

# A Short and Incomplete Introduction to Python

## Part 3: Sequences and for-loops

**Riccardo Murri** <riccardo.murri@uzh.ch>,  
Sergio Maffioletti <sergio.maffioletti@uzh.ch>  
S3IT: Services and Support for Science IT,  
University of Zurich

# Lists and sequences

# Sequences

Python provides a few built-in *sequence* classes:

`list` *mutable*, possibly heterogeneous

`tuple` *immutable*, possibly heterogeneous

`str` *immutable*, only holds characters

Additional sequence types  
are provided by external modules:

`array` *mutable*, homogeneous  
(like C/Fortran arrays, from `NumPy`)

`DataFrame` *mutable*, heterogeneous  
(like R, from `Pandas`)

## Lists

Lists are by far the most common and used sequence type in Python.

Lists are created and initialized by enclosing values into '[' and ']':

```
>>> L = [ 'U', 'Z' ]
```

You can append and remove items from a list:

```
>>> L.append('H')
>>> print (L)
['U', 'Z', 'H']
```

You can append **any** object to a list:

```
>>> L.append([1, 2])
>>> print (L)
['U', 'Z', 'H', [1, 2]]
```

## Lists

Lists are by far the most common and used sequence type in Python.

Lists are created and initialized by enclosing values into '[' and ']':

```
>>> L = [ 'U', 'Z' ]
```

You can append and remove items from a list:

```
>>> L.append('H')
>>> print (L)
['U', 'Z', 'H']
```

You can append **any** object to a list:

```
>>> L.append([1, 2])
>>> print (L)
['U', 'Z', 'H', [1, 2]]
```

## Lists

Lists are by far the most common and used sequence type in Python.

Lists are created and initialized by enclosing values into '[' and ']':

```
>>> L = [ 'U', 'Z' ]
```

You can append and remove items from a list:

```
>>> L.append('H')
>>> print (L)
['U', 'Z', 'H']
```

You can append **any** object to a list:

```
>>> L.append([1, 2])
>>> print (L)
['U', 'Z', 'H', [1, 2]]
```

## Sequences, II

You can access individual items in a sequence using the postfix `[]` operator.

Sequence indices start at 0.

```
>>> L = ['U', 'Z', 'H']
>>> print(L[0], L[1], L[2])
'U' 'Z' 'H'
>>> S = 'UZH'
>>> print(S[0], S[1], S[2])
'U' 'Z' 'H'
```

## Sequence length

The `len()` function returns the number of items in any sequence (not just lists).

```
>>> len(L)  
3
```

Built-in functions `sum()`, `max()`, `min()` also work on list arguments.



**Exercise 3.A:** Write a function `avg()` that takes a list of numbers and returns their mean value.

## Slices

The notation `[n:m]` is used for accessing a *slice* of sequence (the items at positions  $n$ ,  $n + 1$ ,  $\dots$ ,  $m - 1$ ).

```
>>> # list numbers from 0 to 9
>>> R = list(range(0,10))
>>> R[1:4]
[1, 2, 3]
```

If  $n$  is omitted it defaults to 0, if  $m$  is omitted it defaults to the length of the sequence.

A *slice* of a sequence is a sequence *of the same type*.

```
>>> S = 'zurich'
>>> S[0:4]
'zuri'
```

## List mutation

You can replace items in a *mutable* sequence by assigning them a new value:

```
>>> L = ['P', 'y', '2']
>>> L[2] = '3'
>>> print(L)
['P', 'y', '3']
```

You can also replace an entire slice of a mutable sequence:

```
>>> L[0:2] = ['1', '2']
>>> print(L)
['1', '2', '3']
```

The new slice does not need to have the same length:

```
>>> L[2:] = range(5)
>>> print(L)
['1', '2', 0, 1, 2, 3, 4]
```

## List mutation

You can replace items in a *mutable* sequence by assigning them a new value:

```
>>> L = ['P', 'y', '2']
>>> L[2] = '3'
>>> print(L)
['P', 'y', '3']
```

You can also replace an entire slice of a mutable sequence:

```
>>> L[0:2] = ['1', '2']
>>> print(L)
['1', '2', '3']
```

The new slice does not need to have the same length:

```
>>> L[2:] = range(5)
>>> print(L)
['1', '2', 0, 1, 2, 3, 4]
```

## List mutation

You can replace items in a *mutable* sequence by assigning them a new value:

```
>>> L = ['P', 'y', '2']
>>> L[2] = '3'
>>> print(L)
['P', 'y', '3']
```

You can also replace an entire slice of a mutable sequence:

```
>>> L[0:2] = ['1', '2']
>>> print(L)
['1', '2', '3']
```

The new slice does not need to have the same length:

```
>>> L[2:] = range(5)
>>> print(L)
['1', '2', 0, 1, 2, 3, 4]
```

## Operating on lists

Python provides a number of methods to modify a list  $L$ :

### **L.append(x)**

Append item  $x$  to list  $L$ .

