Basics II - Data Structures

Gabriele Pompa
gabriele.pompa@unisi.com

March 23, 2020

Contents

Resources:

- *Python for Finance (2nd ed.)*: Sec. 3.Basic Data Structures (Section 3.Excursus: Functional Programming is optional)
- *The Python Tutorial*: Sec. 3.1.3 (Lists), 4.2 (for Statements), 4.3 (The `range()` Function), 4.4 (break and continue Statemenents, and else Clauses on Loops), 5.1 (More on Lists), 5.3 (Tuples and Sequences), 5.4 (Sets), 5.5 (Dictionaries)

Executive Summary

Intuitively, a *data structure* is an object containing other objects, not necessarily of the same *data type*.

Standard Python provides four basic data structures, which can be differentiated at high level by being: - *ordered* or *not ordered:* that is, whether they preserve the order in which entries are added or not; - *mutable* or *immutable:* that is, whether - once defined - they can be modified or not.

These data-strucutures are:

data-structure	ordered (or not)	mutable (or not)
Tuples	ordered	immutable
Lists	ordered	mutable
Dicts	not ordered	mutable
Sets	not ordered	mutable

The function `type()` can be called over any defined data-structure and returns its type: `tuple` for Tuples, `list` for Lists, `dict` for Dicts and `set` for Sets.

The following sections are organized as follows:

- In Section 1 Tuples (`tuple`) are introduced as the Python data-structure for *ordered* sequence-like objects that *cannot be* modified once defined.
- In Section 2 Lists (`list`) are introduced as the Python data-structure for *ordered* sequence-like objects that *can be* modified once defined. In this context `for` loops are introduced in Sec. Section 2.7.
- In Section 3 Dicts (`dict`) are introduced as the Python data-structure for *not ordered* collection-like objects that *can be* modified once defined and that implement a *key-to-value* map.
- In Section 4 Sets (`set`) are introduced as the Python data-structure for *not ordered* collection-like objects that *can be* modified once defined and that contain unique elements (that is, every elements appears only once).

1 Tuples

Tuples consists of a number of values - of heterogeneous data-type, in general - packed together in an immutable sequence and separated by commas.

In my experience, I didn't use tuple that often, probably because their *immutability* goes against the dynamism of trail-n-error phases of a typical quantitative analysis. In fact, for the same reason, tuples may be a good asset as they guarantee the safety of data stored in them.

1.1 Definition

Tuples can be defined with or without parenthesis `()` surrounding the `,`-separated sequence.

[1]:
```
tup = (1, 0.35, "GBP")

print(tup)
type(tup)
```

[2]:
```
tup = 1, 0.35, "GBP"

print(tup)
type(tup)
```

The number of elements is easily retrieved by the `len()` function:

[3]:
```
len(tup)
```

[3]: 3

1.2 Indexing and Slicing

Tuples share a lot of properties with other sequence-like data-structure. For details take a look at Sequence Types — list, tuple, range page of the Python standard library.

In particular, tuples share indexing features with strings (see Basics_I____Data_Types.ipynb) and lists (see Sec. 2). In particular, elements of a tuple can be accessed by *zero-based* indexes:

[4]:
```
# 0 is the index of the first element of the tuple
print(tup[0])
type(tup[0])
```

```
1
```

[4]: int

[5]:
```
# -1 is the index of the last element of the tuple
print(tup[-1])
type(tup[-1])

tup[len(tup)-1]
```

```
GBP
```

[5]: 'GBP'

and tuples can be sliced. That is, you can select few elements only of the tuple if you want to

[6]:
```
tup_slice = tup[0:2] # elements from position 0 (included) to 2 (excluded)

print(tup_slice)
type(tup_slice)
```

```
(1, 0.35)
```

[6]:
```
tuple
```

[7]:
```
tup[2:5] # elements from position 2 (included) to 5 (excluded)
```

[7]:
```
('GBP',)
```

[8]:
```
tup[:2]   # elements from the beginning to position 2 (excluded) --- equivalent␣
↪to tup[0:2]
```

