Lists, tuples, and dictionaries in Python (3.x)

constructing, indexing, slicing, and modifying

Piotr Beling

Uniwersytet Łódzki
(University of Łódź)

2016

Run ipython qtconsole

Today we will work in a python shell.
Please run *ipython qtconsole*:

- Windows with Anaconda: Start → (All) Applications → Anaconda3 → Jupyter QTConsole
- Linux: `ipython3 qtconsole`

Lists and tuples – basics

- `list` and `tuple` are built-in types;
- both represent ordered sequences of objects;
- both can hold any type and even mixed types of objects;
- `list`s are enclosed in square brackets, for example
  `l = [1, 2, "abc", 3]`
  and `tuple`s in (usually optional) round brackets, for example
  `t = (1, 2, "efg", 3)`
  `t = 1, 2, "efg", 3`
- both are indexed by integers and the first index is 0, e.g.
  `l[0]` is `1`, `t[2]` is `"efg"`;
- negative indices access elements from the end (`-1` refers to the last item, `-2` is the second-last, and so on), e.g.
  `l[-2]` is `"abc"` and `t[-1]` is `3`;
- `list`s can change after being created, `tuple`s can not, e.g.
  `l[0] = "hello!"` changes the first value in the `list` `l`,
  `t[0] = "hello!"` raises `TypeError` exception.

Lists and tuples – basics

- `list` and `tuple` are built-in types;
- both represent ordered sequences of objects;
- both can hold any type and even mixed types of objects;
- `list`s are enclosed in square brackets, for example
  `l = [1, 2, "abc", 3]`
  and `tuple`s in (usually optional) round brackets, for example
  `t = (1, 2, "efg", 3)`
  `t = 1, 2, "efg", 3`
- both are indexed by integers and the first index is 0, e.g.
  `l[0]` is `1`, `t[2]` is `"efg"`;
- negative indices access elements from the end (`-1` refers to the last item, `-2` is the second-last, and so on), e.g.
  `l[-2]` is `"abc"` and `t[-1]` is `3`;
- `list`s can change after being created, `tuple`s can not, e.g.
  `l[0] = "hello!"` changes the first value in the `list` `l`,
  `t[0] = "hello!"` raises `TypeError` exception.

Lists and tuples – basics

- `list` and `tuple` are built-in types;
- both represent ordered sequences of objects;
- both can hold any type and even mixed types of objects;
- `list`s are enclosed in square brackets, for example
  `l = [1, 2, "abc", 3]`
  and `tuple`s in (usually optional) round brackets, for example
  `t = (1, 2, "efg", 3)`
  `t = 1, 2, "efg", 3`
- both are indexed by integers and the first index is 0, e.g.
  `l[0]` is `1`, `t[2]` is `"efg"`;
- negative indices access elements from the end (`-1` refers to the last item, `-2` is the second-last, and so on), e.g.
  `l[-2]` is `"abc"` and `t[-1]` is `3`;
- `list`s can change after being created, `tuple`s can not, e.g.
  `l[0] = "hello!"` changes the first value in the `list` `l`,
  `t[0] = "hello!"` raises `TypeError` exception.

Lists and tuples – basics

- **list** and **tuple** are built-in types;
- both represent ordered sequences of objects;
- both can hold any type and even mixed types of objects;
- **list**s are enclosed in square brackets, for example
  `l = [1, 2, "abc", 3]`
  and **tuple**s in (usually optional) round brackets, for example
  `t = (1, 2, "efg", 3)`
  `t = 1, 2, "efg", 3`
- both are indexed by integers and the first index is 0, e.g.
  `l[0]` is `1`, `t[2]` is `"efg"`;
- negative indices access elements from the end (`-1` refers to the last item, `-2` is the second-last, and so on), e.g.
  `l[-2]` is `"abc"` and `t[-1]` is `3`;
- **list**s can change after being created, **tuple**s can not, e.g.
  `l[0] = "hello!"` changes the first value in the **list** `l`,
  `t[0] = "hello!"` raises `TypeError` exception.

Lists and tuples – basics

- **list** and **tuple** are built-in types;
- both represent ordered sequences of objects;
- both can hold any type and even mixed types of objects;
- **list**s are enclosed in square brackets, for example
  `l = [1, 2, "abc", 3]`
  and **tuple**s in (usually optional) round brackets, for example
  `t = (1, 2, "efg", 3)`
  `t = 1, 2, "efg", 3`
- both are indexed by integers and the first index is 0, e.g.
  `l[0]` is `1`, `t[2]` is `"efg"`;
- negative indices access elements from the end (`-1` refers to the last item, `-2` is the second-last, and so on), e.g.
  `l[-2]` is `"abc"` and `t[-1]` is `3`;
- **list**s can change after being created, **tuple**s can not, e.g.
  `l[0] = "hello!"` changes the first value in the **list** `l`,
  `t[0] = "hello!"` raises `TypeError` exception.

Lists and tuples – basics

- **list** and **tuple** are built-in types;
- both represent ordered sequences of objects;
- both can hold any type and even mixed types of objects;
- **list**s are enclosed in square brackets, for example
  `l = [1, 2, "abc", 3]`
  and **tuple**s in (usually optional) round brackets, for example
  `t = (1, 2, "efg", 3)`
  `t = 1, 2, "efg", 3`
- both are indexed by integers and the first index is 0, e.g.
  `l[0]` is `1`, `t[2]` is `"efg"`;
- negative indices access elements from the end (`-1` refers to the last item, `-2` is the second-last, and so on), e.g.
  `l[-2]` is `"abc"` and `t[-1]` is `3`;
- **list**s can change after being created, **tuple**s can not, e.g.
  `l[0] = "hello!"` changes the first value in the **list** `l`,
  `t[0] = "hello!"` raises `TypeError` exception.

Lists and tuples – basics

- **list** and **tuple** are built-in types;
- both represent ordered sequences of objects;
- both can hold any type and even mixed types of objects;
- **list**s are enclosed in square brackets, for example
  `l = [1, 2, "abc", 3]`
  and **tuple**s in (usually optional) round brackets, for example
  `t = (1, 2, "efg", 3)`
  `t = 1, 2, "efg", 3`
- both are indexed by integers and the first index is 0, e.g.
  `l[0]` is `1`, `t[2]` is `"efg"`;
- negative indices access elements from the end (`-1` refers to the last item, `-2` is the second-last, and so on), e.g.
  `l[-2]` is `"abc"` and `t[-1]` is `3`;
- **list**s can change after being created, **tuple**s can not, e.g.
  `l[0] = "hello!"` changes the first value in the **list** `l`,
  `t[0] = "hello!"` raises `TypeError` exception.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **`list`** of 2 **`int`**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **`tuple`**);
`x = ('a')`	`x` is just a **`str`**ing, not a **`tuple`**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **`tuple`**;
`tuple``(a)`	creates a **`tuple`** initialized from items of `a`;
`list``(t)`	creates a **`list`** initialized from items of `t`;
`c,d = a`	uses **`tuple`** to unpack **`list`**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**`list`** can also be used for unpacking;
`s, = t`	unpacks **`tuple`**; same as `s = t[0]`;
`c,d = 4,5`	uses **`tuple`**s to assign: `c = 4; d = 5`;
`c,d = d,c`	uses **`tuple`**s to swap the values of `c` and `d`;
`z = [0]*10`	**`list`** of 10 zeros; `*` repeats and concatenates;

Exercise: use **`list`**s (on both sides of `=`) to exchange the values of `c` and `d`.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **`list`** of 2 **`int`**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **`tuple`**);
`x = ('a')`	`x` is just a **`str`**ing, not a **`tuple`**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **`tuple`**;
`tuple``(a)`	creates a **`tuple`** initialized from items of `a`;
`list``(t)`	creates a **`list`** initialized from items of `t`;
`c,d = a`	uses **`tuple`** to unpack **`list`**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**`list`** can also be used for unpacking;
`s, = t`	unpacks **`tuple`**; same as `s = t[0]`;
`c,d = 4,5`	uses **`tuple`**s to assign: `c = 4; d = 5`;
`c,d = d,c`	uses **`tuple`**s to swap the values of `c` and `d`;
`z = [0]*10`	**`list`** of 10 zeros; `*` repeats and concatenates;

Exercise: use **`list`**s (on both sides of `=`) to exchange the values of `c` and `d`.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **`list`** of 2 **`int`**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **`tuple`**);
`x = ('a')`	`x` is just a **`str`**ing, not a **`tuple`**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **`tuple`**;
`tuple``(a)`	creates a **`tuple`** initialized from items of `a`;
`list``(t)`	creates a **`list`** initialized from items of `t`;
`c,d = a`	uses **`tuple`** to unpack **`list`**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**`list`** can also be used for unpacking;
`s, = t`	unpacks **`tuple`**; same as `s = t[0]`;
`c,d = 4,5`	uses **`tuple`**s to assign: `c = 4; d = 5`;
`c,d = d,c`	uses **`tuple`**s to swap the values of `c` and `d`;
`z = [0]*10`	**`list`** of 10 zeros; `*` repeats and concatenates;

Exercise: use **`list`**s (on both sides of `=`) to exchange the values of `c` and `d`.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **`list`** of 2 **`int`**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **`tuple`**);
`x = ('a')`	`x` is just a **`str`**ing, not a **`tuple`**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **`tuple`**;
`tuple``(a)`	creates a **`tuple`** initialized from items of `a`;
`list``(t)`	creates a **`list`** initialized from items of `t`;
`c,d = a`	uses **`tuple`** to unpack **`list`**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**`list`** can also be used for unpacking;
`s, = t`	unpacks **`tuple`**; same as `s = t[0]`;
`c,d = 4,5`	uses **`tuple`**s to assign: `c = 4; d = 5`;
`c,d = d,c`	uses **`tuple`**s to swap the values of `c` and `d`;
`z = [0]*10`	**`list`** of 10 zeros; `*` repeats and concatenates;

Exercise: use **`list`**s (on both sides of `=`) to exchange the values of `c` and `d`.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **`list`** of 2 **`int`**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **`tuple`**);
`x = ('a')`	`x` is just a **`str`**ing, not a **`tuple`**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **`tuple`**;
`tuple``(a)`	creates a **`tuple`** initialized from items of `a`;
`list``(t)`	creates a **`list`** initialized from items of `t`;
`c,d = a`	uses **`tuple`** to unpack **`list`**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**`list`** can also be used for unpacking;
`s, = t`	unpacks **`tuple`**; same as `s = t[0]`;
`c,d = 4,5`	uses **`tuple`**s to assign: `c = 4; d = 5`;
`c,d = d,c`	uses **`tuple`**s to swap the values of `c` and `d`;
`z = [0]*10`	**`list`** of 10 zeros; `*` repeats and concatenates;

Exercise: use **`list`**s (on both sides of `=`) to exchange the values of `c` and `d`.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **`list`** of 2 **`int`**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **`tuple`**);
`x = ('a')`	`x` is just a **`str`**ing, not a **`tuple`**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **`tuple`**;
`tuple``(a)`	creates a **`tuple`** initialized from items of `a`;
`list``(t)`	creates a **`list`** initialized from items of `t`;
`c,d = a`	uses **`tuple`** to unpack **`list`**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**`list`** can also be used for unpacking;
`s, = t`	unpacks **`tuple`**; same as `s = t[0]`;
`c,d = 4,5`	uses **`tuple`**s to assign: `c = 4; d = 5`;
`c,d = d,c`	uses **`tuple`**s to swap the values of `c` and `d`;
`z = [0]*10`	**`list`** of 10 zeros; `*` repeats and concatenates;

