FS12

Week 02
Python Basics II

Mikhail Masyagin
Daniil Devyatkin

Python Collections I

- In the previous lecture you have learned basic Python data types:
  - numeric: integers, float, complex;
  - logical (yes/no, true/false): boolean;
  - strings.
- These data types are sufficient for solving small programming tasks like your homework for previous week. However, sometimes you need to do much more complex stuff. For example, remember multiple numbers or strings, provided by user.

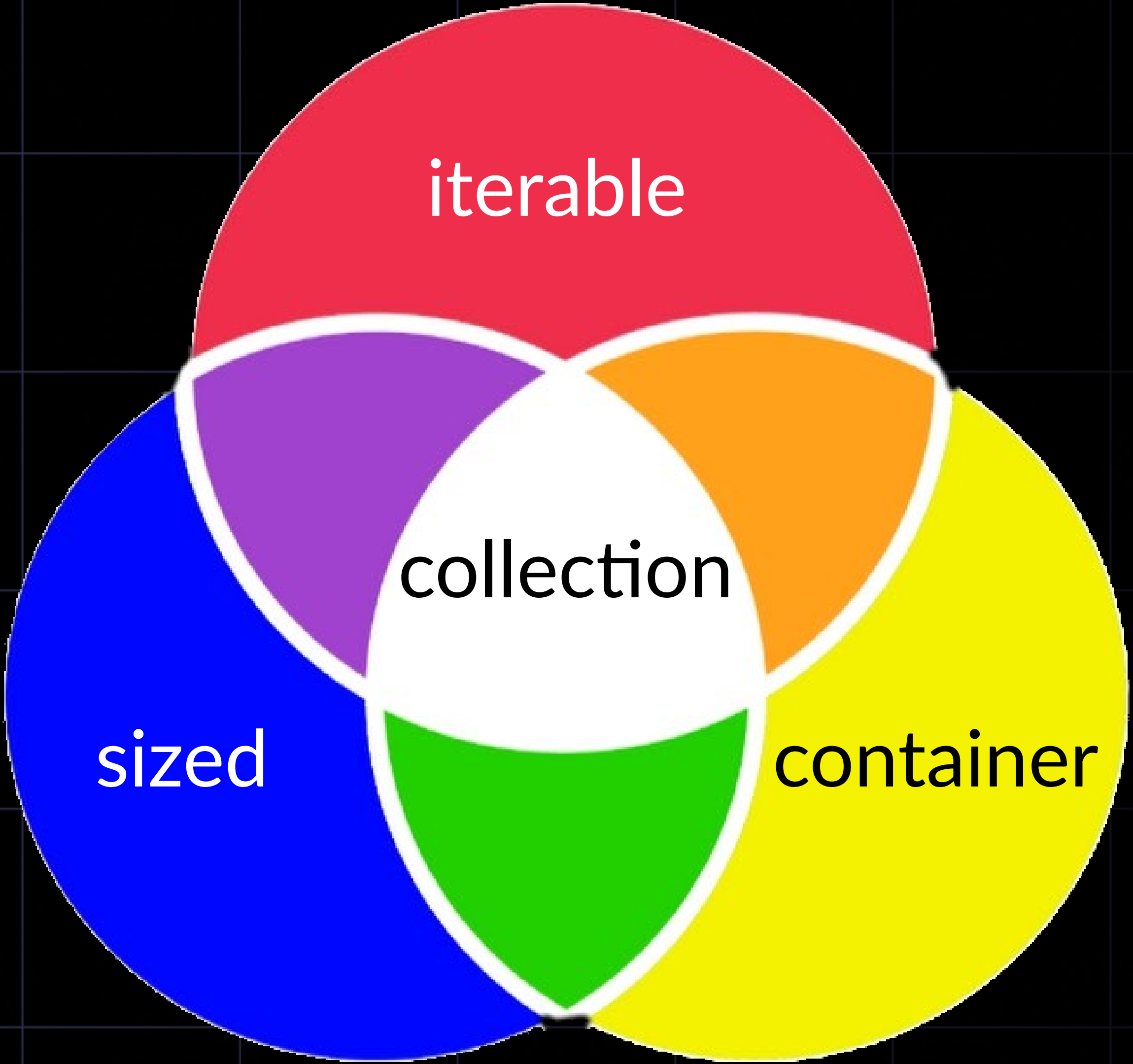
Python Collections II

- Python **Collections** can help us with it!
- Let’s try to answer the question: what is collection?
- hm…
- hm-hm…
- hm-hm-hm…
- Well, from the naming we can assume, that collection is something, that collects elements.
- Good answer, but not enough.