[8]:
```
(1, 0.35)
```

[9]:
```
tup[-2:]  # elements from the second-last (included) to the end
```

[9]:
```
(0.35, 'GBP')
```

1.3 Changing Values (and why you shouldn't do that)

Analogously to strings - but differently from lists - tuples are *immutable* objects. That is, if you try to change one of its elements, you get

```
TypeError: 'tuple' object does not support item assignment
```

[10]:
```
# p[0] = 17
```

In particular, you cannot simply use the + operator as you would do with a string to concatenate characters. That is, something like

```
17 + tup[1:]
```

would cause the following error

```
TypeError: unsupported operand type(s) for +: 'int' and 'tuple'
```

that simply tells you that you cannot *add* `int` objects (like `17`) with `tuple` objects (like the slice `tup[1:]`).

[11]:
```
# 17 + tup[1:]
```

Workaround to modify a tuple: of course there is a workaround (and it will be clear once you have covered Sec. 2 on Lists). You can: - convert the tuple into a list using `list()` casting function, - change the list (which is a mutable object) - convert the list back into a tuple using the `tuple()` casting function.

[12]:
```
list_tup = list(tup) # cast tup as a list

print(list_tup)
type(list_tup)
```

```
[1, 0.35, 'GBP']
```

[12]:
```
list
```

[13]:
```
list_tup[0] = 17 # change the element
```

[14]:
```
tup = tuple(list_tup) # cast-back as a tuple

print(tup)
type(tup)
```

```
(17, 0.35, 'GBP')
```

[14]:
```
tuple
```

Nevertheless, if you need to modify a tuple, the real question is why did you pack your data into a tuple? So, the take-home message is: if you need to modify a tuple, re-think your code… you're likely to need a list instead of a tuple.

1.4 Nested Tuples

Read this section once you have covered Section 2 on Lists

Notice that even if the tuple itself is not mutable, its elements can consist of *mutable* objects (such as lists) and/or *immutable* objects (such as tuple themselves).

[15]:
```
l = [87, 100, 99]          # a list
t = ("ACT/365", "ACT/360") # a tuple

nested_tup = (l, t, 100)

print(nested_tup)
type(nested_tup)
```

```
([87, 100, 99], ('ACT/365', 'ACT/360'), 100)
```

[15]:
```
tuple
```

As we have seen, elements of `nested_tup` can be accessed through indexing:

[16]:
```
print(nested_tup[0])
type(nested_tup[0])
```

```
[87, 100, 99]
```

[16]: list

[17]:
```
print(nested_tup[1])
type(nested_tup[1])
```

```
('ACT/365', 'ACT/360')
```

[17]: tuple

[18]:
```
print(nested_tup[2])
type(nested_tup[2])
```

```
100
```

[18]: int

In the same way as we have seen in Section 2.3 for nested lists, you can as well access nested elements of list `l` and tuple `s` using nested-indexing of `nested_tup`:

[19]:
```
# [0][0] is the index of the first element
# of (list 'l' which is) the first element of the tuple 'nested_tup'
print(nested_tup[0][0])
type(nested_tup[0][0])
```

```
87
```

[19]: int

[20]:
```
# [0][2] is the index of the third element
# of (list 'l' which is) the first element of the tuple 'nested_tup'
print(nested_tup[0][2])
type(nested_tup[0][2])
```

```
99
```

[20]: int

[21]:
```
# [1][0] is the index of the first element
# of (tuple 't' which is) the second element of the tuple 'nested_tup'
print(nested_tup[1][0])
type(nested_tup[1][0])
```

```
ACT/365
```

[21]: str

[22]:
```
# [1][1] is the index of the second element
# of (tuple 't' which is) the second element of the tuple 'nested_tup'
print(nested_tup[1][1])
```

```python
type(nested_tup[1][1])
```

ACT/360

[22]: str

Warning: in the same way as they apply to nested-indexing of lists, indexing of tuples may raise (*repetita iuvant*):

- *out of range* `IndexError`: if you try to refer to an index that does not correspond to any element of the data structure (or of its nested data-structures, if any)

[23]:
```python
# produces: IndexError: tuple index out of range
# because index 3 would refer to the 4th element of nested_tup, that does not
↪exist.

# nested_tup[3]
```