Exercise: use **`list`**s (on both sides of `=`) to exchange the values of `c` and `d`.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **`list`** of 2 **`int`**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **`tuple`**);
`x = ('a')`	`x` is just a **`str`**ing, not a **`tuple`**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **`tuple`**;
`tuple``(a)`	creates a **`tuple`** initialized from items of `a`;
`list``(t)`	creates a **`list`** initialized from items of `t`;
`c,d = a`	uses **`tuple`** to unpack **`list`**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**`list`** can also be used for unpacking;
`s, = t`	unpacks **`tuple`**; same as `s = t[0]`;
`c,d = 4,5`	uses **`tuple`**s to assign: `c = 4; d = 5`;
`c,d = d,c`	uses **`tuple`**s to swap the values of `c` and `d`;
`z = [0]*10`	**`list`** of 10 zeros; `*` repeats and concatenates;

Exercise: use **`list`**s (on both sides of `=`) to exchange the values of `c` and `d`.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **`list`** of 2 **`int`**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **`tuple`**);
`x = ('a')`	`x` is just a **`str`**ing, not a **`tuple`**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **`tuple`**;
`tuple``(a)`	creates a **`tuple`** initialized from items of `a`;
`list``(t)`	creates a **`list`** initialized from items of `t`;
`c,d = a`	uses **`tuple`** to unpack **`list`**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**`list`** can also be used for unpacking;
`s, = t`	unpacks **`tuple`**; same as `s = t[0]`;
`c,d = 4,5`	uses **`tuple`**s to assign: `c = 4; d = 5`;
`c,d = d,c`	uses **`tuple`**s to swap the values of `c` and `d`;
`z = [0]*10`	**`list`** of 10 zeros; `*` repeats and concatenates;

Exercise: use **`list`**s (on both sides of `=`) to exchange the values of `c` and `d`.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **`list`** of 2 **`int`**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **`tuple`**);
`x = ('a')`	`x` is just a **`str`**ing, not a **`tuple`**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **`tuple`**;
`tuple``(a)`	creates a **`tuple`** initialized from items of `a`;
`list``(t)`	creates a **`list`** initialized from items of `t`;
`c,d = a`	uses **`tuple`** to unpack **`list`**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**`list`** can also be used for unpacking;
`s, = t`	unpacks **`tuple`**; same as `s = t[0]`;
`c,d = 4,5`	uses **`tuple`**s to assign: `c = 4; d = 5`;
`c,d = d,c`	uses **`tuple`**s to swap the values of `c` and `d`;
`z = [0]*10`	**`list`** of 10 zeros; `*` repeats and concatenates;

Exercise: use **`list`**s (on both sides of `=`) to exchange the values of `c` and `d`.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **`list`** of 2 **`int`**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **`tuple`**);
`x = ('a')`	`x` is just a **`str`**ing, not a **`tuple`**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **`tuple`**;
`tuple``(a)`	creates a **`tuple`** initialized from items of `a`;
`list``(t)`	creates a **`list`** initialized from items of `t`;
`c,d = a`	uses **`tuple`** to unpack **`list`**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**`list`** can also be used for unpacking;
`s, = t`	unpacks **`tuple`**; same as `s = t[0]`;
`c,d = 4,5`	uses **`tuple`**s to assign: `c = 4; d = 5;`
`c,d = d,c`	uses **`tuple`**s to swap the values of `c` and `d`;
`z = [0]*10`	**`list`** of 10 zeros; `*` repeats and concatenates;

Exercise: use **`list`**s (on both sides of `=`) to exchange the values of `c` and `d`.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **`list`** of 2 **`int`**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **`tuple`**);
`x = ('a')`	`x` is just a **`str`**ing, not a **`tuple`**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **`tuple`**;
`tuple``(a)`	creates a **`tuple`** initialized from items of `a`;
`list``(t)`	creates a **`list`** initialized from items of `t`;
`c,d = a`	uses **`tuple`** to unpack **`list`**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**`list`** can also be used for unpacking;
`s, = t`	unpacks **`tuple`**; same as `s = t[0]`;
`c,d = 4,5`	uses **`tuple`**s to assign: `c = 4; d = 5;`
`c,d = d,c`	uses **`tuple`**s to swap the values of `c` and `d`;
`z = [0]*10`	**`list`** of 10 zeros; `*` repeats and concatenates;

Exercise: use **`list`**s (on both sides of `=`) to exchange the values of `c` and `d`.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **`list`** of 2 **`int`**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **`tuple`**);
`x = ('a')`	`x` is just a **`str`**ing, not a **`tuple`**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **`tuple`**;
`tuple``(a)`	creates a **`tuple`** initialized from items of `a`;
`list``(t)`	creates a **`list`** initialized from items of `t`;
`c,d = a`	uses **`tuple`** to unpack **`list`**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**`list`** can also be used for unpacking;
`s, = t`	unpacks **`tuple`**; same as `s = t[0]`;
`c,d = 4,5`	uses **`tuple`**s to assign: `c = 4; d = 5;`
`c,d = d,c`	uses **`tuple`**s to swap the values of `c` and `d`;
`z = [0]*10`	**`list`** of 10 zeros; `*` repeats and concatenates;

Exercise: use **`list`**s (on both sides of `=`) to exchange the values of `c` and `d`.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **list** of 2 **int**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **tuple**);
`x = ('a')`	`x` is just a **str**ing, not a **tuple**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **tuple**;
tuple`(a)`	creates a **tuple** initialized from items of `a`;
list`(t)`	creates a **list** initialized from items of `t`;
`c,d = a`	uses **tuple** to unpack **list**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**list** can also be used for unpacking;
`s, = t`	unpacks **tuple**; same as `s = t[0]`;
`c,d = 4,5`	uses **tuple**s to assign: `c = 4; d = 5`;
`c,d = d,c`	uses **tuple**s to swap the values of `c` and `d`;
`z = [0]*10`	**list** of 10 zeros; `*` repeats and concatenates;

Exercise: use **list**s (on both sides of `=`) to exchange the values of `c` and `d`.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **`list`** of 2 **`int`**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **`tuple`**);
`x = ('a')`	`x` is just a **`str`**ing, not a **`tuple`**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **`tuple`**;
`tuple``(a)`	creates a **`tuple`** initialized from items of `a`;
`list``(t)`	creates a **`list`** initialized from items of `t`;
`c,d = a`	uses **`tuple`** to unpack **`list`**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**`list`** can also be used for unpacking;
`s, = t`	unpacks **`tuple`**; same as `s = t[0]`;
`c,d = 4,5`	uses **`tuple`**s to assign: `c = 4; d = 5`;
`c,d = d,c`	uses **`tuple`**s to swap the values of `c` and `d`;
`z = [0]*10`	**`list`** of 10 zeros; `*` repeats and concatenates;

Exercise: use **`list`**s (on both sides of `=`) to exchange the values of `c` and `d`.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **list** of 2 **int**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **tuple**);
`x = ('a')`	`x` is just a **str**ing, not a **tuple**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **tuple**;
tuple`(a)`	creates a **tuple** initialized from items of `a`;
list`(t)`	creates a **list** initialized from items of `t`;
`c,d = a`	uses **tuple** to unpack **list**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**list** can also be used for unpacking;
`s, = t`	unpacks **tuple**; same as `s = t[0]`;
`c,d = 4,5`	uses **tuple**s to assign: `c = 4; d = 5`;
`c,d = d,c`	uses **tuple**s to swap the values of `c` and `d`;
`z = [0]*10`	**list** of 10 zeros; `*` repeats and concatenates;

Exercise: use **lists** (on both sides of =) to exchange the values of `c` and `d`.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **`list`** of 2 **`int`**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **`tuple`**);
`x = ('a')`	`x` is just a **`str`**ing, not a **`tuple`**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **`tuple`**;
`tuple``(a)`	creates a **`tuple`** initialized from items of `a`;
`list``(t)`	creates a **`list`** initialized from items of `t`;
`c,d = a`	uses **`tuple`** to unpack **`list`**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**`list`** can also be used for unpacking;
`s, = t`	unpacks **`tuple`**; same as `s = t[0]`;
`c,d = 4,5`	uses **`tuple`**s to assign: `c = 4; d = 5`;
`c,d = d,c`	uses **`tuple`**s to swap the values of `c` and `d`;
`z = [0]*10`	**`list`** of 10 zeros; `*` repeats and concatenates;

Exercise: use **`list`**s (on both sides of `=`) to exchange the values of `c` and `d`.

Lists and tuples – constructing and unpacking

`a = [1,2]`	construct a **list** of 2 **int**s;
`a = [1,2,]`	same as above; an extra trailing comma is allowed,
`t = ('a',)`	or even necessary (for a one-element **tuple**);
`x = ('a')`	`x` is just a **str**ing, not a **tuple**;
`t = 'a',`	same as `t = ('a',)`; brackets are often optional,
`()`	but sometimes needed, e.g. for the empty **tuple**;
tuple`(a)`	creates a **tuple** initialized from items of `a`;
list`(t)`	creates a **list** initialized from items of `t`;
`c,d = a`	uses **tuple** to unpack **list**: `c=a[0]`, `d=a[1]`;
`(c,d) = a`	same as above, but with explicit brackets;
`[c,d] = a`	**list** can also be used for unpacking;
`s, = t`	unpacks **tuple**; same as `s = t[0]`;
`c,d = 4,5`	uses **tuple**s to assign: `c = 4; d = 5`;
`c,d = d,c`	uses **tuple**s to swap the values of `c` and `d`;
`z = [0]*10`	**list** of 10 zeros; `*` repeats and concatenates;

Exercise: use **list**s (on both sides of =) to exchange the values of `c` and `d`. Code: `[c, d] = [d, c]`

Lists and tuples – examples and exercises

Execute:

`a = [1, 5, 5]`	construct a **list** of 3 **int**egers;
`b = [6, 7, 'b']`	a list with mixed item types;
`c = [[3,5], 4]`	a list with another list inside of it;
`len(c)`	`c` has 2 items: the list `[3,5]` and `4`;
`a + b`	concatenates `a` and `b`;
`b * 3`	concatenates 3 repetitions of `b`;
`5 in a`	tests if `5` is in the list `a`;
`a.count(5)`	the number of occurrences of `5` in `a`;
`a.index(5)`	an index of the first occurrence of `5` in `a`;
`3 in c`	`3` is not a member of `c`;
`3 in c[0]`	`3` is a member of the first item of `c`;

All the operations above are also applicable to **tuple**s.