Python Collections III

- To find the best definition for Collections let’s discuss three base Python objects concepts first:
  - Being **<u>Container</u>** – object’s ability to answer the question "if it contains something or not?".
  - Being **<u>Iterable</u>** – ability to loop through all object’s elements.
  - Being **<u>Sized</u>** – object’s ability to answer the question "how many elements does it contain?".
  - **<u>Collection</u>** is something that is **<u>Container</u>**, **<u>Iterable</u>** & **<u>Sized</u>**.

Python Collections IV



Container I

- Being **Container** – object's ability to answer the question "if it contains something or not?".
- All collections are containers, however not all containers are collections (!!!).
- Example: $(a, b) = \{X \in \mathbb{R} | a < x < b\}$:
  - This mathematical interval is a **Container**, because it can if number lies in it.
  - It is NOT **Iterable**, because it contains infinite number of elements and you can not choose step.
  - It is not **Sized** for similar reason.

Container II

```
from collections.abc import Container
from dataclasses import dataclass


@dataclass
class Interval(Container):
    a: float
    b: float

    def __contains__(self, x):
        return self.a < x < self.b

interval = Interval(0, 1)
print(interval) > Interval(a=0, b=1)
```

```
for x in (0.5, 0.7, 1.2):
    c = x in interval
    print(f"{x}: {c}")
> 0.5: True\n 0.7: True\n
1.2: False

for x in interval:
    print(x)
> TypeError: 'Interval'
object is not iterable

len(interval) > TypeError:
object of type 'Interval'
has no len()
```

Iterable I

- Being **Iterable** – ability to loop through all object’s elements.
- All collections are Iterable, however not all Iterables are collections (!!!)
- Example: generator (special object, which return new values each time it is being called):
  - Generator is iterable, because you can loop through its generated new values.
  - It is NOT a **Container**: it does not store elements.
  - It is NOT **Sized**, because their "size" depends on external conditions.

Iterable II

```
def generator(n):
    yield from range(n)

for i in generator(10):
    print(i, end=" ")
> 0 1 2 3 4 5 6 7 8 9
len(generator(10)) > TypeError: object
of type 'generator' has no len()
g = generator(10)
print(0 in g) > True
print(0 in g) > False
```

Sized I

- Being **Sized** – object’s ability to answer the question "how many elements does it contain?".
- All collections are Sized, however not all Sized are collections (!!!)
- Example: line segment in N-dim space:
  - It is a **Container**: it can say does point belong to it or not.
  - It is **Sized**, because its size can be calculated as distance between its two main points.
  - It is not **Iterable**, because it contains infinite number of elements.

Sized II

```
@dataclass
class Interval(Container):
    a: float
    b: float

    def __contains__(self, x):
        return self.a < x < self.b

    # TODO
```

Python Collections V

- Python Collections can be divided in two big groups: **Sequences** and **Mappings**.
- However, there are also sets, etc.
- Collection is a **Sequence** if all its elements are ordered → so then can be indexed by their serial number (e.g. array indexing).
- Collection is a **Mapping** if it provides a way to map (translate) one elements to another elements. For example strings into integers. From this point of view **Sequence** is a **Mapping** from integers (its indexes) to some data type.

Python Collections VI

- Finally, let's note that all Python Collections support following operations on them:
  - `` `len(c)` `` - getting the number of elements in collection.
  - `` `el in c` `` - checking if element is present in collection.
  - `` `for el in c:` `` - iterating through all collection's elements.