[24]:
```python
# produces: IndexError: list index out of range
# because index 3 would refer to the 4th element of nested_tup[0] (i.e. list
↪'l'), that does not exist

# nested_tup[0][3]
```

[25]:
```python
# produces: IndexError: tuple index out of range
# because index 2 would refer to the 3rd element of nested_tup[1] (i.e. tuple
↪'t'), that does not exist

# nested_tup[1][2]
```

- *object is not subscriptable* `TypeError`: if you try to refer with an index to an element that is not indexable (like Integers, Floats,…)

[26]:
```python
nested_tup[2]
```

[26]: 100

[27]:
```python
# produces: TypeError: 'int' object is not subscriptable
# because we are trying to refer to the first element of nested_tup[2] (i.e.
↪integer 100),
# that, in poor words, does not have any element inside and thus doesn't admit
↪indexing.

# nested_tup[2][0]
```

Keeping in mind that if you need to modify a tuple, you're likely to have chosen the wrong data structure to pack together your data (why not opting for a `list` in the first place?), still for completeness let's briefly discuss what you can modify in a nested tuple.

Getting back to our nested tuple, you can modify only its mutable nested elements (if any):

[28]:
```
nested_tup
```

[28]:
```
([87, 100, 99], ('ACT/365', 'ACT/360'), 100)
```

[29]:
```
nested_tup[0][1] = 98
nested_tup
```

[29]:
```
([87, 98, 99], ('ACT/365', 'ACT/360'), 100)
```

[30]:
```
nested_tup[0].append(75)  # integer 75 is appended at the end of list [87, 98, 
 ↪99]
nested_tup
```

[30]:
```
([87, 98, 99, 75], ('ACT/365', 'ACT/360'), 100)
```

but you cannot explicitly re-define elements of the tuple `nested_tup` because this would constrast with the immutability of the (nested) tuple itself.

To make an example: an explicit redefinition of the element `nested_tup[0]` of `nested_tup` like this one

```
nested_tup[0] = [67, 89]
```

would produce

```
TypeError: 'tuple' object does not support item assignment
```

If you find strange that you can change values of the tuple's element `nested_tup[0]` or even adding values to it, as above, but you cannot re-define it as a whole... well, I'm with you. Let's go ahead, you can live with this :)

2 Lists

Lists consists of a number of values - in general, of heterogeneous data-type - packed together in a mutable sequence and separated by commas between square brackets.

Lists are very versatile data structures, since they offer flexibility (since they are mutable) and feature several built-in methods that can speed up coding.

2.1 Definition

Lists are defined with square brackets `[]` surrounding the `,`-separated sequence.

[31]:
```
lis = [1, 0.35, "GBP"]

print(lis)
type(lis)
```

```
[1, 0.35, 'GBP']
```

[31]: list

The number of elements is easily retrieved by the `len()` function:

[32]:
```
len(lis)
```

[32]: 3

2.2 Indexing and Slicing

Lists share a lot of properties with other sequence-like data-structures. In particular, they share *zero-based* indexing and slicing with Strings and Tuples.

[33]:
```
# 0 is the index of the first element of the list
print(lis[0])
type(lis[0])
```

```
1
```

[33]: int

[34]:
```
# -1 is the index of the last element of the list
print(lis[-1])
type(lis[-1])
```

```
GBP
```

[34]: str

Here is how to slice a list (yes, always the same way):

[35]:
```
lis_slice = lis[0:2] # elements from position 0 (included) to 2 (excluded)

print(lis_slice)
type(lis_slice)
```

```
[1, 0.35]
```

[35]: list

[36]:
```
lis[2:5] # elements from position 2 (included) to 5 (excluded)
```

[36]: ['GBP']

[37]:
```
lis[:2]   # elements from the beginning to position 2 (excluded) --- equivalent␣
↪to lis[0:2]
```

[37]: [1, 0.35]

[38]:
```
lis[-2:]  # elements from the second-last (included) to the end
```

[38]: `[0.35, 'GBP']`

2.3 Nested Lists

Lists can nest other data structures, both *mutable* objects (such as other lists) and/or *immutable* objects (such as tuples).