Exercise: construct the tuple `t=(1, ("ef",2), [3,9])` and:

1. find the length of `t`
2. concatenate 5 repetitions of `t`
3. find an index of the **str**ing `"ef"` in `t`

Lists and tuples – examples and exercises

Execute:

`a = [1, 5, 5]`	construct a **list** of 3 **int**egers;
`b = [6, 7, 'b']`	a list with mixed item types;
`c = [[3,5], 4]`	a list with another list inside of it;
`len(c)`	`c` has 2 items: the list `[3,5]` and `4`;
`a + b`	concatenates `a` and `b`;
`b * 3`	concatenates 3 repetitions of `b`;
`5 in a`	tests if `5` is in the list `a`;
`a.count(5)`	the number of occurrences of `5` in `a`;
`a.index(5)`	an index of the first occurrence of `5` in `a`;
`3 in c`	`3` is not a member of `c`;
`3 in c[0]`	`3` is a member of the first item of `c`;

All the operations above are also applicable to **tuple**s.

Exercise: construct the tuple `t=(1, ("ef",2), [3,9])` and:

1. find the length of `t`
2. concatenate 5 repetitions of `t`
3. find an index of the **str**ing `"ef"` in `t`

Lists and tuples – examples and exercises

Execute:

`a = [1, 5, 5]`	construct a **list** of 3 **int**egers;
`b = [6, 7, 'b']`	a list with mixed item types;
`c = [[3,5], 4]`	a list with another list inside of it;
`len(c)`	`c` has 2 items: the list `[3,5]` and `4`;
`a + b`	concatenates `a` and `b`;
`b * 3`	concatenates 3 repetitions of `b`;
`5 in a`	tests if `5` is in the list `a`;
`a.count(5)`	the number of occurrences of `5` in `a`;
`a.index(5)`	an index of the first occurrence of `5` in `a`;
`3 in c`	`3` is not a member of `c`;
`3 in c[0]`	`3` is a member of the first item of `c`;

All the operations above are also applicable to **tuple**s.

Exercise: construct the tuple `t=(1, ("ef",2), [3,9])` and:

1. find the length of `t`
2. concatenate 5 repetitions of `t`
3. find an index of the **str**ing `"ef"` in `t`

Lists and tuples – examples and exercises

Execute:

`a = [1, 5, 5]`	construct a **list** of 3 **int**egers;
`b = [6, 7, 'b']`	a list with mixed item types;
`c = [[3,5], 4]`	a list with another list inside of it;
`len(c)`	`c` has 2 items: the list `[3,5]` and `4`;
`a + b`	concatenates `a` and `b`;
`b * 3`	concatenates 3 repetitions of `b`;
`5 in a`	tests if `5` is in the list `a`;
`a.count(5)`	the number of occurrences of `5` in `a`;
`a.index(5)`	an index of the first occurrence of `5` in `a`;
`3 in c`	`3` is not a member of `c`;
`3 in c[0]`	`3` is a member of the first item of `c`;

All the operations above are also applicable to **tuple**s.

Exercise: construct the tuple `t=(1, ("ef",2), [3,9])` and:

1. find the length of `t`
2. concatenate 5 repetitions of `t`
3. find an index of the **str**ing `"ef"` in `t`

Lists and tuples – examples and exercises

Execute:

`a = [1, 5, 5]`	construct a **list** of 3 **int**egers;
`b = [6, 7, 'b']`	a list with mixed item types;
`c = [[3,5], 4]`	a list with another list inside of it;
`len(c)`	`c` has 2 items: the list `[3,5]` and `4`;
`a + b`	concatenates `a` and `b`;
`b * 3`	concatenates 3 repetitions of `b`;
`5 in a`	tests if `5` is in the list `a`;
`a.count(5)`	the number of occurrences of `5` in `a`;
`a.index(5)`	an index of the first occurrence of `5` in `a`;
`3 in c`	`3` is not a member of `c`;
`3 in c[0]`	`3` is a member of the first item of `c`;

All the operations above are also applicable to **tuple**s.

Exercise: construct the tuple `t=(1, ("ef",2), [3,9])` and:

1. find the length of `t`
2. concatenate 5 repetitions of `t`
3. find an index of the **str**ing `"ef"` in `t`

Lists and tuples – examples and exercises

Execute:

`a = [1, 5, 5]`	construct a **`list`** of 3 **`int`**egers;
`b = [6, 7, 'b']`	a list with mixed item types;
`c = [[3,5], 4]`	a list with another list inside of it;
`len``(c)`	`c` has 2 items: the list `[3,5]` and `4`;
`a + b`	concatenates `a` and `b`;
`b * 3`	concatenates 3 repetitions of `b`;
`5 in a`	tests if `5` is in the list `a`;
`a.count(5)`	the number of occurrences of `5` in `a`;
`a.index(5)`	an index of the first occurrence of `5` in `a`;
`3 in c`	`3` is not a member of `c`;
`3 in c[0]`	`3` is a member of the first item of `c`;

All the operations above are also applicable to **`tuple`**s.

Exercise: construct the tuple `t=(1, ("ef",2), [3,9])` and:

1. find the length of `t`
2. concatenate 5 repetitions of `t`
3. find an index of the **`str`**ing `"ef"` in `t`

Lists and tuples – examples and exercises

Execute:

`a = [1, 5, 5]`	construct a **list** of 3 **int**egers;
`b = [6, 7, 'b']`	a list with mixed item types;
`c = [[3,5], 4]`	a list with another list inside of it;
`len(c)`	`c` has 2 items: the list `[3,5]` and `4`;
`a + b`	concatenates `a` and `b`;
`b * 3`	concatenates 3 repetitions of `b`;
`5 in a`	tests if `5` is in the list `a`;
`a.count(5)`	the number of occurrences of `5` in `a`;
`a.index(5)`	an index of the first occurrence of `5` in `a`;
`3 in c`	`3` is not a member of `c`;
`3 in c[0]`	`3` is a member of the first item of `c`;

All the operations above are also applicable to **tuple**s.

Exercise: construct the tuple `t=(1, ("ef",2), [3,9])` and:

1. find the length of `t`
2. concatenate 5 repetitions of `t`
3. find an index of the **str**ing `"ef"` in `t`

Lists and tuples – examples and exercises

Execute:

`a = [1, 5, 5]`	construct a **list** of 3 **int**egers;
`b = [6, 7, 'b']`	a list with mixed item types;
`c = [[3,5], 4]`	a list with another list inside of it;
`len(c)`	`c` has 2 items: the list `[3,5]` and `4`;
`a + b`	concatenates `a` and `b`;
`b * 3`	concatenates 3 repetitions of `b`;
`5 in a`	tests if `5` is in the list `a`;
`a.count(5)`	the number of occurrences of `5` in `a`;
`a.index(5)`	an index of the first occurrence of `5` in `a`;
`3 in c`	`3` is not a member of `c`;
`3 in c[0]`	`3` is a member of the first item of `c`;

All the operations above are also applicable to **tuple**s.

Exercise: construct the tuple `t=(1, ("ef",2), [3,9])` and:

1. find the length of `t`
2. concatenate 5 repetitions of `t`
3. find an index of the **str**ing `"ef"` in `t`

Lists and tuples – examples and exercises

Execute:

`a = [1, 5, 5]`	construct a **list** of 3 **int**egers;
`b = [6, 7, 'b']`	a list with mixed item types;
`c = [[3,5], 4]`	a list with another list inside of it;
`len(c)`	`c` has 2 items: the list `[3,5]` and `4`;
`a + b`	concatenates `a` and `b`;
`b * 3`	concatenates 3 repetitions of `b`;
`5 in a`	tests if `5` is in the list `a`;
`a.count(5)`	the number of occurrences of `5` in `a`;
`a.index(5)`	an index of the first occurrence of `5` in `a`;
`3 in c`	`3` is not a member of `c`;
`3 in c[0]`	`3` is a member of the first item of `c`;

All the operations above are also applicable to **tuple**s.

Exercise: construct the tuple `t=(1, ("ef",2), [3,9])` and:

1. find the length of `t`
2. concatenate 5 repetitions of `t`
3. find an index of the **str**ing `"ef"` in `t`

Lists and tuples – examples and exercises

Execute:

`a = [1, 5, 5]`	construct a **list** of 3 **int**egers;
`b = [6, 7, 'b']`	a list with mixed item types;
`c = [[3,5], 4]`	a list with another list inside of it;
`len(c)`	`c` has 2 items: the list `[3,5]` and `4`;
`a + b`	concatenates `a` and `b`;
`b * 3`	concatenates 3 repetitions of `b`;
`5 in a`	tests if `5` is in the list `a`;
`a.count(5)`	the number of occurrences of `5` in `a`;
`a.index(5)`	an index of the first occurrence of `5` in `a`;
`3 in c`	`3` is not a member of `c`;
`3 in c[0]`	`3` is a member of the first item of `c`;

All the operations above are also applicable to **tuple**s.

Exercise: construct the tuple `t=(1, ("ef",2), [3,9])` and:

1. find the length of `t`
2. concatenate 5 repetitions of `t`
3. find an index of the **str**ing `"ef"` in `t`

Lists and tuples – examples and exercises

Execute:

`a = [1, 5, 5]`	construct a **list** of 3 **int**egers;
`b = [6, 7, 'b']`	a list with mixed item types;
`c = [[3,5], 4]`	a list with another list inside of it;
`len(c)`	`c` has 2 items: the list `[3,5]` and `4`;
`a + b`	concatenates `a` and `b`;
`b * 3`	concatenates 3 repetitions of `b`;
`5 in a`	tests if `5` is in the list `a`;
`a.count(5)`	the number of occurrences of `5` in `a`;
`a.index(5)`	an index of the first occurrence of `5` in `a`;
`3 in c`	`3` is not a member of `c`;
`3 in c[0]`	`3` is a member of the first item of `c`;

All the operations above are also applicable to **tuple**s.

Exercise: construct the tuple `t=(1, ("ef",2), [3,9])` and:

1. find the length of `t`
2. concatenate 5 repetitions of `t`
3. find an index of the **str**ing `"ef"` in `t`

Lists and tuples – examples and exercises

Execute:

`a = [1, 5, 5]`	construct a **list** of 3 **int**egers;
`b = [6, 7, 'b']`	a list with mixed item types;
`c = [[3,5], 4]`	a list with another list inside of it;
`len(c)`	`c` has 2 items: the list `[3,5]` and `4`;
`a + b`	concatenates `a` and `b`;
`b * 3`	concatenates 3 repetitions of `b`;
`5 in a`	tests if `5` is in the list `a`;
`a.count(5)`	the number of occurrences of `5` in `a`;
`a.index(5)`	an index of the first occurrence of `5` in `a`;
`3 in c`	`3` is not a member of `c`;
`3 in c[0]`	`3` is a member of the first item of `c`;

All the operations above are also applicable to **tuple**s.

Exercise: construct the tuple `t=(1, ("ef",2), [3,9])` and:

1. find the length of `t`
2. concatenate 5 repetitions of `t`
3. find an index of the **str**ing `"ef"` in `t`

Lists and tuples – examples and exercises

Execute:

`a = [1, 5, 5]`	construct a **list** of 3 **int**egers;
`b = [6, 7, 'b']`	a list with mixed item types;
`c = [[3,5], 4]`	a list with another list inside of it;
`len(c)`	`c` has 2 items: the list `[3,5]` and `4`;
`a + b`	concatenates `a` and `b`;
`b * 3`	concatenates 3 repetitions of `b`;
`5 in a`	tests if `5` is in the list `a`;
`a.count(5)`	the number of occurrences of `5` in `a`;
`a.index(5)`	an index of the first occurrence of `5` in `a`;
`3 in c`	`3` is not a member of `c`;
`3 in c[0]`	`3` is a member of the first item of `c`;

All the operations above are also applicable to **tuple**s.