### **L.insert(n, x)**

Insert item  $x$  at position  $n$  of list  $L$ ; other items are “shited to the right” to make place.

### **L.remove(x)**

Remove first occurence of item having value  $x$  from list  $L$ .

### **L.pop(n)**

Remove item at position  $n$  from list  $L$ .

## Operating on lists, II

### **L.index(x)**

Return position of first item in  $L$  having value  $x$ .

### **L.count(x)**

Return number of items in  $L$  having value  $x$ .

### **L.extend(K)**

Graft a list  $K$  to the end of list  $L$ .

*Reference:* <https://docs.python.org/2/library/stdtypes.html#typeseq>

## Lists operators

You can concatenate two lists using the + operator:

```
>>> [1, 2] + [3, 4]
[1, 2, 3, 4]
```

You can mutate a list *in place* with the += operator:

```
>>> L = [1, 2]
>>> L += [3, 4]
>>> print(L)
[1, 2, 3, 4]
```

The \* operator also works on lists:

```
>>> L = [1, 2]
>>> print(L*3)
[1, 2, 1, 2, 1, 2 ]
```



# for-loops

## for-loops

With the `for` statement, you can **loop over the items of a sequence**:

```
for i in range(0, 4):  
    # loop block  
    print (i*i)
```

To break out of a `for` loop, use the `break` statement.

To jump to the next iteration of a `for` loop, use the `continue` statement.

The `for` statement can be used to loop over elements in *any sequence*.

```
>>> for val in [1,2,3]:
```

```
...     print(val)
```

```
1
```

```
2
```

```
3
```

Loop over lists

The `for` statement can be used to loop over elements in *any sequence*.

```
>>> for val in 'abc':  
...     print(val)  
'a'  
'b'  
'c'
```

Loop over strings

If you want to loop over a *sorted* sequence you can use the function `sorted()` :

```
>>> for val in sorted([1,3,2]):  
...     print(val)  
1  
2  
3
```

and to loop over a sequence in *inverted* order you can use the `reversed()` function:

```
>>> for val in reversed('abc'):  
...     print(val)  
'c'  
'b'  
'a'
```

**Exercise 3.B:** Write a function `odd` that takes a list of integers and returns a list of all the odd ones.

**Exercise 3.C:** Write a function `deviation(L, m)` that takes a list `L` of numbers and a single value `m` returns a list with the difference of `m` and each element `x` of `L`.

## map, reduce, filter (1)

Constructing a new list by looping over a given list and applying a function on all elements is so common that there are specialized functions for that:

### **map(fn, L)**

Return a new list formed by applying function  $f_n(x)$  to every element  $x$  of list  $L$

### **filter(fn, L)**

Return a new list formed by elements  $x$  of list  $L$  for which  $f_n(x)$  evaluates to a “True” value.

## map, reduce, filter (2)

### **reduce(fn2, L)**

Apply function `fn2(x, y)` to the first two items `x` and `y` of list `L`, then apply `fn2` to the result and the third element of `L`, and so on until all elements have been processed — return the final result.

See also: <http://www.python-course.eu/lambda.php> and <https://docs.python.org/3/howto/functional.html> (more advanced)



This is how you could rewrite Exercises 7 and 8 using `map` and `filter`.

```
# *** Exercise 3.B ***
```

```
def is_odd(x):  
    return (x % 2 == 1)  
  
def odd(L):  
    return filter(is_odd, L)
```

```
# *** Exercise 3.C ***
```

```
def deviation(L, m):  
    # note: can define  
    # func's in func's!  
    def delta(x):  
        return abs(x-m)  
    return map(delta, L)
```

## Other containers

The following builtin containers are always available:

**dict** *mutable* key/value mapping.

**set** *mutable*, unordered set of *unique* elements.

**frozenset** *immutable*, unordered set of *unique* elements.

Other specialized containers are available in the `collections` module:

**deque** a generalization of stacks and queues

**namedtuple** similar to a tuple, but allows you to access the elements *by name*

**OrderedDict** dictionary that remembers the order that the items were inserted.

## Other containers

The following builtin containers are always available:

**dict** *mutable* key/value mapping.

**set** *mutable*, unordered set of *unique* elements.

**frozenset** *immutable*, unordered set of *unique* elements.

Other specialized containers are available in the `collections` module:

**deque** a generalization of stacks and queues

**namedtuple** similar to a tuple, but allows you to access the elements *by name*

**OrderedDict** dictionary that remembers the order that the items were inserted.

# Plotting basics

## Plotting

- Plotting is a key part of data analysis
- It helps you to visualise data and see patterns
- It can help you to identify outliers and trends
- It can help you to communicate your findings
- There are many different types of plots
- You should choose the right plot for your data

Plotting basics

## Plotting libraries

**Matplotlib** is the most-used plotting library in the Python community: it provides a large array of (mostly low level) facilities for making plots, and a more high-level interface largely inspired by MATLAB plotting system.