[39]:
```
l = [87, 100, 99]           # a list
t = ("ACT/365", "ACT/360") # a tuple

nested_lis = [l, t, 100]

print(nested_lis)
type(nested_lis)
```

```
[[87, 100, 99], ('ACT/365', 'ACT/360'), 100]
```

[39]: `list`

As we have seen, elements of `nested_lis` can be accessed through indexing:

[40]:
```
print(nested_lis[0])
type(nested_lis[0])
```

```
[87, 100, 99]
```

[40]: `list`

[41]:
```
print(nested_lis[1])
type(nested_lis[1])
```

```
('ACT/365', 'ACT/360')
```

[41]: `tuple`

[42]:
```
print(nested_lis[2])
type(nested_lis[2])
```

```
100
```

[42]: `int`

You can as well access elements of list `l` and tuple `s` using a nested-indexing of `nested_lis`:

[43]:
```
nested_lis
```

[43]: `[[87, 100, 99], ('ACT/365', 'ACT/360'), 100]`

[44]:
```
# [0][0] is the index of the first element
# of (list 'l' which is) the first element of the list 'nested_lis'
print(nested_lis[0][0])
type(nested_lis[0][0])
```

```
87
```

[44]:
```
int
```

[45]:
```
# [0][2] is the index of the third element
# of (list 'l' which is) the first element of the list 'nested_lis'
print(nested_lis[0][2])
type(nested_lis[0][2])
```

```
99
```

[45]:
```
int
```

[46]:
```
# [1][0] is the index of the first element
# of (list 't' which is) the second element of the list 'nested_lis'
print(nested_lis[1][0])
type(nested_lis[1][0])
```

```
ACT/365
```

[46]:
```
str
```

[47]:
```
# [1][1] is the index of the second element
# of (tuple 't' which is) the second element of the list 'nested_lis'
print(nested_lis[1][1])
type(nested_lis[1][1])
```

```
ACT/360
```

[47]:
```
str
```

2.4 Indexing of Nested Sequences

Ok you have understood how it works… This is actually a general rule, that applies to all the sequence-like data structures: Tuples (`tuple`), Lists (`list`) but also Numpy arrays (`numpy.ndarray`, with the slightly changed syntax `[i,j]` instead of `[i][j]`, as we'll show in a future lesson) that will be introduced in a future notebook.

If a sequence-like data structure, say `seq`, has nested sequence-like elements, then

`seq[i][j]`

is the element of index `j` of the element of index `i`, `seq[i]`, of `seq`. That is, `seq[i][j]` is the $(j+1)$-th element of the $(i+1)$-th element, `seq[i]`, of `seq`.

Warning: nested-indexing of lists may raise:

- *out of range* `IndexError`: if you try to refer to an index that does not correspond to any element of the data structure (or of its nested data-structures, if any)

[48]:
```
nested_lis
```

[48]:
```
[[87, 100, 99], ('ACT/365', 'ACT/360'), 100]
```

[49]:
```
# produces: IndexError: list index out of range
# because index 3 would refer to the 4th element of nested_lis, that does not␣
↪exist.

nested_lis[len(nested_lis)-1]
```

[49]:
```
100
```

[50]:
```
# produces: IndexError: list index out of range
# because index 3 would refer to the 4th element of nested_lis[0] (i.e. list␣
↪'l'), that does not exist

# nested_lis[0][3]
```

[51]:
```
# produces: IndexError: tuple index out of range
# because index 2 would refer to the 3rd element of nested_lis[1] (i.e. tuple␣
↪'t'), that does not exist

nested_lis[1][-1]
```

[51]:
```
'ACT/360'
```

- *object is not subscriptable* `TypeError`: if you try to refer with an index to an element that is not indexable (like Integers, Floats,…)

[52]:
```
nested_lis[2]
```

[52]:
```
100
```

[53]:
```
# produces: TypeError: 'int' object is not subscriptable
# because we are trying to refer to the first element of nested_lis[2] (i.e.␣
↪integer 100),
# that, in poor words, does not have any element inside and thus doesn't admit␣
↪indexing.

# nested_lis[2][0]
```

2.5 Changing Values

Differently from strings and tuples, lists are *mutable* objects and their elements can be changed in a straightforward way.

