Containers in this course...

Some examples of containers in this course:

- **lists**
  
  ```python
  a = [1,2, 'some text', [-20.3, \texttt{sin}(0.8)]]
  ```

- **tuples**
  
  ```python
  b = 3,4,5, 'some text'
  ```

- **dictionaries**
  
  ```python
  c = { 'age':23, 'length':1.84, 'town':'Tomellila'}
  ```

- **sets**
  
  ```python
  d = { 'banana', 'apple', 'kiwi'}
  ```

We now look at these in more detail.
Lists and slicing

Slicing – \( L[i:j] \)

*Slicing* a list between index \( i \) and \( j \) is forming a new list by taking elements with indices starting at \( i \) and ending *just before* \( j \).

Alternatively ...

\( L[i:j] \) means: create a list by keeping all list elements from position \( i \) up to position \( j \) but not including the element at \( j \)!

Example

\[
L = ['C', 'l', 'o', 'u', 'd']
\]
\[
L[1:4] \quad # ['l', 'o', 'u']
\]

# start list at position 1 and end at position 4-1=3:
Partial slicing. Using:

One may omit the first or last bound of the slicing:

```python
L = ['C', 'l', 'o', 'u', 'd']

# take all elements except the element at position 0
L[1:]  # ['l', 'o', 'u', 'd']

# take all elements up to position 3 (not including)
L[:3]  # ['C', 'l', 'o']

# take all elements from position -2 to the end
L[-2:]  # ['u', 'd']  # negative indexing!

# take all elements up to position -2 (not including)
L[:2]  # ['C', 'l', 'o']

L[:]  # take the entire list
```

Mathematical Analogy

This is similar to half lines in $\mathbb{R}$: $(-\infty, a)$ means: take all numbers strictly lower than $a$, cf. syntax of $L[::j]$. 
Some more examples to test your understanding

Red signifies the part of the list you keep

<table>
<thead>
<tr>
<th>Example</th>
<th>List</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>L[2:]</code></td>
<td>0 1 2 3 ··· -3 -2 -1</td>
<td></td>
</tr>
<tr>
<td><code>L[:2]</code></td>
<td>0 1 2 3 ··· -3 -2 -1</td>
<td></td>
</tr>
<tr>
<td><code>L[:−2]</code></td>
<td>0 1 2 3 ··· -3 -2 -1</td>
<td></td>
</tr>
<tr>
<td><code>L[−2:]</code></td>
<td>0 1 2 3 ··· -3 -2 -1</td>
<td></td>
</tr>
<tr>
<td><code>L[2:−1]</code></td>
<td>0 1 2 3 ··· -3 -2 -1</td>
<td></td>
</tr>
<tr>
<td><code>L[2:5]</code></td>
<td>0 1 2 3 4 5 ··· -1</td>
<td></td>
</tr>
<tr>
<td><code>L[−4:−1]</code></td>
<td>0 1 ··· -5 -4 -3 -2 -1</td>
<td></td>
</tr>
</tbody>
</table>
Dictionaries

A dictionary is an unordered structure of (key, value) pairs. The objects, within the dictionary, are accessed by *keys* instead of *index*.

To define dictionaries we need to use square brackets:

```python
homework = {'Anna': True, 'Kerstin': False}

homework['Anna'] # True

# changing a value:
homework['Kerstin'] = True

# deleting an item
del homework['Anna']
```

*Note:* Keys must be objects which are *immutable* (unchangable), such as strings, numbers, Booleans. Lists cannot be used as keys.
Looping through Dictionaries

A dictionary is an object with the following useful methods:

- `keys`,
- `values` or `items`

By default a dictionary is considered as a list of `keys`:

```python
for key in homework:  # or homework.keys()
    print('{k} {res}'.format(k=key, res=homework[key]))
```

One may also use `items` to loop through both keys and values:

```python
for key, value in homework.items():
    print('{k} {v}'.format(k=key, v=value))
```

You can also print just the values by using the `values` method:

```python
for value in homework.values():
    print('{v}'.format(v=value))
```
Dictionaries in this course

Dictionaries are mainly used for

- providing functions with arguments in a compact way (see Unit 4).
- to collect options of a method:
  
  ```python
  {'tol':1.e-3,'step size':0.1,'maxit':1000}
  ```
Sets

Sets are containers which mimic sets in their mathematical sense:

