Introduction to Python

Session 04

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

“Container objects”

- Lists
- Tuples
- Dictionaries
- Sets

upf.

“Container objects”

- **Lists:** Ordered by position, indexed. access them via index
- **Tuples:** Ordered by position. Immutable (no item assignment).
- **Dictionaries:** Mapping Key:Value. Unique key-value pairs !
- **Sets**

upf.

“Container objects”

- **Lists:** Ordered by position, indexed.
- **Tuples:** Ordered by position. Immutable (no item assignment).
- **Dictionaries:** Mapping Key:Value. Unique key-value pairs !
- **Sets**

upf.

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

upf.

Set operators

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

upf.

Sets

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

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

- **not in** to see if something is not in the set

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

upf.

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
```

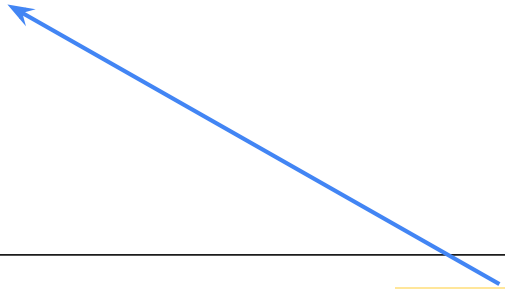
upf.

Sets

- Traversal of a Set: **for** element **in** set_object

iterates through all the elements in the set

```
>>> 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.'

upf.

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** to check if one set is subset or superset to another
- **union**
- **isdisjoint** sets that do not contain any common element

upf.

“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.

upf.

“Container objects”: Summary

- Objects that contain other objects: **containers**
- Organize objects according to different requirements:
  - Should I allow **duplicated** objects?
    - Allow duplicated objects: Lists, Tuples. Dict (values)
    - Do not allow duplicate objects: Dict (keys), Sets

upf.

“Container objects”: Summary

- Objects that contain other objects: **containers**
- Organize objects according to different requirements:
  - Should I be able to **modify** my group of objects?
    - No: Tuples
    - Yes: Lists, Dictionaries, Sets

upf.

Conversion between containers

- 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

upf.

Conversion between containers

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

upf.

EXAM!!!

Creating lists

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

```
>>> 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 and **`None`** is returned.

`sorted()` returns a new sorted list object!

upf.

Generators

Iterators vs. Generators

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

upf.

Iterators vs. Generators

- **Generator**
  - Function that behaves as an iterator.
    Can be used in for loops
  - **Lazy evaluation**: items are only generated when requested.
  - It retains the state between calls.
  - **Memory** efficient

upf.

Iterators vs. Generators

- **Generator**
  - A **function** can be converted to a **generator** with the **yield** statement, instead of return.
  - Each time **next()** is called on the generator, it generator resumes it where it left-off .

upf.

Iterators vs. Generators

- **Examples of generators**
  - Some built-in functions:
    - range: `range(1, 20)`
    - zip: `zip([1,2,3], ["a", "b", "c"])`
    - enumerate: `enumerate(["a", "b", "c"])`
  - Create a generator from a function

upf.

Iterators vs. 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]
```

Not a generator

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

upf.

Iterators vs. Generators

- **Example**. A 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!

upf.

Iterators vs. Generators

- **Example**. A 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>

>>> a = create_range(5)
>>> print(next(a))
>>> print(next(a))
>>> print(next(a))
>>> print(next(a))
>>> print(next(a))
>>> print(next(a))
```

upf.

Exercises. Session 04

Create a python script called**I<uID>_S04.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)`**

In a normal function, return ends the function entirely. In a generator, yield (identifier, sequence) pauses the function and gives the data to the user. When the loop asks for the next item, the fuinction "wakes uip" exactly where it left off.

Exercises. Session 04

Create a python script called **ID_exercise_block1_part4.py** with:

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

   Note 2: It must use the FASTA_iterator function created in exercise 1.

   Function name:
   `compare_fasta_file_identifiers( fasta_filenames_list )`

upf.