[54]:
```python
lis
```

[54]:
```
[1, 0.35, 'GBP']
```

[55]:
```python
lis[0] = 17
lis
```

[55]:
```
[17, 0.35, 'GBP']
```

Being a mutable object, you can modify both mutable and immutable (tuples) objects of a nested list, but of course you cannot change elements inside a nested immutable object (e.g. a tuple inside the list).

Let's go back to our `nested_lis` and suppose you want to change the first day count convention from "ACT/365" to "ACT/360"

[56]:
```python
nested_lis
```

[56]:
```
[[87, 100, 99], ('ACT/365', 'ACT/360'), 100]
```

you cannot explicitly modify the element "ACT/365" of the tuple

```python
("ACT/365", "ACT/360")
```

because this would result in a

```
TypeError: 'tuple' object does not support item assignment
```

[57]:
```python
# produced TypeError

# nested_lis[1][0] = "ACT/360"
```

but you could directly re-define the whole tuple `nested_lis[1]` as it is an element of the `nested_lis` list, which is a mutable object.

[58]:
```python
nested_lis[1] = 0.345
nested_lis
```

[58]:
```
[[87, 100, 99], 0.345, 100]
```

Other - mutable - elements can be modified as you want:

[59]:
```python
nested_lis[0][1] = 98
nested_lis
```

[59]:
```
[[87, 98, 99], 0.345, 100]
```

[60]:
```
nested_lis[0].append(75)  # integer 75 is appended at the end of list [87, 98,␣
↪99]
nested_lis
```

[60]: [[87, 98, 99, 75], 0.345, 100]

[61]:
```
nested_lis[2] -= 10 # x -= 1 is a short-cut for x = x-1. Other are +=, *= and /=
nested_lis
```

[61]: [[87, 98, 99, 75], 0.345, 90]

2.6 Built-in methods

For details on built-in methods see 5.1. More on Lists of the Python tutorial. In particular, two particularly useful built-in methods are worth of mention: - `list.append(x)`: which appends element `x` to the end of the list, extending it

[62]:
```
lis
```

[62]: [17, 0.35, 'GBP']

[63]:
```
lis.append('EUR')
lis
```

[63]: [17, 0.35, 'GBP', 'EUR']

- `list.sort()`: that sorts in ascending order a list.

Notice that the list to be sorted must have elements of homogenous data-type, otherwise the interpreter will complain, as in this case:

```
TypeError: '<' not supported between instances of 'str' and 'float'
```

[64]:
```
# lis.sort()
```

[65]:
```
lis[:2]
```

[65]: [17, 0.35]

but, for our list `lis`, the sorting will work if we define a new string as the `[17, 0.35]` slice of the original one

[66]:
```
lis_slice = lis[:2]  # [17, 0.35] slice
lis_slice.sort()
lis_slice
```

[66]: [0.35, 17]

[67]:
```
lis_slice2 = lis[-2:]
lis_slice2
```

[67]: ['GBP', 'EUR']

[68]:
```
lis_slice2.sort()
lis_slice2
```

[68]: ['EUR', 'GBP']

2.7 for loop

A for loop in Python is declared as follows:

```
for variable in sequence:
    statement(s) possibly using variable
```

A `sequence` can be a Python `list`, a Numpy `numpy.ndarray` or other sequence-like data structures. The Python interpreter *loops* from one element to the next one of the `sequence` assigning each time that element to the `variable`. With this value of `variable`, the `statement(s)` are executed. The loop ends when all the elements of the sequence have been considered.

2.7.1 for loop over a list

To make an example, we can print elements of a list:

[69]:
```
x = [10, 20, 30]

for xi in x:
    print(xi)
```

```
10
20
30
```

Notice how this `for` loop is not *counter-based*, that is is the Python interpreter that loops into the sequence, returning us the current element of the sequence at each iteration.

2.7.2 Counter-based looping and range() function

In some occasions it could be good to loop over a sequence being able to access to its elements through their indexes. This can be achieved using the `range()` function which generates a sequence of numbers as an object of the (strange) type `range`.