Sequences I

- There are following sequential basic data types in Python:
  - list – changeable elements sequence;
  - tuple – UNchangeable elements sequence;
  - range – generated sequence of elements;
  - string – Python strings;
  - bytearray – changeable bytes sequence;
  - bytes – UNchangeable bytes sequence.

Sequences II

- All sequences support indexing operations:
  - `` `s[i]` `` - gets i'th element of sequence s (starting from 0 (!!!));
  - `` `s[i:j]` `` - gets subsequence of elements from i'th to j’th (including i'th and excluding j’th);
  - `` `s[i:j:k]` `` - gets subsequence of elements from i'th to j’th with step k (including i'th and excluding j’th);
  - You can skip beginning or (and) starting indexes. Their default values are `` `0` `` and `` `len(s)-1` ``

Sequences III

- Python allows to index sequences not only from `` `0` `` and `` `len(s)-1` ``. You can use negative values too. For example, `` `s[-1]` `` returns the last element of sequence, `` `s[-5]` `` returns fifth element from sequence’s end.

Sequences IV

- All sequences support following operations on them:
  - All collection operations.
  - `s1+s2` - two sequences concatenation.
  - `s1*n` - duplication of sequence n times.
  - `min(s)|max(s)` - min (max) element.
  - `s.index(el)` - index of first occurrence of the selected element in sequence.
  - `s.count(el)` - number of occurrences of the selected element in sequence.

List I

- List is the most basic sequence type in Python. It’s pretty similar to arrays, vectors or slices in C, C++, Go correspondingly. You can think that Python list is something like pill box:

- List elements can be accessed by indexes.
- Number of elements in list can shrink and grow over the time.
- Lists are easy to use but complicated inside.

List II

- Empty list creation:

```
a, b = [], list() # similar result
```

- Non-empty list creation:

```
a = [6, 7, 8, 9, 10, 11]
b = list(smth_iterable)
```

- Add element to list:

```
a.append(12)
print(a) > [6, 7, 8, 9, 10, 11, 12]
a += [13]
print(a)
> [6, 7, 8, 9, 10, 11, 12, 13]
```

List III

```
a.extend([14, 15])
print(a)
> [6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
a.insert(2, 22)
> [6, 7, 22, 8, 9, 10, 11, 12, 13, 14,
15]
```

- Delete element from list:

```
del a[0]; del a[0]; del a[3:5]
print(a) > [22, 8, 9, 12, 13, 14, 15]
a.pop(); a.pop(0) > 15\n 22
print(a) > [8, 9, 12, 13, 14]
```

List IV

```
a.remove(9)
print(a) > [8, 12, 13, 14]
a.remove(15) > ValueError:
list.remove(x): x not in list
```

- Reverse list:

```
a.reverse()
print(a) > [14, 13, 12, 8]
```

- Sort list:

```
a.sort()
print(a) > [8, 12, 13, 14]
sorted([3, 2, 1]) > [1, 2, 3]
```

List V

- Copy list:

  ```
  a.copy() > [8, 12, 13, 14]
  ```
- Be careful, `sorted()` and `copy()` result in new lists, so more memory is used (!!!)
- Delete all elements from list:

  ```
  a.clear()\n print(a) > []
  ```
- Of course, lists support all sequences operations.
- Arrays in languages like C\C++, Go, Rust, etc. allow you to store only elements of single type in them. In Python elements of any time can be saved in list (!!!)

  ```
  a = [8, 'Hello!', [[]], list()]
  ```

List VI

- List is a powerful sequence container, which allows you to store and process different elements (objects) in one place.
- Internally it is a following C structure:

```
typedef struct {
        PyObject_VAR_HEAD // macro
        PyObject **ob_item;
        Py_ssize_t allocated;
} PyListObject;
```

- `PyObject **ob_item;` shows us that list

List VII

does not store elements, it stores pointers to this elements.

- It grows every time, when we need to add new elements. Growing operation is quite heavy and expensive, so its calls are minimized: each time when you append new element to list, it grows on more then one elements. Some of its elements are just invisible for users.
- Adding new element to the end of list or deleting its last element are quite fast operations.