Exercise: construct the tuple `t=(1, ("ef",2), [3,9])` and:

1. find the length of `t`
2. concatenate 5 repetitions of `t`
3. find an index of the **str**ing `"ef"` in `t`

Lists and tuples – examples and exercises

Execute:

`a = [1, 5, 5]`	construct a **list** of 3 **int**egers;
`b = [6, 7, 'b']`	a list with mixed item types;
`c = [[3,5], 4]`	a list with another list inside of it;
`len(c)`	`c` has 2 items: the list `[3,5]` and `4`;
`a + b`	concatenates `a` and `b`;
`b * 3`	concatenates 3 repetitions of `b`;
`5 in a`	tests if `5` is in the list `a`;
`a.count(5)`	the number of occurrences of `5` in `a`;
`a.index(5)`	an index of the first occurrence of `5` in `a`;
`3 in c`	`3` is not a member of `c`;
`3 in c[0]`	`3` is a member of the first item of `c`;

All the operations above are also applicable to **tuple**s.

Exercise: construct the tuple `t=(1, ("ef",2), [3,9])` and:

1. find the length of `t` Code: `len(t)`
2. concatenate 5 repetitions of `t` Code: `t * 5`
3. find an index of the **str**ing `"ef"` in `t` Code: `t.index("ef")`

Lists and tuples – indexing [1/2]

Execute:

```
a = ['a', 'b', 'c', 'd', 'e', 'f']
```

`len(a)`	the length of `a` is 6;
`a[0]`	`0` refers to the first item,
`a[1]`	`1` refers to the second item, and so on;
`a[6]`	raises `IndexError` since 5=`len(a)-1` is the highest valid index;
`a[-1]`	`-1` refers to the last item,
`a[-2]`	`-2` is the second-last, and so on;
`a[-7]`	raises `IndexError` since -6=`-len(a)` is the lowest (negative) valid index;

(The same indexing scheme is used also by `tuple`s, `str`ings, ...)

list	[	'a',	'b',	'c',	'd',	'e',	'f'	]
indices		0	1	2	3	4	5	
from the end		-6	-5	-4	-3	-2	-1	

Lists and tuples – indexing [1/2]

Execute:

```
a = ['a', 'b', 'c', 'd', 'e', 'f']
```

`len(a)`	the length of `a` is 6;
`a[0]`	`0` refers to the first item,
`a[1]`	`1` refers to the second item, and so on;
`a[6]`	raises `IndexError` since 5=`len(a)-1` is the highest valid index;
`a[-1]`	`-1` refers to the last item,
`a[-2]`	`-2` is the second-last, and so on;
`a[-7]`	raises `IndexError` since -6=`-len(a)` is the lowest (negative) valid index;

(The same indexing scheme is used also by `tuple`s, `str`ings, ...)

list	[	'a',	'b',	'c',	'd',	'e',	'f']
indices		0	1	2	3	4	5
from the end		-6	-5	-4	-3	-2	-1

Lists and tuples – indexing [1/2]

Execute:

```
a = ['a', 'b', 'c', 'd', 'e', 'f']
```

`len(a)`	the length of `a` is 6;
`a[0]`	`0` refers to the first item,
`a[1]`	`1` refers to the second item, and so on;
`a[6]`	raises `IndexError` since 5=`len(a)-1` is the highest valid index;
`a[-1]`	`-1` refers to the last item,
`a[-2]`	`-2` is the second-last, and so on;
`a[-7]`	raises `IndexError` since -6=`-len(a)` is the lowest (negative) valid index;

(The same indexing scheme is used also by `tuple`s, `str`ings, ...)

list	[	'a',	'b',	'c',	'd',	'e',	'f'	]
indices		0	1	2	3	4	5	
from the end		-6	-5	-4	-3	-2	-1	

Lists and tuples – indexing [1/2]

Execute:

```
a = ['a', 'b', 'c', 'd', 'e', 'f']
```

`len(a)`	the length of `a` is 6;
`a[0]`	`0` refers to the first item,
`a[1]`	`1` refers to the second item, and so on;
`a[6]`	raises `IndexError` since 5=`len(a)-1` is the highest valid index;
`a[-1]`	`-1` refers to the last item,
`a[-2]`	`-2` is the second-last, and so on;
`a[-7]`	raises `IndexError` since -6=`-len(a)` is the lowest (negative) valid index;

(The same indexing scheme is used also by `tuple`s, `str`ings, ...)

list	[	'a',	'b',	'c',	'd',	'e',	'f'	]
indices		0	1	2	3	4	5	
from the end		-6	-5	-4	-3	-2	-1	

Lists and tuples – indexing [1/2]

Execute:

```
a = ['a', 'b', 'c', 'd', 'e', 'f']
```

`len(a)`	the length of `a` is 6;
`a[0]`	`0` refers to the first item,
`a[1]`	`1` refers to the second item, and so on;
`a[6]`	raises `IndexError` since 5=`len(a)-1` is the highest valid index;
`a[-1]`	`-1` refers to the last item,
`a[-2]`	`-2` is the second-last, and so on;
`a[-7]`	raises `IndexError` since -6=`-len(a)` is the lowest (negative) valid index;

(The same indexing scheme is used also by `tuple`s, `str`ings, ...)

list	[	'a',	'b',	'c',	'd',	'e',	'f']
indices		0	1	2	3	4	5
from the end		-6	-5	-4	-3	-2	-1

Lists and tuples – indexing [1/2]

Execute:

```
a = ['a', 'b', 'c', 'd', 'e', 'f']
```

`len(a)`	the length of `a` is 6;
`a[0]`	`0` refers to the first item,
`a[1]`	`1` refers to the second item, and so on;
`a[6]`	raises `IndexError` since 5=`len(a)-1` is the highest valid index;
`a[-1]`	`-1` refers to the last item,
`a[-2]`	`-2` is the second-last, and so on;
`a[-7]`	raises `IndexError` since -6=`-len(a)` is the lowest (negative) valid index;

(The same indexing scheme is used also by `tuple`s, `str`ings, ...)

list	[	'a',	'b',	'c',	'd',	'e',	'f']
indices		0	1	2	3	4	5
from the end		-6	-5	-4	-3	-2	-1

Lists and tuples – indexing [1/2]

Execute:

```
a = ['a', 'b', 'c', 'd', 'e', 'f']
```

`len(a)`	the length of `a` is 6;
`a[0]`	`0` refers to the first item,
`a[1]`	`1` refers to the second item, and so on;
`a[6]`	raises `IndexError` since 5=`len(a)-1` is the highest valid index;
`a[-1]`	`-1` refers to the last item,
`a[-2]`	`-2` is the second-last, and so on;
`a[-7]`	raises `IndexError` since -6=-`len(a)` is the lowest (negative) valid index;

(The same indexing scheme is used also by `tuples`, `strings`, ...)

list	[	'a',	'b',	'c',	'd',	'e',	'f']
indices		0	1	2	3	4	5
from the end		-6	-5	-4	-3	-2	-1

Lists and tuples – indexing [1/2]

Execute:

```
a = ['a', 'b', 'c', 'd', 'e', 'f']
```

`len(a)`	the length of `a` is 6;
`a[0]`	`0` refers to the first item,
`a[1]`	`1` refers to the second item, and so on;
`a[6]`	raises `IndexError` since 5=`len(a)-1` is the highest valid index;
`a[-1]`	`-1` refers to the last item,
`a[-2]`	`-2` is the second-last, and so on;
`a[-7]`	raises `IndexError` since -6=`-len(a)` is the lowest (negative) valid index;

(The same indexing scheme is used also by **`tuple`**s, **`str`**ings, ...)

list	[	'a',	'b',	'c',	'd',	'e',	'f']
indices		0	1	2	3	4	5
from the end		-6	-5	-4	-3	-2	-1

Lists and tuples – slicing [2/2]

`a[2:4]`	slice (sublist) of `a`; items with indices in range $[2, 4)$; 4 is the first index **not** in the selected slice; you can also think of the indices as pointing **between** and just **before** items;
`a[2:]`	= `a[2:len(a)]`; no upper bound = "to the end";
`a[:4]`	= `a[0:4]`; no lower bound = "from the beginning";
`a[-3:-1]`	negative indices access items from the end;
`a[-3:]`	= `a[-3:len(a)]`; last 3 items;
`a[1:5:2]`	every second item in range $[1, 5)$; 2 is a step;
`a[5:1:-2]`	step can be negative (with reversed range!);
`a[:]`	(shallow copy of) the whole list `a`;
`a[::-1]`	reversed (shallow copy of) the whole list `a`;
`a[4:-3]`	an empty range gives an empty list (no error);

list	[	'a',	'b' ,	'c',	'd',	'e',	'f'	]
indices		0	1	[2	3	4)	5	
from the end		-6	-5	-4	-3	-2	-1	

Lists and tuples – slicing [2/2]

`a[2:4]`	slice (sublist) of `a`; items with indices in range $[2, 4)$; 4 is the first index **not** in the selected slice; you can also think of the indices as pointing **between** and just **before** items;
`a[2:]`	= `a[2:len(a)]`; no upper bound = "to the end";
`a[:4]`	= `a[0:4]`; no lower bound = "from the beginning";
`a[-3:-1]`	negative indices access items from the end;
`a[-3:]`	= `a[-3:len(a)]`; last 3 items;
`a[1:5:2]`	every second item in range $[1, 5)$; 2 is a step;
`a[5:1:-2]`	step can be negative (with reversed range!);
`a[:]`	(shallow copy of) the whole list `a`;
`a[::-1]`	reversed (shallow copy of) the whole list `a`;
`a[4:-3]`	an empty range gives an empty list (no error);

list	[	'a',	'b' ,	'c',	'd',	'e',	'f']
indices		0	1	↖2	3	↖4	5
from the end		-6	-5	-4	-3	-2	-1

Lists and tuples – slicing [2/2]

`a[2:4]`	slice (sublist) of `a`; items with indices in range $[2, 4)$; 4 is the first index **not** in the selected slice; you can also think of the indices as pointing **between** and just **before** items;
`a[2:]`	= `a[2:len(a)]`; no upper bound = "to the end";
`a[:4]`	= `a[0:4]`; no lower bound = "from the beginning";
`a[-3:-1]`	negative indices access items from the end;
`a[-3:]`	= `a[-3:len(a)]`; last 3 items;
`a[1:5:2]`	every second item in range $[1, 5)$; 2 is a step;
`a[5:1:-2]`	step can be negative (with reversed range!);
`a[:]`	(shallow copy of) the whole list `a`;
`a[::-1]`	reversed (shallow copy of) the whole list `a`;
`a[4:-3]`	an empty range gives an empty list (no error);

list	[	'a',	'b',	'c',	'd',	'e',	'f']
indices		0	1	[2	3	4	5
from the end		-6	-5	-4	-3	-2	-1

Lists and tuples – slicing [2/2]