[70]:
```
x  = range(10)

print(x)
type(x)
```

```
range(0, 10)
```

[70]: range

`range()` is mostly used in `for` loops as follows:

[71]:
```
# a loop over the first 5 numbers from zero
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

[72]:
```
# a loop over numbers from 1 to 9
for i in range(1,10):
    print(i)
```

```
1
2
3
4
5
6
7
8
9
```

But it can be used also to loop over a list, accessing its indexes:

[73]:
```
# a loop over the elements of list `lis`
for i in range(len(x)):
    print(x[i])
```

```
0
1
2
3
4
5
6
7
8
9
```

[74]:
```
x = [10, 20, 30]

for i in range(3):
    print(i)
```

```
    print(x[i])
```

```
0
10
1
20
2
30
```

where we have used the fact that `range(len(x))` returns numbers from `0` to `len(x)-1`, which are first and last indexes of list `x`.

Example: let's get back to our Fibonacci numbers (see Sec. 3.1 while loop of Basics_I____Data_Types.ipynb for a refresh) and let's suppose we want to compute the n-th number F_n

[75]:
```
def fib_nth(n):
    """
    This function computes the n-th Fibonacci number using inline assignments.

    Parameters:
        n (int): which number to compute.

    Returns:
        F_n2 (int): n-th Fibonacci number.
    """

    if n <= 2:
        return n-1

    # inline initialization
    F_n2, F_n1 = 0, 1 # F_{n-2}, F_{n-1}


    for k in range(n-1):

        # uncomment this line below if you want to print to screen the current␣
↪number
        print(F_n2)

        # inline update of the last two numbers
        F_n2, F_n1 = F_n1, F_n1 + F_n2

    return F_n2
```

[76]:
```
N = 10
```

```
fib_nth(N)
```

```
0
1
1
2
3
5
8
13
21
```

[76]: 34

2.7.3 break Statement in loops

The break statement interrupts a while or for loop that wouldn't be concluded yet otherwise. It is typically used in combination with an if statement to break the loop once the condition triggered by the if is met.

To make a practical example, let's check whether an item is in a list looping over the list.

[77]:
```python
lis = [1, "A", 0.35, 1/4, "@"]

item = 0.35

for element in lis:

    print("Checking element {}".format(element))

    if element == item:
        print("Item {} found".format(item))
        break
```

```
Checking element 1
Checking element A
Checking element 0.35
Item 0.35 found
```

As you can see, elements `1/4` and `@` are not checked, because the `element == item` condition of the `if` statement is triggered for element `0.35`.

Just for your knowledge, this was just a pedantic example. If you really want to check whether an item is in a list, well it's super-easy: you have to use the general syntax: 'python object in sequence where is the `in` operator that manages the checking operations and returns `True` or `False` depending on the fact that the `object` is really found in the `sequence` or not. Therefore `in` is typically used in the `condition` part of an `if` statement.

[78]:
```
if (item in lis):
    print("Item {} found".format(item))
else:
    print("Item {} not found".format(item))
```

```
Item 0.35 found
```

2.7.4. `enumerate()` looping There is one more way to loop over a list, which allows us to access both indexes and values of a sequence. It's the built-in function ernumerate().

[79]:
```
x = [10, 20, 30]

for i, xi in enumerate(x):
    print("index i={}; value xi={}".format(i, xi))  # xi is equivalent to do␣
 ↪x[i]
```

```
index i=0; value xi=10
index i=1; value xi=20
index i=2; value xi=30
```

2.8 8. List comprehension

Let's suppose we want to create a list of the squares of the first `n` numbers: 0, 1, 4, 9,…. We could do this way with a `for` loop

[80]:
```
def create_squares_list(n):

    lis = []  # an empty list

    for i in range(n):
        print(lis)
        lis += [i**2]  # equivalent to lis.append(i**2)

    return lis
```

[81]:
```
n = 10
```

[82]:
```
x = create_squares_list(n)
x
```

```
[]
[0]
[0, 1]
[0, 1, 4]
[0, 1, 4, 9]
[0, 1, 4, 9, 16]
[0, 1, 4, 9, 16, 25]
[0, 1, 4, 9, 16, 25, 36]
```

