

# Introduction to Python

Block 1. Part 4.

Sets

Conversions and functions on: Lists, Dictionaries, Tuples, Sets

Generators

# “Container objects”

- Lists
- Tuples
- Dictionaries
- Sets

# “Container objects”

- Lists: Ordered by position, index
- Tuples: Ordered by position. Similar to lists but does not support item assignment.
- Dictionaries: Mapping Key:Value. Unique key-value pairs!
- Sets

# “Container objects”

- Lists: Ordered by position.
- Tuples: Ordered by position. Similar to lists but does not support item assignment.
- Dictionaries: Mapping Key:Value.
- Sets

# Sets

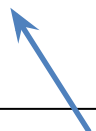
- Unordered collection of unique elements.
- To create an empty set: **set()**
- To create a set with initial data:

**{ obj1, obj2, ... }**

**set([obj1, obj2, ...])**

```
>>> my_set = set([1,2,3,4,5,6])
>>> my_set
set([1, 2, 3, 4, 5, 6])

>>> my_set = set(["a","b","c",1,2,"a"])
>>> my_set
set(['a', 1, 'c', 'b', 2])
```



A set can contain objects of any types

# Sets operators

- As **unordered** containers, sets do not have the operators:
  - ~~Concatenate: +~~
  - ~~Replicate: \*~~
  - ~~Indexing: []~~
  - ~~Slicing: [:]~~

# Sets

- Operator **in**: check if an element is found in the set

```
>>> my_set = set([1,2,3,4,5])
>>> 4 in my_set
True
```

- **not in**

```
>>> my_set = set([1,2,3,4,5])
>>> "4" in my_set
False
```

# Sets

- len: built-in function to get the length of a set

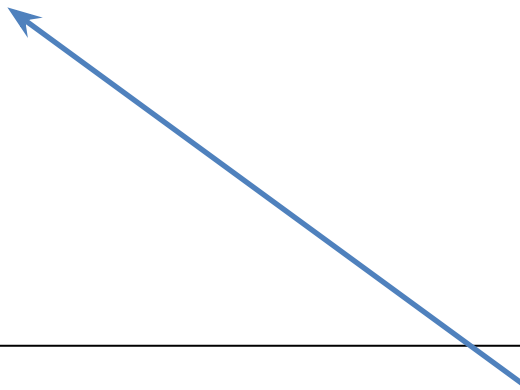
```
>>> my_set = set([1,2,3,4,5,1,3,1,5,10,2,1])  
>>> len(my_set)  
6
```



# Sets

- Traversal of a Set: **for** element **in** set\_object

```
>>> my_set =  
set(["a", "b", "c", "d", "e", "f", "g", "h"])  
>>> for letter in my_set:  
...     print(letter)  
...  
a  
c  
b  
e  
d  
g  
f  
h
```



Not ordered!

Order can be different in different computers!

# Set methods: creation, modification

- **add**: add an element to the set object.
- **clear**: remove all elements in the set.
- **update**: update with the union to other lists/sets
- **pop**: Remove and return an arbitrary set element. Raises `KeyError` if the set is empty.
- **remove**: Remove an element from a set; it must be a member. If the element is not a member, raise a `KeyError`
- **discard**: Remove an element from a set if it is a member. If the element is not a member, do nothing.'

# Set methods: comparing sets

- **difference**: returns a new set object
- **difference\_update**: modifies the set object
- **intersection**: returns a new set object
- **intersection\_update**: modifies the set object
- **issubset / issuperset**
- **union**
- **isdisjoint**

# “Container objects” summary

- Objects that contain other objects: containers
- Organize objects according to different requirements:
  - How I should be able to find objects?
    - By position? **Lists and tuples**
    - By a key ? **Dictionaries**
    - No direct access. **Sets**.
  - Should I remove “duplicate” objects, or allow duplicated ones?
    - Allow duplicated objects: **Lists, Tuples. Dict** (values)
    - Do not allow duplicated objects: **Dict** (keys), **Sets**
  - Should I be able to “modify” my group of objects?
    - » No: **Tuples**
    - » Yes: **Lists, Dictionaries, Sets**

# Conversions between container objects

- List to tuple: `tuple( list_object )`
- Tuple to list: `list( tuple_object )`
- List of tuples to dictionary: `dict(list_of_tuples)`

```
>>> my_list = [ ("key1", 1), ("key2", 2), ("key3", 3) ]  
>>> dict(my_list)  
{'key3': 3, 'key2': 2, 'key1': 1}
```

- Dictionary to list of tuples:
  - `dict_object.items()`: Returns an iterator of tuples

# Conversions between container objects

- List (or tuple) to set: `set( list_object )`
- Set to list: `list( set_object )`
- Set to tuple: `tuple( set_object )`

# Creating lists

- Built-in methods to create lists:
  - **sorted**: Sorts an iterator object and returns a new list object.

```
>>> sorted([3,1,10,-1,2,20])  
[-1, 1, 2, 3, 10, 20]
```

sorted is different to List.sort:

The method sort of the object List sorts the elements inside the list object, but None is returned.

**sorted** returns a new sorted list object!

# Generators



# Python iterators/Generators

- **Iterator:** Object that allows different types of iteration.
  - Strings, tuples, Lists, Dictionaries, Sets
    - **for** element **in** iterator\_object
    - **len**(iterator\_object)

# Python iterators/Generators

- **Generator**

- Tool to create iterators.
- A function can be converted to a generator with the **yield** statement, instead of return.
- Each time next() is called, the generator resumes it left-off (it remembers all data values and which statement was last executed).

# Python iterators/Generators

- **Generator: examples**

- Some methods of objects are generators:
  - Dictionaries: `items()`, `keys()`, `values()`
- Create a generator from a function

# Python iterators/Generators

- **Example.** A function that returns a list

```
def create_range(n):  
    i = 0  
    my_list = []  
    while( i < n ):  
        my_list.append(i)  
        i += 1  
    return my_list
```

```
>>> create_range(5)  
[0,1,2,3,4]
```

```
>>> for element in create_range(5):  
...     print(element)  
...  
0  
1  
2  
3  
4
```

**Not a generator**

# Python iterators/Generators

- **Example: Generator function**

```
def create_range(n):  
    i = 0  
    while( i < n ):  
        yield i  
        i += 1
```

```
>>> print(create_range(5))  
<generator object my_generator at 0x109b1e230>
```

```
>>> for element in create_range(5):  
...     print(element)  
...  
0  
1  
2  
3  
4
```

This is a generator!

# Exercises. Block 1. Part 4.

Create a python script called `NIE_exercise_block1_part4.py` with:

- 1) A Generator Function that reads a Fasta file. In each iteration, the function must return a tuple with the following format: (`identifier`, `sequence`). Function name:

`FASTA_iterator( fasta_filename )`

- 2) Given a list of FASTA files, create a function that returns a dictionary that contains the 4 following keys with the associated values:
  - “intersection”: a set with the common identifiers found in all the files
  - “union”: a set with all the identifiers (unique) found in all the files
  - “frequency”: a dictionary with all the identifiers as keys and the number of files in which it appears as values (int)
  - “specific”: a dictionary with the name of the input files as keys and a set with the specific identifiers as values (i.e. identifiers that are exclusive in that fasta file)

Note: Common identifier equivalence must be case-insensitive (i.e. `Code_A`, `code_a` and `CODE_A` are equivalents).

Note: It must use the `FASTA_iterator` function created in exercise 1. Function name:

`compare_fasta_file_identifiers( fasta_filenames_list )`