`a[2:4]`	slice (sublist) of `a`; items with indices in range $[2, 4)$; 4 is the first index **not** in the selected slice; you can also think of the indices as pointing **between** and just **before** items;
`a[2:]`	= `a[2:len(a)]`; no upper bound = "to the end";
`a[:4]`	= `a[0:4]`; no lower bound = "from the beginning";
`a[-3:-1]`	negative indices access items from the end;
`a[-3:]`	= `a[-3:len(a)]`; last 3 items;
`a[1:5:2]`	every second item in range $[1, 5)$; 2 is a step;
`a[5:1:-2]`	step can be negative (with reversed range!);
`a[:]`	(shallow copy of) the whole list `a`;
`a[::-1]`	reversed (shallow copy of) the whole list `a`;
`a[4:-3]`	an empty range gives an empty list (no error);

list	[	'a',	'b' ,	'c',	'd',	'e',	'f']
indices		0	1	2	3	4)	5
from the end		-6	-5	-4	-3	-2	-1

Lists and tuples – slicing [2/2]

`a[2:4]`	slice (sublist) of `a`; items with indices in range $[2, 4)$; 4 is the first index **not** in the selected slice; you can also think of the indices as pointing **between** and just **before** items;
`a[2:]`	= `a[2:len(a)]`; no upper bound = "to the end";
`a[:4]`	= `a[0:4]`; no lower bound = "from the beginning";
`a[-3:-1]`	negative indices access items from the end;
`a[-3:]`	= `a[-3:len(a)]`; last 3 items;
`a[1:5:2]`	every second item in range $[1, 5)$; 2 is a step;
`a[5:1:-2]`	step can be negative (with reversed range!);
`a[:]`	(shallow copy of) the whole list `a`;
`a[::-1]`	reversed (shallow copy of) the whole list `a`;
`a[4:-3]`	an empty range gives an empty list (no error);

list	[	'a',	'b',	'c',	'd',	'e',	'f']
indices		0	1	2	3	4	5
from the end		-6	-5	-4	[-3	-2	-1)

Lists and tuples – slicing [2/2]

`a[2:4]`	slice (sublist) of `a`; items with indices in range $[2, 4)$; 4 is the first index **not** in the selected slice; you can also think of the indices as pointing **between** and just **before** items;
`a[2:]`	= `a[2:len(a)]`; no upper bound = "to the end";
`a[:4]`	= `a[0:4]`; no lower bound = "from the beginning";
`a[-3:-1]`	negative indices access items from the end;
`a[-3:]`	= `a[-3:len(a)]`; last 3 items;
`a[1:5:2]`	every second item in range $[1, 5)$; 2 is a step;
`a[5:1:-2]`	step can be negative (with reversed range!);
`a[:]`	(shallow copy of) the whole list `a`;
`a[::-1]`	reversed (shallow copy of) the whole list `a`;
`a[4:-3]`	an empty range gives an empty list (no error);

list	[	'a',	'b' ,	'c',	'd',	'e',	'f']
indices		0	1	2	3	4	5
from the end		-6	-5	-4	[-3	-2	-1

Lists and tuples – slicing [2/2]

`a[2:4]`	slice (sublist) of `a`; items with indices in range $[2, 4)$; 4 is the first index **not** in the selected slice; you can also think of the indices as pointing **between** and just **before** items;
`a[2:]`	= `a[2:len(a)]`; no upper bound = "to the end";
`a[:4]`	= `a[0:4]`; no lower bound = "from the beginning";
`a[-3:-1]`	negative indices access items from the end;
`a[-3:]`	= `a[-3:len(a)]`; last 3 items;
`a[1:5:2]`	every second item in range $[1, 5)$; 2 is a step;
`a[5:1:-2]`	step can be negative (with reversed range!);
`a[:]`	(shallow copy of) the whole list `a`;
`a[::-1]`	reversed (shallow copy of) the whole list `a`;
`a[4:-3]`	an empty range gives an empty list (no error);

list	[	'a',	'b' ,	'c',	'd',	'e',	'f']
indices		0	[1	2	3	4	5)
from the end		-6	-5	-4	-3	-2	-1

Lists and tuples – slicing [2/2]

`a[2:4]`	slice (sublist) of `a`; items with indices in range $[2, 4)$; 4 is the first index **not** in the selected slice; you can also think of the indices as pointing **between** and just **before** items;
`a[2:]`	= `a[2:len(a)]`; no upper bound = "to the end";
`a[:4]`	= `a[0:4]`; no lower bound = "from the beginning";
`a[-3:-1]`	negative indices access items from the end;
`a[-3:]`	= `a[-3:len(a)]`; last 3 items;
`a[1:5:2]`	every second item in range $[1, 5)$; 2 is a step;
`a[5:1:-2]`	step can be negative (with reversed range!);
`a[:]`	(shallow copy of) the whole list `a`;
`a[::-1]`	reversed (shallow copy of) the whole list `a`;
`a[4:-3]`	an empty range gives an empty list (no error);

list	[	'a',	'b' ,	'c',	'd',	'e',	'f'	]
indices		0	(1	2	3	4	5]	
from the end		-6	-5	-4	-3	-2	-1	

Lists and tuples – slicing [2/2]

`a[2:4]`	slice (sublist) of `a`; items with indices in range $[2, 4)$; 4 is the first index **not** in the selected slice; you can also think of the indices as pointing **between** and just **before** items;
`a[2:]`	= `a[2:len(a)]`; no upper bound = "to the end";
`a[:4]`	= `a[0:4]`; no lower bound = "from the beginning";
`a[-3:-1]`	negative indices access items from the end;
`a[-3:]`	= `a[-3:len(a)]`; last 3 items;
`a[1:5:2]`	every second item in range $[1, 5)$; 2 is a step;
`a[5:1:-2]`	step can be negative (with reversed range!);
`a[:]`	(shallow copy of) the whole list `a`;
`a[::-1]`	reversed (shallow copy of) the whole list `a`;
`a[4:-3]`	an empty range gives an empty list (no error);

list	[	'a',	'b',	'c',	'd',	'e',	'f']
indices		0	1	2	3	4	5
from the end		-6	-5	-4	-3	-2	-1

Lists and tuples – slicing [2/2]

`a[2:4]`	slice (sublist) of `a`; items with indices in range $[2, 4)$; 4 is the first index **not** in the selected slice; you can also think of the indices as pointing **between** and just **before** items;
`a[2:]`	= `a[2:len(a)]`; no upper bound = "to the end";
`a[:4]`	= `a[0:4]`; no lower bound = "from the beginning";
`a[-3:-1]`	negative indices access items from the end;
`a[-3:]`	= `a[-3:len(a)]`; last 3 items;
`a[1:5:2]`	every second item in range $[1, 5)$; 2 is a step;
`a[5:1:-2]`	step can be negative (with reversed range!);
`a[:]`	(shallow copy of) the whole list `a`;
`a[::-1]`	reversed (shallow copy of) the whole list `a`;
`a[4:-3]`	an empty range gives an empty list (no error);

list	[	'a',	'b',	'c',	'd',	'e',	'f'	]
indices		0	1	2	3	4	5	⟵
from the end		-6	-5	-4	-3	-2	-1	

Lists and tuples – slicing [2/2]

`a[2:4]`	slice (sublist) of `a`; items with indices in range $[2, 4)$; 4 is the first index **not** in the selected slice; you can also think of the indices as pointing **between** and just **before** items;
`a[2:]`	= `a[2:len(a)]`; no upper bound = "to the end";
`a[:4]`	= `a[0:4]`; no lower bound = "from the beginning";
`a[-3:-1]`	negative indices access items from the end;
`a[-3:]`	= `a[-3:len(a)]`; last 3 items;
`a[1:5:2]`	every second item in range $[1, 5)$; 2 is a step;
`a[5:1:-2]`	step can be negative (with reversed range!);
`a[:]`	(shallow copy of) the whole list `a`;
`a[::-1]`	reversed (shallow copy of) the whole list `a`;
`a[4:-3]`	an empty range gives an empty list (no error);

list	[	'a',	'b' ,	'c',	'd',	'e',	'f'	]
indices		0	1	2	3	[4	5	
from the end		-6	-5	-4	-3)	-2	-1	

Slices – exercises

Slices are supported also by other sequence types, like **`tuple`**s or **`str`**ings.

Exercise: Construct the tuple
`t = ("ab", (2,3), "e", "f", 9)`
and use slice operation on `t` to select:

1. items with indices in range $[1, 3)$
2. items with indices in range $[1, 3]$
3. first 4 items
4. all items except the first and the last ones
5. the last 2 items
6. everything except the last 2 items
7. every third item of the whole `t`
8. every third item of reversed `t`

Slices – exercises

Slices are supported also by other sequence types, like **`tuple`**s or **`str`**ings.

Exercise: Construct the tuple
`t = ("ab", (2,3), "e", "f", 9)`
and use slice operation on `t` to select:

1. items with indices in range $[1, 3)$
2. items with indices in range $[1, 3]$
3. first 4 items
4. all items except the first and the last ones
5. the last 2 items
6. everything except the last 2 items
7. every third item of the whole `t`
8. every third item of reversed `t`

Slices – exercises

Slices are supported also by other sequence types, like **`tuple`**s or **`str`**ings.

Exercise: Construct the tuple
`t = ("ab", (2,3), "e", "f", 9)`
and use slice operation on `t` to select:

1. items with indices in range $[1, 3)$ Code: `t[1:3]`
2. items with indices in range $[1, 3]$ Code: `t[1:4]`
3. first 4 items Code: `t[:4]`
4. all items except the first and the last ones Code: `t[1:-1]`
5. the last 2 items Code: `t[-2:]`
6. everything except the last 2 items Code: `t[:-2]`
7. every third item of the whole `t` Code: `t[::3]`
8. every third item of reversed `t` Code: `t[::-3]`

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['a', 'b']

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['a', 'b', 'a', 'b', 'a', 'b']

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['a', 'b', 'a', 'b', 'a', 'b']

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['a', 'b', 'a', 'b', 'a', ~~'b'~~9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['a', 'b', 'a', 'b', 'a', 9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['a', 'b', 'a', 'b', 'a', 9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 `str`ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other `list` as the last item;
`a.extend([8,9])`	appends items from the other `list`;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a `tuple` content replaces the first 5 items;
`del a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as `del a[:]`.

['b', 'a', 'b', 'a', 9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['b', ~~'a',~~ 'b', 'a', 9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['b', 'b', 'a', 9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['b', ~~'b', 'a',~~ 9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

```
['b', 9]
```

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['b', 9, 'zz']

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['b', 9, 'zz']

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['b', 9, 7, 'zz']

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['b', 9, 7, 'zz']

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['b', 9, 7, 'zz', [8, 9]]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['b', 9, 7, 'zz', [8, 9]]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['b', 9, 7, 'zz', [8, 9], 8, 9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['b', 9, 7, 'zz', [8, 9], 8, 9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['b', 9, 7, 'zz', [8, 9], 8, 9, 8, 9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['b', 9, 7, 'zz', [8, 9], 8, 9, 8, 9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

['b', 9, 7, 'zz', [8, 9], 1, 2, 8, 9, 8, 9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

[1, 2, 8, 9, 8, 9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

[~~1,~~ 2, ~~8,~~ 9, ~~8,~~ 9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

[2, 9, 9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

[2, 0, 0, 0, 0, 9, 9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

[2, 0, 0, 0, 0, 9, 9]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

[~~2, 0, 0, 0, 0, 9, 9~~]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

[]

Modifying list – examples

`a = ['a', 'b']`	constructs a list of 2 **str**ings;
`a *= 3`	the same as `a = a * 3`;
`a[-1] = 9`	replaces the last item with `9`;
`del` `a[0]`	deletes the first item;
`a.remove('a')`	remove the first occurrence of `'a'`;
`del` `a[1:3]`	deletes slice, items with indices 1 and 2;
`a.append('zz')`	appends `'zz'` at the end of `a`;
`a.insert(2, 7)`	inserts `7` at index 2;
`a.append([8,9])`	appends the other **`list`** as the last item;
`a.extend([8,9])`	appends items from the other **`list`**;
`a += [8,9]`	the same as the previous one;
`a[:5] = 1,2`	a **`tuple`** content replaces the first 5 items;
`del` `a[::2]`	deletes every second item;
`a[1:1]=[0]*4`	inserts 4 zeros by replacing an empty slice;
`a.clear()`	removes all items; the same as **`del`** `a[:]`.