List VIII

- Adding & deleting elements in the beginning or middle of list is quite slow. Try to always prevent it in your code.
- You may ask lecturers: well, how fast and how slow?
- Okay, come to the fifth lesson and you will know.

Tuple I

- Earlier we said, that you can think about lists as a pill boxes with some nuances. Well, tuple is completely a pill box without any nuances:

- Tuple elements can be accessed by indexes.
- Number of elements is unchangeable
- Like a pill box tuples can heal you code: make it faster and safer.

Tuples II

- Empty tuple creation:

```
a, b = (), tuple() # similar result
```

- Non-empty tuple creation:

```
a, b = (1, 2, 3), (1,) # (el,) !!!
```

- `c = tuple(smth_iterable)`
- Add element to tuple:

Tuples III

- Tuples can be used when you want to create change-protected code.
- Because tuple sizes are constant, their creation is much faster than lists.
- Because tuple sizes are constant, they need less space than lists.

Tuples IV

- Tuples are faster than lists because of their internal C structure:

```
typedef struct {
    PyObject_VAR_HEAD // macro
    PyObject *ob_item[1];
} PyTupleOpbject;
```

- `PyObject *ob_item[1];` shows us that you do not need to use additional allocations.
- Also Python has tuples cache so small tuples (with length less than 20) are cached and if you delete

Tuples V

delete small tuple and ask Python to create it again, internally it will use the same tuple without any additional memory allocations.

- `PyTupleOpbject` does not have `Py_ssize_t allocated;`, so it requires less memory than `PyListOpbject`.

Range I

- Ranges are like Python tuples, however they do not store all their values in memory. They generate them when you ask them for it. They are quite efficient from memory point of view. Sometimes they are quite efficient from CPU utilization too.
- Ranges support all tuple operations.
- In Python 2 `` `range()` `` function call returns list instead of range. To make ranges work in Python2 you have to use `` `xrange()` `` function instead of `` `range()` ``.

Range II

- You can try to make following experiment to compare lists and ranges:

```
a = range(0, 1000000000000000000000000)
1000000000000000000000000 in a > True
1000000000000000000000001 in a > False
b = list(a) # OOM can happen
1000000000000000000000000 in b > True
1000000000000000000000001 in b > False
```

- List of 1000000000000000000000000 requires too many memory. Range does not require it at all.

Strings I

- You have already worked with Python strings on the first lesson. Strings allow you to
- Strings support all tuple operations.
- Strings in Python are immutable (unchangeable), so every time when you change string’s character, the new string is created. It prevents programmer from complicated errors with changing strings and their copying, however, it requires much more memory.

Strings II

- Strings in Python can be defined in two ways:
  - Classical string: `'Hello', "World"`.
  - Long strings:

```
'''Hello!
My name is Mikhail Masyagin.
I work at BMSTU as a lecturer.
Bye!'''
"""Hi!
My name is Daniil Devyatkin.
I'm a lecturer at BMSTU too.
Have a nice day!"""
```

Strings III

- Python natively supports Unicode strings, so strings with Russian or Chinese languages are okay:

```
'''Привет!
私の名前はミハイル・マシャギンです
ഞാൻ BMSTU-ൽ ലക്ചററായി ജോലി
ചെയ്യുന്നു.
سيتۋيلڭ!'''
```

Strings III

- Python natively supports Unicode strings, so strings with Russian or Chinese languages are okay:

```
'''Привет!
私の名前はミハイル・マシャギンです
ഞാൻ BMSTU-ൽ ലക്ചററായി ജോലി
ചെയ്യുന്നു.
سيتۋيلڭ!'''
```

Strings IV

- You can build Python strings using format strings syntax. Format string is a string with prefix `` `f` ``:

```
name = 'Mikhail'
n = 24
print(f'I\'m {name}, my age is {n}')
> I'm Mikhail, my age is 24
k, ending = 1830, '!'
print('BMSTU was founded at {}{}'.
      format(k, ending))
> BMSTU was founded at 1830!
```