Definition
A set is a collection of well defined and distinct objects, considered as an object in its own right. (Wikipedia)

```
fruitbasket = set( ['apple', 'pear', 'banana'] )
```

The most important operation is in, meaning ∈ (is element of):

```
'plum' in fruitbasket  # returns False
'pear' in fruitbasket  # returns True
```

Note: the following syntax can also be used to define a set,

```
fruitbasket = { 'mango', 'kiwi' }
```

Compare with dictionaries: { ’a’: ’mango’, ’b’:’kiwi’ }
Operations on Sets

Operations on sets are \( A \cap B \), \( A \cup B \) and \( A/B \) (intersection, union and relative complement):

```python
fruitbasket = set(['apple', 'pear', 'banana'])
rotten = set(['pear'])  # defines a new set
exotic = set(['mango', 'kiwi'])  # defines another set
emptyset = set([])  # defines the empty set

# Are all fruits edible (non-rotten)?
emptyset == fruitbasket.intersection(rotten)

# Add some diversity
extendedbasket = fruitbasket.union(exotic)

# The edible fruits
goodfruits = fruitbasket - rotten  # relative complement
```
No duplicate elements

“...distinct objects ...” (see mathematical definition)
In mathematics a set with elements \{a, b, a, a\} is the same as the set with elements \{a, b, b\} or the set with elements \{b, a\}.

This is reflected in Python by

```python
set({'apple', 'apple', 'pear'})
== set({'apple', 'pear'})  # True
```
Tuples - tuples are lists which do not change!

**Definition**

A *tuple* is an *immutable* list. Immutable means cannot be modified.

**Example**

```python
my_tuple = (1, 2, 3)  # our first tuple!
my_tuple = 1, 2, 3     # parentheses not required
len(my_tuple)          # 3, same method as for lists

my_tuple[0] = 'a'     # error! tuples are immutable

singleton = 1,        # note the comma
singleton = (1,)      # preferred for readability
len(singleton)         # 1
```
One may assign several variables at once by *unpacking* a list or tuple:

```python
a, b = 0, 1  # object a gets 0 and object b gets 1
a, b = [0, 1]  # exactly the same effect
(a, b) = 0, 1  # same
[a, b] = [0, 1]  # same thing for lists
```

**The swapping trick!**

Use packing and unpacking to *swap* the contents of two variables.

```python
a, b = b, a
```
Returning Multiple Values

A function may return several values:

```python
def argmin(L):  # return the minimum and index
    ...
    return minimum, minimum_index  # a tuple

min_info = argmin([1, 2, 0])
min_info[0]  # 0
min_info[1]  # 2
# often unpacking is used directly
m, i = argmin([1, 2, 0])  # m is 0, i is 2
```
A final word on tuples

▶ Tuples are *nothing else* than immutable lists
▶ In most cases lists may be used instead of tuples
▶ The parentheses free notation is nice but *dangerous*, you should *use parentheses when you are not sure*:

```python
a, b = b, a  # the swap trick, equivalent to:
(a, b) = (b, a)
# but
1, 2 == 3, 4  # returns (1, False, 4)
(1, 2) == (3, 4)  # returns False
```
## Conversions

Containers can be converted to other container types:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>Tuple</td>
<td><code>tuple([1, 2, 3])</code></td>
</tr>
<tr>
<td>Tuple</td>
<td>List</td>
<td><code>list((1, 2, 3))</code></td>
</tr>
<tr>
<td>List, Tuple</td>
<td>Set</td>
<td><code>set([1,2,3]), set((1,2,3))</code></td>
</tr>
<tr>
<td>Set</td>
<td>List</td>
<td><code>list({1,2,3})</code></td>
</tr>
<tr>
<td>Dictionary</td>
<td>List</td>
<td><code>{’a’:4}.values()</code></td>
</tr>
<tr>
<td>List</td>
<td>Dictionary</td>
<td>– not possible</td>
</tr>
</tbody>
</table>
## Summary and Overview

<table>
<thead>
<tr>
<th>Type</th>
<th>Access</th>
<th>Order</th>
<th>Duplicate Values</th>
<th>Mutability</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>index</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Tuple</td>
<td>index</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Dictionary</td>
<td>key</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Set</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>