Instances of **`tuple`** and **`str`** cannot change after being created.

Modifying lists – exercises

1. construct the `list` `x` $= [\overbrace{1, 1, \ldots, 1}^{10}, \overbrace{2, 2, \ldots, 2}^{10}]$
2. change the third element of `x` to `'abc'`
3. insert `'yz'` `str`ing exactly in the middle of `x`
4. append four `'yz'` `str`ings to the end of `x`
5. delete the first 6 items of `x`
6. delete each third item of `x` (starting from the first one)
7. replace the first occurrence of `'yz'` in `x` with `'bb'`

Modifying lists – exercises

1. construct the **list** `x` $= [\overbrace{1, 1, \ldots, 1}^{10}, \overbrace{2, 2, \ldots, 2}^{10}]$
   Code: `x = [1]*10 + [2]*10`
2. change the third element of `x` to `'abc'`
   Code: `x[2] = 'abc'`
3. insert `'yz'` **str**ing exactly in the middle of `x`
   Code: `x.insert(len(x)//2, 'yz')`
4. append four `'yz'` **str**ings to the end of `x`
   Code: `x += ('yz',)*4`
5. delete the first 6 items of `x`
   Code: `del x[:6]`
6. delete each third item of `x` (starting from the first one)
   Code: `del x[::3]`
7. replace the first occurrence of `'yz'` in `x` with `'bb'`
   Code: `x[x.index('yz')] = 'bb'`

Copying lists and another objects

Operator = assigns the reference to the object, **without** copying it:

`a = [1, [2, 3]]`	constructs a `list`;
`b = a`	now `b` refers to the same object as `a` does;
`a[0] = 5`	modifies the object referenced by `a` (and `b`);
`a[1][0] = 7`	modifies the `list` nested in this object;
`print(b)`	displays `b`: `[5, [7, 3]]`

To change `a` without changing `b`, you can use one of the following methods (in place of `b = a`) to make a copy of `a`:

- `b = list(a)` – constructor makes a shallow copy;
- `b = a[:]` – slice operator also makes a shallow copy;
- `import copy` imports module for copying objects of any type:
- `b = copy.copy(a)` – shallow copy (`a` can be of any type);
- `b = copy.deepcopy(a)` – deep copy (`a` can be of any type).

After **shallow copy**: the members of `a` and `b` refer to the same objects. Final `b` in our example: `[1, [7, 3]]`

After **deep copy**: all the members (like nested lists) have been (deep) copied. Final `b` in our example: `[1, [2, 3]]`

Copying lists and another objects

Operator = assigns the reference to the object, **without** copying it:

`a = [1, [2, 3]]`	constructs a **`list`**;
`b = a`	now `b` refers to the same object as `a` does;
`a[0] = 5`	modifies the object referenced by `a` (and `b`);
`a[1][0] = 7`	modifies the **`list`** nested in this object;
`print``(b)`	displays `b`: `[5, [7, 3]]`

To change `a` without changing `b`, you can use one of the following methods (in place of `b = a`) to make a copy of `a`:

- `b =` **`list`**`(a)` – constructor makes a shallow copy;
- `b = a[:]` – slice operator also makes a shallow copy;
- **`import`** `copy` imports module for copying objects of any type:
- `b = copy.copy(a)` – shallow copy (`a` can be of any type);
- `b = copy.deepcopy(a)` – deep copy (`a` can be of any type).

After **shallow copy**: the members of `a` and `b` refer to the same objects. Final `b` in our example: `[1, [7, 3]]`

After **deep copy**: all the members (like nested lists) have been (deep) copied. Final `b` in our example: `[1, [2, 3]]`

Copying lists and another objects

Operator = assigns the reference to the object, **without** copying it:

`a = [1, [2, 3]]`	constructs a **`list`**;
`b = a`	now `b` refers to the same object as `a` does;
`a[0] = 5`	modifies the object referenced by `a` (and `b`);
`a[1][0] = 7`	modifies the **`list`** nested in this object;
`print(b)`	displays `b`: `[5, [7, 3]]`

To change `a` without changing `b`, you can use one of the following methods (in place of `b = a`) to make a copy of `a`:

- `b = list(a)` – constructor makes a shallow copy;
- `b = a[:]` – slice operator also makes a shallow copy;
- **`import`** `copy` imports module for copying objects of any type:
- `b = copy.copy(a)` – shallow copy (`a` can be of any type);
- `b = copy.deepcopy(a)` – deep copy (`a` can be of any type).

After **shallow copy**: the members of `a` and `b` refer to the same objects. Final `b` in our example: `[1, [7, 3]]`

After **deep copy**: all the members (like nested lists) have been (deep) copied. Final `b` in our example: `[1, [2, 3]]`

Copying lists and another objects

Operator = assigns the reference to the object, **without** copying it:

`a = [1, [2, 3]]`	constructs a **`list`**;
`b = a`	now `b` refers to the same object as `a` does;
`a[0] = 5`	modifies the object referenced by `a` (and `b`);
`a[1][0] = 7`	modifies the **`list`** nested in this object;
`print(b)`	displays `b`: `[5, [7, 3]]`

To change `a` without changing `b`, you can use one of the following methods (in place of `b = a`) to make a copy of `a`:

- `b = list(a)` – constructor makes a shallow copy;
- `b = a[:]` – slice operator also makes a shallow copy;
- **`import`** `copy` imports module for copying objects of any type:
- `b = copy.copy(a)` – shallow copy (`a` can be of any type);
- `b = copy.deepcopy(a)` – deep copy (`a` can be of any type).

After **shallow copy**: the members of `a` and `b` refer to the same objects. Final `b` in our example: `[1, [7, 3]]`

After **deep copy**: all the members (like nested lists) have been (deep) copied. Final `b` in our example: `[1, [2, 3]]`

Copying lists and another objects

Operator = assigns the reference to the object, **without** copying it:

`a = [1, [2, 3]]`	constructs a **`list`**;
`b = a`	now `b` refers to the same object as `a` does;
`a[0] = 5`	modifies the object referenced by `a` (and `b`);
`a[1][0] = 7`	modifies the **`list`** nested in this object;
`print(b)`	displays `b`: `[5, [7, 3]]`

To change `a` without changing `b`, you can use one of the following methods (in place of `b = a`) to make a copy of `a`:

- `b = list(a)` – constructor makes a shallow copy;
- `b = a[:]` – slice operator also makes a shallow copy;
- **`import`** `copy` imports module for copying objects of any type:
- `b = copy.copy(a)` – shallow copy (`a` can be of any type);
- `b = copy.deepcopy(a)` – deep copy (`a` can be of any type).

After **shallow copy**: the members of `a` and `b` refer to the same objects. Final `b` in our example: `[1, [7, 3]]`

After **deep copy**: all the members (like nested lists) have been (deep) copied. Final `b` in our example: `[1, [2, 3]]`

Copying lists and another objects

Operator = assigns the reference to the object, **without** copying it:

`a = [1, [2, 3]]`	constructs a **`list`**;
`b = a`	now `b` refers to the same object as `a` does;
`a[0] = 5`	modifies the object referenced by `a` (and `b`);
`a[1][0] = 7`	modifies the **`list`** nested in this object;
`print``(b)`	displays `b`: `[5, [7, 3]]`

To change `a` without changing `b`, you can use one of the following methods (in place of `b = a`) to make a copy of `a`:

- `b = list(a)` – constructor makes a shallow copy;
- `b = a[:]` – slice operator also makes a shallow copy;
- **`import`** `copy` imports module for copying objects of any type:
- `b = copy.copy(a)` – shallow copy (`a` can be of any type);
- `b = copy.deepcopy(a)` – deep copy (`a` can be of any type).

After **shallow copy**: the members of `a` and `b` refer to the same objects. Final `b` in our example: `[1, [7, 3]]`

After **deep copy**: all the members (like nested lists) have been (deep) copied. Final `b` in our example: `[1, [2, 3]]`

Copying lists and another objects

Operator = assigns the reference to the object, **without** copying it:

`a = [1, [2, 3]]`	constructs a **`list`**;
`b = a`	now `b` refers to the same object as `a` does;
`a[0] = 5`	modifies the object referenced by `a` (and `b`);
`a[1][0] = 7`	modifies the **`list`** nested in this object;
`print``(b)`	displays `b`: `[5, [7, 3]]`

To change `a` without changing `b`, you can use one of the following methods (in place of `b = a`) to make a copy of `a`:

- `b =` **`list`**`(a)` – constructor makes a shallow copy;
- `b = a[:]` – slice operator also makes a shallow copy;
- **`import`** `copy` imports module for copying objects of any type:
- `b = copy.copy(a)` – shallow copy (`a` can be of any type);
- `b = copy.deepcopy(a)` – deep copy (`a` can be of any type).

After **shallow copy**: the members of `a` and `b` refer to the same objects. Final `b` in our example: `[1, [7, 3]]`

After **deep copy**: all the members (like nested lists) have been (deep) copied. Final `b` in our example: `[1, [2, 3]]`

Copying lists and another objects

Operator = assigns the reference to the object, **without** copying it:

`a = [1, [2, 3]]`	constructs a **`list`**;
`b = a`	now `b` refers to the same object as `a` does;
`a[0] = 5`	modifies the object referenced by `a` (and `b`);
`a[1][0] = 7`	modifies the **`list`** nested in this object;
`print``(b)`	displays `b`: `[5, [7, 3]]`

To change `a` without changing `b`, you can use one of the following methods (in place of `b = a`) to make a copy of `a`:

- `b =` **`list`**`(a)` – constructor makes a shallow copy;
- `b = a[:]` – slice operator also makes a shallow copy;
- **`import`** `copy` imports module for copying objects of any type:
- `b = copy.copy(a)` – shallow copy (`a` can be of any type);
- `b = copy.deepcopy(a)` – deep copy (`a` can be of any type).