Bytearray

- Bytearray is pretty similar to lists: it is a mutable bytes sequence:

```
b = bytearray(b'hello world!')
print(b) > bytearray(b'hello world!')
b[0] = 103
print(b) > bytearray(b'gello world!')
print(len(b)) > 12
for i in range(len(b)):
    b[i] += i
print(b) > bytearray(b'hfnos%}vzun,')
```

Bytes

- Bytes type is pretty similar to tuples: it is a immutable bytes sequence. So you can think of Bytes like immutable Bytearray.

```
b = b'bytes'
print(b) > b'bytes'
print(len(b)) > 5
list(b) > [98, 121, 116, 101, 115]
```

Mappings I

- **Mappings** provide a way to map (translate) one elements to another elements.
- You may say that list is already a mapping: it maps integers from 0 to n-1 (indexes) to some values. From this point of view, yes, list is a mapping.
- However, sometimes you need mapping not from all values from 0 to n-1 but only from their subset. You can try to achieve it, setting some list values to `None` and assuming that it represents no value. This is a good idea, however number of elements

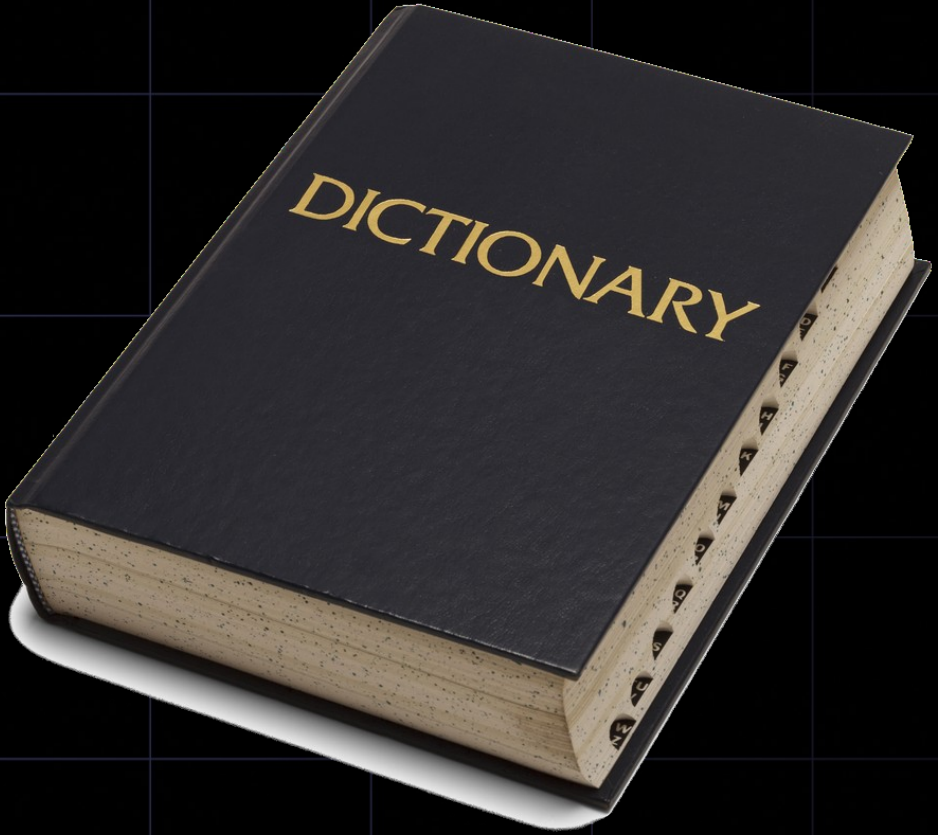
Mappings II

in this mapping will be wrong: it counts both empty and non-empty values as list elements.

- Sometimes you need to map not from integers but from strings or some other data type. Be careful, this data type should be hashable and comparable (!!!). For now you can simply remember that integers and strings can be used in mappings.
- The values from which we are mapping are named **Keys**, and the values to which we are mapping are named **Values**.
- **Keys** should be unique (!!!)

