaultdict

```
>>> from collections import defaultdict
>>> d = defaultdict(int)
>>> d["Price"] = d["Price"] + 1
>>> d["Price"]
1
>>> d = defaultdict(list)
>>> d["someKeyThatDoesNotYetExist"]
[]
>>> d = defaultdict(str)
>>> d["someKeyThatDoesNotYetExist"]
''
```

- `str` ⇒ empty string
- `dict` ⇒ empty dictionary
- `set` ⇒ empty set

Summary: Mutability and Hashing

	Object/value	Type	immutable	hashable
1	`17`	**int**eger	✓	✓
2	`42.0`	**float**	✓	✓
3	`True`	**bool**ean	✓	✓
4	`"Python"`	**str**ing	✓	✓
5	`[1, 2, 3]`	**list**	X	X
6	`(1, 2, 3)`	**tuple**	✓	✓
7	`(1, 2, [3, 4])`	**tuple**	✓	X
8	`{3, 6, 2}`	**set**	X	X
9	`frozenset([3, 6])`	**frozenset**	✓	✓
10	`{"a":1, "b":2}`	**dict**ionary	X	X

Summary: Mutability and Hashing

- Immutable = Can not be changed after creation
  - Caveat: May contain mutable (i.e. changeable) objects
- Hashable = Can provide a hash value (call `hash(x)`)
  - Hashable objects must be immutable, else the hash value could change.
- All basic types (int, float, bool, str) are both immutable and hashable.
- Collections must be immutable to be hashable.
- Collections are only hashable if **all** their content is hashable, too!
- Lists and tuples can have any kind of items.
- Sets and frozensets can **only** have hashable items.
- Dictionary **keys** have to hashable, while their **values** can be any kind.

Advanced: Why do we need Hashing?

Hashing allows us to represent large objects as a single integer. Hashing is repeatable, e.g. calling hash("Hello") will always return the same value. How the hash is computed depends on the type and its implementation.

Unsorted collections (sets, dictionaries) use so called *hash tables* to store their object pointers, rather than indices. This makes looking up particular values a lot faster. For example `3 in {1,2,3}` is faster than `3 in [1,2,3]`, because we don't have to look through the entire list.

Hashable objects must be immutable, because if the object were to change, its hash value would also change. This would break the hash table lookup.