After **shallow copy**: the members of `a` and `b` refer to the same objects. Final `b` in our example: `[1, [7, 3]]`

After **deep copy**: all the members (like nested lists) have been (deep) copied. Final `b` in our example: `[1, [2, 3]]`

Copying lists and another objects

Operator = assigns the reference to the object, **without** copying it:

`a = [1, [2, 3]]`	constructs a **`list`**;
`b = a`	now `b` refers to the same object as `a` does;
`a[0] = 5`	modifies the object referenced by `a` (and `b`);
`a[1][0] = 7`	modifies the **`list`** nested in this object;
`print``(b)`	displays `b`: `[5, [7, 3]]`

To change `a` without changing `b`, you can use one of the following methods (in place of `b = a`) to make a copy of `a`:

- `b =` **`list`**`(a)` – constructor makes a shallow copy;
- `b = a[:]` – slice operator also makes a shallow copy;
- **`import`** `copy` imports module for copying objects of any type:
- `b = copy.copy(a)` – shallow copy (`a` can be of any type);
- `b = copy.deepcopy(a)` – deep copy (`a` can be of any type).

After **shallow copy**: the members of `a` and `b` refer to the same objects. Final `b` in our example: `[1, [7, 3]]`

After **deep copy**: all the members (like nested lists) have been (deep) copied. Final `b` in our example: `[1, [2, 3]]`

Copying lists and another objects

Operator = assigns the reference to the object, **without** copying it:

`a = [1, [2, 3]]`	constructs a **`list`**;
`b = a`	now `b` refers to the same object as `a` does;
`a[0] = 5`	modifies the object referenced by `a` (and `b`);
`a[1][0] = 7`	modifies the **`list`** nested in this object;
`print``(b)`	displays `b`: `[5, [7, 3]]`

To change `a` without changing `b`, you can use one of the following methods (in place of `b = a`) to make a copy of `a`:

- `b =` **`list`**`(a)` – constructor makes a shallow copy;
- `b = a[:]` – slice operator also makes a shallow copy;
- **`import`** `copy` imports module for copying objects of any type:
  - `b = copy.copy(a)` – shallow copy (`a` can be of any type);
  - `b = copy.deepcopy(a)` – deep copy (`a` can be of any type).

After **shallow copy**: the members of `a` and `b` refer to the same objects. Final `b` in our example: `[1, [7, 3]]`

After **deep copy**: all the members (like nested lists) have been (deep) copied. Final `b` in our example: `[1, [2, 3]]`

Copying lists and another objects

Operator = assigns the reference to the object, **without** copying it:

`a = [1, [2, 3]]`	constructs a **`list`**;
`b = a`	now `b` refers to the same object as `a` does;
`a[0] = 5`	modifies the object referenced by `a` (and `b`);
`a[1][0] = 7`	modifies the **`list`** nested in this object;
`print``(b)`	displays `b`: `[5, [7, 3]]`

To change `a` without changing `b`, you can use one of the following methods (in place of `b = a`) to make a copy of `a`:

- `b =` **`list`**`(a)` – constructor makes a shallow copy;
- `b = a[:]` – slice operator also makes a shallow copy;
- **`import`** `copy` imports module for copying objects of any type:
- `b = copy.copy(a)` – shallow copy (`a` can be of any type);
- `b = copy.deepcopy(a)` – deep copy (`a` can be of any type).

After **shallow copy**: the members of `a` and `b` refer to the same objects. Final `b` in our example: `[1, [7, 3]]`

After **deep copy**: all the members (like nested lists) have been (deep) copied. Final `b` in our example: `[1, [2, 3]]`

Copying lists and another objects

Operator = assigns the reference to the object, **without** copying it:

`a = [1, [2, 3]]`	constructs a **`list`**;
`b = a`	now `b` refers to the same object as `a` does;
`a[0] = 5`	modifies the object referenced by `a` (and `b`);
`a[1][0] = 7`	modifies the **`list`** nested in this object;
`print``(b)`	displays `b`: `[5, [7, 3]]`

To change `a` without changing `b`, you can use one of the following methods (in place of `b = a`) to make a copy of `a`:

- `b =` **`list`**`(a)` – constructor makes a shallow copy;
- `b = a[:]` – slice operator also makes a shallow copy;
- **`import`** `copy` imports module for copying objects of any type:
- `b = copy.copy(a)` – shallow copy (`a` can be of any type);
- `b = copy.deepcopy(a)` – deep copy (`a` can be of any type).

After **shallow copy**: the members of `a` and `b` refer to the same objects. Final `b` in our example: `[1, [7, 3]]`

After **deep copy**: all the members (like nested lists) have been (deep) copied. Final `b` in our example: `[1, [2, 3]]`

Copying lists and another objects

Operator = assigns the reference to the object, **without** copying it:

`a = [1, [2, 3]]`	constructs a **`list`**;
`b = a`	now `b` refers to the same object as `a` does;
`a[0] = 5`	modifies the object referenced by `a` (and `b`);
`a[1][0] = 7`	modifies the **`list`** nested in this object;
`print``(b)`	displays `b`: `[5, [7, 3]]`

To change `a` without changing `b`, you can use one of the following methods (in place of `b = a`) to make a copy of `a`:

- `b =` **`list`**`(a)` – constructor makes a shallow copy;
- `b = a[:]` – slice operator also makes a shallow copy;
- **`import`** `copy` imports module for copying objects of any type:
- `b = copy.copy(a)` – shallow copy (`a` can be of any type);
- `b = copy.deepcopy(a)` – deep copy (`a` can be of any type).

After **shallow copy**: the members of `a` and `b` refer to the same objects. Final `b` in our example: `[1, [7, 3]]`

After **deep copy**: all the members (like nested lists) have been (deep) copied. Final `b` in our example: `[1, [2, 3]]`

Copying lists and another objects

Operator = assigns the reference to the object, **without** copying it:

`a = [1, [2, 3]]`	constructs a **`list`**;
`b = a`	now `b` refers to the same object as `a` does;
`a[0] = 5`	modifies the object referenced by `a` (and `b`);
`a[1][0] = 7`	modifies the **`list`** nested in this object;
`print``(b)`	displays `b`: `[5, [7, 3]]`

To change `a` without changing `b`, you can use one of the following methods (in place of `b = a`) to make a copy of `a`:

- `b =` **`list`**`(a)` – constructor makes a shallow copy;
- `b = a[:]` – slice operator also makes a shallow copy;
- **`import`** `copy` imports module for copying objects of any type:
- `b = copy.copy(a)` – shallow copy (`a` can be of any type);
- `b = copy.deepcopy(a)` – deep copy (`a` can be of any type).

After **shallow copy**: the members of `a` and `b` refer to the same objects. Final `b` in our example: `[1, [7, 3]]`

After **deep copy**: all the members (like nested lists) have been (deep) copied. Final `b` in our example: `[1, [2, 3]]`

Dictionaries

A **dict**ionary is an unordered, mutable collection which maps unique hashable (≈immutable) key objects to value objects.

Execute:

`d = {'a':1, 3:5}`	constructs a `dict` instance;
`len(d)`	`d` has 2 items (key–value pairs);
`d['a']`	returns the value for the key `'a'`;
`d.get('a')`	similar to the above;
`d[1]`	raises `KeyError`; key not found;
`d.get(1)`	in a similar situation, `get` returns `None`,
`d.get(1, 0)`	or value given as a second argument;
`d['a'] = 2`	changes the value for the key `'a'` to `2`;
`d['zz'] = 6`	adds the key `'zz'` with the value `6`;
`del d[3]`	erases the key `3` together with its value;
`d[5] = [1,2]`	a `list` can be a value,
`d[[1,2]] = 5`	but not a key, as it is unhashable/mutable;
`d[1,2] = 5`	OK; a `tuple` is hashable (and immutable);
`d.copy()`	a shallow copy of `d`.

Dictionaries

A **dict**ionary is an unordered, mutable collection which maps unique hashable (≈immutable) key objects to value objects.

Execute:

`d = {'a':1, 3:5}`	constructs a **dict** instance;
`len(d)`	`d` has 2 items (key–value pairs);
`d['a']`	returns the value for the key `'a'`;
`d.get('a')`	similar to the above;
`d[1]`	raises `KeyError`; key not found;
`d.get(1)`	in a similar situation, `get` returns `None`,
`d.get(1, 0)`	or value given as a second argument;
`d['a'] = 2`	changes the value for the key `'a'` to `2`;
`d['zz'] = 6`	adds the key `'zz'` with the value `6`;
`del d[3]`	erases the key `3` together with its value;
`d[5] = [1,2]`	a **list** can be a value,
`d[[1,2]] = 5`	but not a key, as it is unhashable/mutable;
`d[1,2] = 5`	OK; a **tuple** is hashable (and immutable);
`d.copy()`	a shallow copy of `d`.

Dictionaries

A **dict**ionary is an unordered, mutable collection which maps unique hashable (≈immutable) key objects to value objects.

Execute:

`d = {'a':1, 3:5}`	constructs a **dict** instance;
len`(d)`	`d` has 2 items (key–value pairs);
`d['a']`	returns the value for the key `'a'`;
`d.get('a')`	similar to the above;
`d[1]`	raises `KeyError`; key not found;
`d.get(1)`	in a similar situation, `get` returns `None`,
`d.get(1, 0)`	or value given as a second argument;
`d['a'] = 2`	changes the value for the key `'a'` to `2`;
`d['zz'] = 6`	adds the key `'zz'` with the value `6`;
del `d[3]`	erases the key `3` together with its value;
`d[5] = [1,2]`	a **list** can be a value,
`d[[1,2]] = 5`	but not a key, as it is unhashable/mutable;
`d[1,2] = 5`	OK; a **tuple** is hashable (and immutable);
`d.copy()`	a shallow copy of `d`.

Dictionaries

A **dict**ionary is an unordered, mutable collection which maps unique hashable (≈immutable) key objects to value objects.

Execute:

`d = {'a':1, 3:5}`	constructs a **dict** instance;
len`(d)`	`d` has 2 items (key–value pairs);
`d['a']`	returns the value for the key `'a'`;
`d.get('a')`	similar to the above;
`d[1]`	raises `KeyError`; key not found;
`d.get(1)`	in a similar situation, `get` returns `None`,
`d.get(1, 0)`	or value given as a second argument;
`d['a'] = 2`	changes the value for the key `'a'` to `2`;
`d['zz'] = 6`	adds the key `'zz'` with the value `6`;
del `d[3]`	erases the key `3` together with its value;
`d[5] = [1,2]`	a **list** can be a value,
`d[[1,2]] = 5`	but not a key, as it is unhashable/mutable;
`d[1,2] = 5`	OK; a **tuple** is hashable (and immutable);
`d.copy()`	a shallow copy of `d`.

Dictionaries

A **dict**ionary is an unordered, mutable collection which maps unique hashable (≈immutable) key objects to value objects.

Execute:

`d = {'a':1, 3:5}`	constructs a **dict** instance;
len`(d)`	`d` has 2 items (key–value pairs);
`d['a']`	returns the value for the key `'a'`;
`d.get('a')`	similar to the above;
`d[1]`	raises `KeyError`; key not found;
`d.get(1)`	in a similar situation, `get` returns `None`,
`d.get(1, 0)`	or value given as a second argument;
`d['a'] = 2`	changes the value for the key `'a'` to `2`;
`d['zz'] = 6`	adds the key `'zz'` with the value `6`;
del `d[3]`	erases the key `3` together with its value;
`d[5] = [1,2]`	a **list** can be a value,
`d[[1,2]] = 5`	but not a key, as it is unhashable/mutable;
`d[1,2] = 5`	OK; a **tuple** is hashable (and immutable);
`d.copy()`	a shallow copy of `d`.