**Seaborn** is an add-on library that provides:

- ▶ better default visual styles
- ▶ easier plotting functions for many commonly-used types of plots

## Enabling plotting in code

To use Matplotlib and Seaborn in a Jupyter notebook to *embed* graphics in the notebook, run this code in a cell:

```
%matplotlib inline  
  
import matplotlib.pyplot as plt  
import seaborn as sea
```

The same code (minus the `%matplotlib inline` “magic”) can be used in any Python script. By default, graphics will appear in a separate pop-up window.

## Line plots, I

The `plt.plot(x, y)` function can be used to make a 2D line plot.

Arguments  $x$  and  $y$  are sequences: corresponding items in the two sequences give the 2D coordinates of points in the plot.

Note that:

- ▶  $x$  must be *sorted*!
- ▶  $x$  and  $y$  must have the same length.

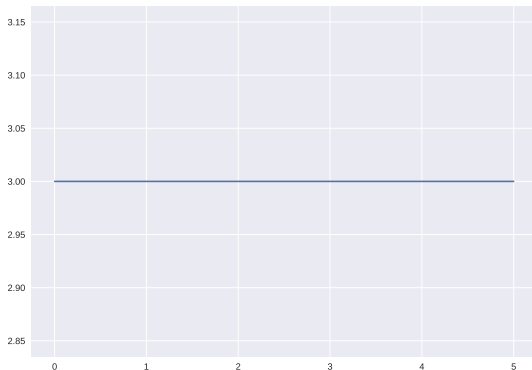
## Line plots, II

```
In [1]: x = [0, 1, 2, 3, 4, 5]
```

```
In [2]: y = [3, 3, 3, 3, 3, 3]
```

```
In [3]: plt.plot(x, y)
```

```
Out [3]: [<matplotlib.lines.Line2D at 0x7fe8a3454750>]
```





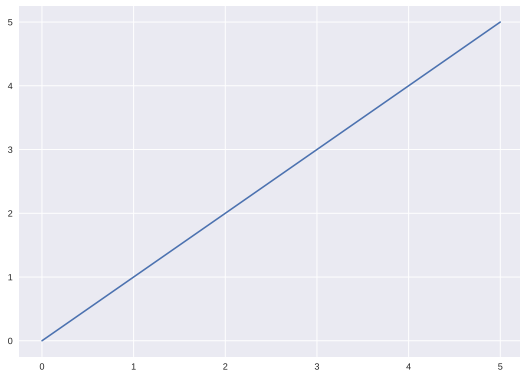
## Line plots, III

```
In [4]: x = [0, 1, 2, 3, 4, 5]
```

```
In [5]: y = x
```

```
In [6]: plt.plot(x, y)
```

```
Out[6]: [<matplotlib.lines.Line2D at 0x7fe8a3454750>]
```



## Line plots, IV

Plotting different series of data in the same figure requires a bit more work.

```
fig, ax = plt.subplots(1, 1, figsize=[10, 7])
```

```
# common x-axis items
```

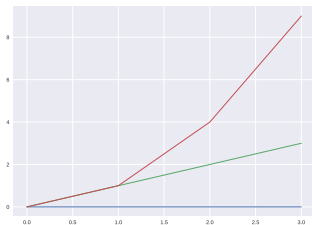
```
x = [0, 1, 2, 3]
```

```
# three lines
```

```
ax.plot(x, [0, 0, 0, 0])
```

```
ax.plot(x, [0, 1, 2, 3])
```

```
ax.plot(x, [0, 1, 4, 9])
```



```
# save to file
```

```
fig.savefig('fig/lineplot2.pdf')
```

## Line plots, V

Plotting different series of data in the same figure requires a bit more work.