```
[0, 1, 4, 9, 16, 25, 36, 49]
[0, 1, 4, 9, 16, 25, 36, 49, 64]
```

[82]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Timing the code (let's re-define `create_squares_list(n)` function without the print, which would consumes a lot of computing time otherwise)

[83]:
```python
def create_squares_list(n):

    lis = []  # an empty list

    for i in range(n):
        lis += [i**2]  # equivalent to lis.append(i**2)

    return lis
```

[84]:
```python
%timeit create_squares_list(n)
```

```
3.46 µs ± 40.9 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Alternatively we can create a list through a list comprehension feature (see sectio 5.1.3. List Comprehensions), which is an elegant and fast way to generate lists:

[85]:
```python
x = [i**2 for i in range(n)]
x
```

[85]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

[86]:
```python
y = [pippo**3 for pippo in range(n)]
y
```

[86]:
```
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

[87]:
```python
9**3
```

[87]:
```
729
```

as you can see, the result is the same. For small dimensions like `n=10`, also the time is comparable

[88]:
```python
%timeit [i**2 for i in range(n)]
```

```
2.98 µs ± 38.4 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

but let's see what happen for bigger dimensions:

[89]:
```python
n = int(1e6)
n
```

[89]:
```
1000000
```

[90]:
```
%timeit create_squares_list(n)
```

```
368 ms ± 3.51 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[91]:
```
%timeit [i**2 for i in range(n)]
```

```
334 ms ± 30.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

For such an easy definition (we are only computing squares) the improvement is marginal, but for more complex list definitions, the speed-improvement can be significant. Plus, what the entire `create_squares_list(n)` does has been replaced by the one-line of code `[i**2 for i in range(n)]`. Nice, isn't it?

3 Dicts

Dicts consists of collection of *key: value* pairs where *keys* are a *unique* set of any immutable data-type.

Dictionaries are very good to implement mapping-tables or any kind of logic association between a given set of unique indexers (the keys) and data (the values).

3.1 Definition

Dicts are defined with curly brackets `{}` surrounding the `,`-separated list of `key: value` pairs.

[92]:
```
d = {"AAA": 5, "A+": 20, "BBB": 50, "D": 100}

d
```

[92]:
```
{'AAA': 5, 'A+': 20, 'BBB': 50, 'D': 100}
```

So, differently from lists, that are indexed by a range of numbers, dictionaries are indexed by (unique) keys.

Notice that keys must be of immutable data-type: so ok `str`, `int`, `float`... but not `list` (because they can be modified in place)

[93]:
```
d1 = {
    10: 1,
    20: 2,
    30: 3
}

d1
```

[93]:
```
{10: 1, 20: 2, 30: 3}
```

If you try to use lists as keys of a dictionary you get

```
TypeError: unhashable type: 'list'
```

[94]:
```python
# d2 = {
#     ["AAA"]: 5,
#     ['A+']: 20,
#     ['BBB']: 50,
#     ['D']: 100
# }
```

Another difference from lists is that dicts are not ordered (don't have memory of the order in which the list of key: value pairs has been inputed) and, in general, cannot be sorted. In particular, there is not such indexing like `dictName[0]`, `dictName[1]`, or so, simply because there is no guarantee that the first pair that we put in input is actually stored in the first position (whatever *first-position* could mean) and that could be retrieved accordingly.

3.2 key-based indexing

Indexing in dictionaries is implemented through their keys. Once you know the key, you can retrieve the corresponding value as `dictName[key]`. Like this:

[95]:
```python
d
```

[95]:
```
{'AAA': 5, 'A+': 20, 'BBB': 50, 'D': 100}
```

[96]:
```python
d["AAA"]
```

[96]:
```
5
```

Once we have a dict, representing a map key-to-value, we typically want to look whether a key is in the dictionary:

[97]:
```python
key_list = ["AAA", "AA", "B+", "D"]

for key in key_list:
    if key in d:
        print("key {} found".format(key))
    else:
        print("key {} not found".format(key))
```

```
key AAA found
key AA not found
key B+ not found
key D found
```

Notice that if you try to access a dictionary with a key it does not have, you are going to receive a `KeyError` simply stating that that key is not part of the dictionary's keys.