Dictionaries

A **dict**ionary is an unordered, mutable collection which maps unique hashable (≈immutable) key objects to value objects.

Execute:

`d = {'a':1, 3:5}`	constructs a **dict** instance;
len`(d)`	`d` has 2 items (key–value pairs);
`d['a']`	returns the value for the key `'a'`;
`d.get('a')`	similar to the above;
`d[1]`	raises `KeyError`; key not found;
`d.get(1)`	in a similar situation, `get` returns `None`,
`d.get(1, 0)`	or value given as a second argument;
`d['a'] = 2`	changes the value for the key `'a'` to `2`;
`d['zz'] = 6`	adds the key `'zz'` with the value `6`;
del `d[3]`	erases the key `3` together with its value;
`d[5] = [1,2]`	a **list** can be a value,
`d[[1,2]] = 5`	but not a key, as it is unhashable/mutable;
`d[1,2] = 5`	OK; a **tuple** is hashable (and immutable);
`d.copy()`	a shallow copy of `d`.

Dictionaries

A **dict**ionary is an unordered, mutable collection which maps unique hashable (≈immutable) key objects to value objects.

Execute:

`d = {'a':1, 3:5}`	constructs a **dict** instance;
len`(d)`	`d` has 2 items (key–value pairs);
`d['a']`	returns the value for the key `'a'`;
`d.get('a')`	similar to the above;
`d[1]`	raises `KeyError`; key not found;
`d.get(1)`	in a similar situation, `get` returns `None`,
`d.get(1, 0)`	or value given as a second argument;
`d['a'] = 2`	changes the value for the key `'a'` to `2`;
`d['zz'] = 6`	adds the key `'zz'` with the value `6`;
del `d[3]`	erases the key `3` together with its value;
`d[5] = [1,2]`	a **list** can be a value,
`d[[1,2]] = 5`	but not a key, as it is unhashable/mutable;
`d[1,2] = 5`	OK; a **tuple** is hashable (and immutable);
`d.copy()`	a shallow copy of `d`.

Dictionaries

A **dict**ionary is an unordered, mutable collection which maps unique hashable (≈immutable) key objects to value objects.

Execute:

`d = {'a':1, 3:5}`	constructs a **dict** instance;
len`(d)`	`d` has 2 items (key–value pairs);
`d['a']`	returns the value for the key `'a'`;
`d.get('a')`	similar to the above;
`d[1]`	raises `KeyError`; key not found;
`d.get(1)`	in a similar situation, `get` returns `None`,
`d.get(1, 0)`	or value given as a second argument;
`d['a'] = 2`	changes the value for the key `'a'` to `2`;
`d['zz'] = 6`	adds the key `'zz'` with the value `6`;
del `d[3]`	erases the key `3` together with its value;
`d[5] = [1,2]`	a **list** can be a value,
`d[[1,2]] = 5`	but not a key, as it is unhashable/mutable;
`d[1,2] = 5`	OK; a **tuple** is hashable (and immutable);
`d.copy()`	a shallow copy of `d`.

Dictionaries

A **dict**ionary is an unordered, mutable collection which maps unique hashable (≈immutable) key objects to value objects.

Execute:

`d = {'a':1, 3:5}`	constructs a **dict** instance;
`len(d)`	`d` has 2 items (key–value pairs);
`d['a']`	returns the value for the key `'a'`;
`d.get('a')`	similar to the above;
`d[1]`	raises `KeyError`; key not found;
`d.get(1)`	in a similar situation, `get` returns `None`,
`d.get(1, 0)`	or value given as a second argument;
`d['a'] = 2`	changes the value for the key `'a'` to `2`;
`d['zz'] = 6`	adds the key `'zz'` with the value `6`;
`del d[3]`	erases the key `3` together with its value;
`d[5] = [1,2]`	a **list** can be a value,
`d[[1,2]] = 5`	but not a key, as it is unhashable/mutable;
`d[1,2] = 5`	OK; a **tuple** is hashable (and immutable);
`d.copy()`	a shallow copy of `d`.

Dictionaries

A **dict**ionary is an unordered, mutable collection which maps unique hashable (≈immutable) key objects to value objects.

Execute:

`d = {'a':1, 3:5}`	constructs a **dict** instance;
`len(d)`	`d` has 2 items (key–value pairs);
`d['a']`	returns the value for the key `'a'`;
`d.get('a')`	similar to the above;
`d[1]`	raises `KeyError`; key not found;
`d.get(1)`	in a similar situation, `get` returns `None`,
`d.get(1, 0)`	or value given as a second argument;
`d['a'] = 2`	changes the value for the key `'a'` to `2`;
`d['zz'] = 6`	adds the key `'zz'` with the value `6`;
`del d[3]`	erases the key `3` together with its value;
`d[5] = [1,2]`	a **list** can be a value,
`d[[1,2]] = 5`	but not a key, as it is unhashable/mutable;
`d[1,2] = 5`	OK; a **tuple** is hashable (and immutable);
`d.copy()`	a shallow copy of `d`.

Dictionaries

A **dict**ionary is an unordered, mutable collection which maps unique hashable (≈immutable) key objects to value objects.

Execute:

`d = {'a':1, 3:5}`	constructs a **dict** instance;
`len(d)`	`d` has 2 items (key–value pairs);
`d['a']`	returns the value for the key `'a'`;
`d.get('a')`	similar to the above;
`d[1]`	raises `KeyError`; key not found;
`d.get(1)`	in a similar situation, `get` returns `None`,
`d.get(1, 0)`	or value given as a second argument;
`d['a'] = 2`	changes the value for the key `'a'` to `2`;
`d['zz'] = 6`	adds the key `'zz'` with the value `6`;
`del d[3]`	erases the key `3` together with its value;
`d[5] = [1,2]`	a **list** can be a value,
`d[[1,2]] = 5`	but not a key, as it is unhashable/mutable;
`d[1,2] = 5`	OK; a **tuple** is hashable (and immutable);
`d.copy()`	a shallow copy of `d`.

Dictionaries

A **dict**ionary is an unordered, mutable collection which maps unique hashable (≈immutable) key objects to value objects.

Execute:

`d = {'a':1, 3:5}`	constructs a **dict** instance;
`len(d)`	`d` has 2 items (key–value pairs);
`d['a']`	returns the value for the key `'a'`;
`d.get('a')`	similar to the above;
`d[1]`	raises `KeyError`; key not found;
`d.get(1)`	in a similar situation, `get` returns `None`,
`d.get(1, 0)`	or value given as a second argument;
`d['a'] = 2`	changes the value for the key `'a'` to `2`;
`d['zz'] = 6`	adds the key `'zz'` with the value `6`;
`del d[3]`	erases the key `3` together with its value;
`d[5] = [1,2]`	a **list** can be a value,
`d[[1,2]] = 5`	but not a key, as it is unhashable/mutable;
`d[1,2] = 5`	OK; a **tuple** is hashable (and immutable);
`d.copy()`	a shallow copy of `d`.

Dictionaries

A **dict**ionary is an unordered, mutable collection which maps unique hashable (≈immutable) key objects to value objects.

Execute:

`d = {'a':1, 3:5}`	constructs a **dict** instance;
len`(d)`	`d` has 2 items (key–value pairs);
`d['a']`	returns the value for the key `'a'`;
`d.get('a')`	similar to the above;
`d[1]`	raises `KeyError`; key not found;
`d.get(1)`	in a similar situation, `get` returns `None`,
`d.get(1, 0)`	or value given as a second argument;
`d['a'] = 2`	changes the value for the key `'a'` to `2`;
`d['zz'] = 6`	adds the key `'zz'` with the value `6`;
del `d[3]`	erases the key `3` together with its value;
`d[5] = [1,2]`	a **list** can be a value,
`d[[1,2]] = 5`	but not a key, as it is unhashable/mutable;
`d[1,2] = 5`	OK; a **tuple** is hashable (and immutable);
`d.copy()`	a shallow copy of `d`.

Dictionaries

A **dict**ionary is an unordered, mutable collection which maps unique hashable (≈immutable) key objects to value objects.

Execute:

`d = {'a':1, 3:5}`	constructs a **dict** instance;
`len(d)`	`d` has 2 items (key–value pairs);
`d['a']`	returns the value for the key `'a'`;
`d.get('a')`	similar to the above;
`d[1]`	raises `KeyError`; key not found;
`d.get(1)`	in a similar situation, `get` returns `None`,
`d.get(1, 0)`	or value given as a second argument;
`d['a'] = 2`	changes the value for the key `'a'` to `2`;
`d['zz'] = 6`	adds the key `'zz'` with the value `6`;
`del d[3]`	erases the key `3` together with its value;
`d[5] = [1,2]`	a **list** can be a value,
`d[[1,2]] = 5`	but not a key, as it is unhashable/mutable;
`d[1,2] = 5`	OK; a **tuple** is hashable (and immutable);
`d.copy()`	a shallow copy of `d`.

Dictionaries

A **dict**ionary is an unordered, mutable collection which maps unique hashable (≈immutable) key objects to value objects.

Execute:

`d = {'a':1, 3:5}`	constructs a **dict** instance;
len`(d)`	`d` has 2 items (key–value pairs);
`d['a']`	returns the value for the key `'a'`;
`d.get('a')`	similar to the above;
`d[1]`	raises `KeyError`; key not found;
`d.get(1)`	in a similar situation, `get` returns `None`,
`d.get(1, 0)`	or value given as a second argument;
`d['a'] = 2`	changes the value for the key `'a'` to `2`;
`d['zz'] = 6`	adds the key `'zz'` with the value `6`;
del `d[3]`	erases the key `3` together with its value;
`d[5] = [1,2]`	a **list** can be a value,
`d[[1,2]] = 5`	but not a key, as it is unhashable/mutable;
`d[1,2] = 5`	OK; a **tuple** is hashable (and immutable);
`d.copy()`	a shallow copy of `d`.

Homework

Construct the `list`

$$\texttt{b} = [\underbrace{\text{'ab', 'ab', ..., 'ab'}}_{6}, \underbrace{2, 2, \ldots, 2}_{10}, \underbrace{\text{'z', 'z', ..., 'z'}}_{15}]$$

and modify it as follows:

1. change the last item to `'last'`
2. change the item that is exactly in the middle of `b` to `'x'`
3. replace the second item with 3 items: `'a', 2, 'zz'`
4. erase the items with indices in range $[3, 9]$
5. remove the first occurrence of `2`
6. reverse the fragment of `b`, from index 2 to 11 inclusive

Next, count occurrences of `2` in `b` and find the index of the first occurrence.

Please note all the expressions you used.

Bibliography

- Official *Python documentation* available on https://docs.python.org/3/, topics: *Data Structures* (in *The Python Tutorial*), *Sequence Types – list, tuple, range* and *Mapping Types – dict* (in *Library Reference / Built-in Types*)
- Les De Shay *Python Lists and Tuples*, available on http://pythoncentral.io/python-lists-and-tuples/
- Peter Wentworth, et al. *How to Think Like a Computer Scientist: Learning with Python 3*, chapters: *Tuples*, *Lists*, *Dictionaries*, available on http://openbookproject.net/...
- *Python List* and *Python Tuple*, available on http://www.programiz.com/python-programming/list and http://www.programiz.com/python-programming/tuple