1. First use the `plt.subplots` function to create *figure* and an *axes* object
2. An *axes object* is a “frame” for a single plot – use methods `.plot()` to lay a graph onto the canvas. Each invocation of `.plot()` *adds* a plot onto the canvas.
3. The *figure object* contains all the axes can be used for saving the final output with `.savefig()`

### Exercise 3.D (Homework):

Write a function `plotfn(xs, f)` that takes two arguments:

- ▶ a sequence of numbers `xs`, and
- ▶ a function `f`, which takes one single argument (a number) and returns a number. Function `plotfn()` should display a line plot of the mathematical function `f` over the set of numbers `xs`.

**Bonus points:** Change the `plotfn()` function so to take an additional argument (a file name) and save the figure into that file.

**More bonus points:** Change the `plotfn` function so to take a *list* of mathematical functions `fs` and plot all of them.

## Scatter plots

Use the `plt.scatter(x, y)` function.

Everything else works as in line plots.

## Bar plots

Use Seaborn's `sea.barplot(x, y)` function.

Everything else works *almost* as in line plots; when you need to plot onto an axis (the `ax` object of previous examples), then you need to pass the axis as an additional parameter:

```
sea.barplot(x, y, ax=ax)
```

# Appendix

## Sets (1)

The `set` type implements an **unordered** container that holds exactly one object per equivalence class:

```
>>> S = set()
>>> S.add(1)
>>> S.add('two')
>>> S.add(1)
>>> S
set([1, 'two'])
```



## Sets (2)

You can create a set and add elements to it in one go:

```
>>> S2 = set ([1, 2, 3, 4])
```

and remove elements:

```
>>> S2.remove(2)
```

```
>>> S2.pop()
```

```
1
```

```
>>> S2
```

```
set ([3, 4])
```

## Sets (3)

Sets are often used to get unique values from a list:

```
>>> L = [1, 1, 2, 2, 3, 3]
>>> set(L)
set([1, 2, 3])
```

Of course, you can also create a list from a set:

```
>>> S = set((1, 2, 3))
>>> list(S)
[1, 2, 3]
```

**Q:** *In what order will the set items appear in the resulting list?*

## Sets (3)

Sets are often used to get unique values from a list:

```
>>> L = [1, 1, 2, 2, 3, 3]
>>> set(L)
set([1, 2, 3])
```

Of course, you can also create a list from a set:

```
>>> S = set((1, 2, 3))
>>> list(S)
[1, 2, 3]
```

*Q: In what order will the set items appear in the resulting list?*

## Sets (3)

Sets are often used to get unique values from a list:

```
>>> L = [1, 1, 2, 2, 3, 3]
>>> set(L)
set([1, 2, 3])
```

Of course, you can also create a list from a set:

```
>>> S = set((1, 2, 3))
>>> list(S)
[1, 2, 3]
```

**Q:** *In what order will the set items appear in the resulting list?*

# Tuples

Tuples are like lists

```
>>> T = (1, 2, 3)
>>> T[0]
1
>>> T[0:1]
(1,)
```

but they are *immutable*

```
>>> T[0] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

## Multiple assignment

You can assign multiple variables at the same time

```
>>> a, b, c = (1, 2, 3)
```

```
>>> print(a)
```

```
1
```

```
>>> print(b)
```

```
2
```

It works with any sequence:

```
>>> a, b, c = 'UZH'
```

```
>>> print(a)
```

```
U
```

*Q: Can you think of a way to swap the values of two variables using this?*

## Multiple assignment

You can assign multiple variables at the same time

```
>>> a, b, c = (1, 2, 3)
```

```
>>> print(a)
```

```
1
```

```
>>> print(b)
```

```
2
```

It works with any sequence:

```
>>> a, b, c = 'UZH'
```

```
>>> print(a)
```

```
U
```

**Q:** *Can you think of a way to swap the values of two variables using this?*

## Multiple assignment

You can assign multiple variables at the same time

```
>>> a, b, c = (1, 2, 3)
```

```
>>> print(a)
```

```
1
```

```
>>> print(b)
```

```
2
```

It works with any sequence:

```
>>> a, b, c = 'UZH'
```

```
>>> print(a)
```

```
U
```

**Q:** *Can you think of a way to swap the values of two variables using this?*

```
>>> a, b = b, a
```



## Multiple assignment (2)

Multiple assignment can be used in `for` statements as well.

```
>>> L = [(1, 'a'), (2, 'b'), (3, 'c')]
>>> for x, y in L:
...     print ("first is " + str(x)
...           + " and second is " + y)
```

This is particularly useful with functions that return a tuple. For instance the `enumerate()` function (look it up with `help()`!).

# Data structures recap

<b>mutable</b>	<b>immutable</b>	
set	frozenset	unordered container of unique elements
list	tuple	ordered sequence
dict	—	key/values mapping
—	str	ordered sequence of characters