[98]:
```python
d
```

[98]:
```
{'AAA': 5, 'A+': 20, 'BBB': 50, 'D': 100}
```

[99]:
```
# raises KeyError
# d["B+"]
```

3.3 Changing Values

If you want to change a value corresponding to a given key, you just assign to it a new value: `dictName[key] = newValue`

[100]:
```
d["AAA"] = 1
d
```

[100]:
```
{'AAA': 1, 'A+': 20, 'BBB': 50, 'D': 100}
```

3.4 Built-in methods

Some useful methods: - `dict.keys()` returns a sequence of dict keys; - `dict.values()` returns a sequence of dict values; - `dict.items()` returns a sequence of key: value pairs packed as tuples

[101]:
```
d.keys()
```

[101]:
```
dict_keys(['AAA', 'A+', 'BBB', 'D'])
```

[102]:
```
d.values()
```

[102]:
```
dict_values([1, 20, 50, 100])
```

[103]:
```
d.items()
```

[103]:
```
dict_items([('AAA', 1), ('A+', 20), ('BBB', 50), ('D', 100)])
```

3.5 Looping over dicts

Looping can be done in several ways. The basic one is looping over the keys of the dictionary, which can be done in this way:

[104]:
```
for key in d:
    print("key: {} - value: {}".format(key, d[key]))
```

```
key: AAA - value: 1
key: A+ - value: 20
key: BBB - value: 50
key: D - value: 100
```

The interpreter finds out that `d` is a dictionary and understands that it has to look for its keys and loop over them, assigning each tyme to `key` variable.

4 Sets

Sets consists of an unordered collection with no duplicate elements.

Sets are typically used to store unique values and to check whether a value is in there. Plus basic sets operations like union, intersection, etc…

4.1 Definition

Sets are defined with curly brackets `{}` surrounding a `,`-separated list

[105]:
```
my_set = {1, 2, 2, 4,  6, 6}
my_set
```

[105]:
```
{1, 2, 4, 6}
```

Alernatively you can use the key-word `set()`

[106]:
```
my_set = set([1, 2, 2, 4,  6, 6])
my_set
```

[106]:
```
{1, 2, 4, 6}
```

Notice how repeated values are automatically counted only one.

4.2 Test for membership

Checking whether an element is in the set can be done easily:

[107]:
```
element_list = [i for i in range(10)]
element_list
for element in element_list:
    if element in my_set:
        print("element {} found".format(element))
    else:
        print("element {} not found".format(element))
```

```
element 0 not found
element 1 found
element 2 found
element 3 not found
element 4 found
element 5 not found
element 6 found
element 7 not found
element 8 not found
element 9 not found
```

[108]:
```
element_list
```

[108]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

4.3 Set operations

Let's review basic set operations:

[109]:
```
set1 = set([i for i in range(10)])     # yes, you can use list comprehension here
set2 = {i**2 for i in range(10)}       # yes, this is another way to create a␣
 ↪set: "set-comprehension"
```

[110]:
```
set1
```

[110]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

[111]:
```
set2
```

[111]: {0, 1, 4, 9, 16, 25, 36, 49, 64, 81}

4.3.1. .union()

The union of two sets is the set of elements in `set1`, `set2`, or both

[112]:
```
set1.union(set2)
```

[112]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 16, 25, 36, 49, 64, 81}

or alternatively

[113]:
```
set1 | set2
```

[113]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 16, 25, 36, 49, 64, 81}

4.3.2. .intersection()

The intersection of two sets is the set of elements in common between `set1` and `set2`

[114]:
```
set1.intersection(set2)
```

[114]: {0, 1, 4, 9}

or alternatively

[115]:
```
set1 & set2
```

[115]: {0, 1, 4, 9}

4.3.3. .difference()

The difference of two sets is the set of elements in `set1` but not in `set2`

[116]:
```
set1.difference(set2) # notice that is different from set2.difference(set1)
```

[116]: {2, 3, 5, 6, 7, 8}

or alternatively

[117]:
```
set1 - set2
```

[117]: {2, 3, 5, 6, 7, 8}

4.4 Getting rid of duplicates from a list

To conclude, often sets are used to get rid of duplicates in a list:

[118]:
```
lis = [(-1)**n for n in range(10)]
lis
```

[118]: [1, -1, 1, -1, 1, -1, 1, -1, 1, -1]

[119]:
```
set(lis)
```

[119]: {-1, 1}