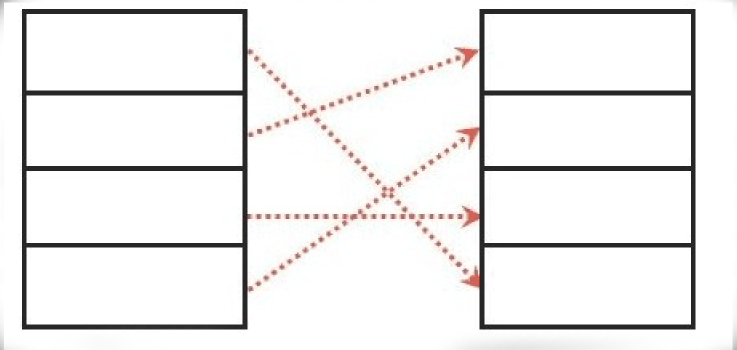
Mappings III

- Python supports only one mapping data type:
  - dictionary (dict).



Dict I

- Dict’s **<u>Values</u>** can be accessed by **<u>Keys</u>**.
- Number of elements in dict can shrink and grow over the time.
- Dicts are easy to use but complicated inside.



Dict II

- Empty dict creation:

```
a, b = {}, dict() # similar result
```

- Non-empty list creation:

```
a = {'a': 1, 'b': 2, 'c': 3}
b = {1: 4, 2: 9, 3: 84}
```

- Add element to dict:

```
a['d'] = 12
print(a) > {'a': 1, ..., 'd': 12}
b[5] = 6
print(b) > {1: 4, ..., 5: 6}
```

Dict III

```
a.update({'e': 5})
print(a) > {'a': 1, ..., 'e': 5}
b.update({7: 5, 99: 1})
print(b) > {1: 4, ..., 7: 5, 99: 1}
```

- Delete key-value pairs from dict:

```
del b[1]
b.pop(7) > 5
b.popitem() > random pair e.g. (99, 1)
print(b) > {2: 9, 3: 84}
```

- Copy dict:

```
a.copy() > {'a': 1, ..., 'e': 5}
```

Dict IV

- Delete all pairs from dict:

```
b.clear()
print(b) > {}
```

- Get all dict keys:

```
a.keys() > dict_keys(['a', ..., 'e'])
```

- Get all dict values:

```
a.values() > dict_values([1, ..., 5])
```

- Get all dict items:

```
a.items() > dict_items([('a', 1), ...
 ('e', 5)])
```

Dict V

- Of course, dicts support all mappings (collections) operations.
- Dict size:

```
len(a) > 5
```

- Iterating over dict pairs:

```
for k, v in a.items():
    print(k, v) > a 1\n...e 5\n
```

- Iterating over dict keys:

```
for k in a.keys():
    print(k) > a\n ...e\n
```

Dict VI

- Iterating over dict values:

```
for v in a.values():
    print(v) > 1\n ...5\n
```

- Before Python 3.6 order of dict keys was completely random. You could not rely on it. In Python 3.6 keys were sorted in insertion-order internally (in CPython). In Python 3.7 insertion-order sorting became a part of the official Python standard.

Dict VII

- Mapping in languages like C++, Go, Rust, etc. allow you to store only elements of single type as keys and as values. In Python elements of any time can be saved as keys and values of dict:

```
a = {2: 3, 'Hello!': 'Hi!', 1: list()}
```

Dict VIII

- Dict is a powerful data structure, which allows you to map one elements to another:
- Internally it is a following C structure:

```
typedef struct {
    PyObject_HEAD // macro
    Py_ssize_t ma_used;
    uint64_t ma_version_tag;
    PyDictKeysObject *ma_keys;
    PyObject **ma_values;
} PyDictObject;
```

Dict IX

- Each `PyDictKeysObject` has pointer to following structure:

```
typedef struct {
        Py_hash_t me_hash;
        PyObject *me_key;
        PyObject *me_value;
} PyDictKeyEntry;
```

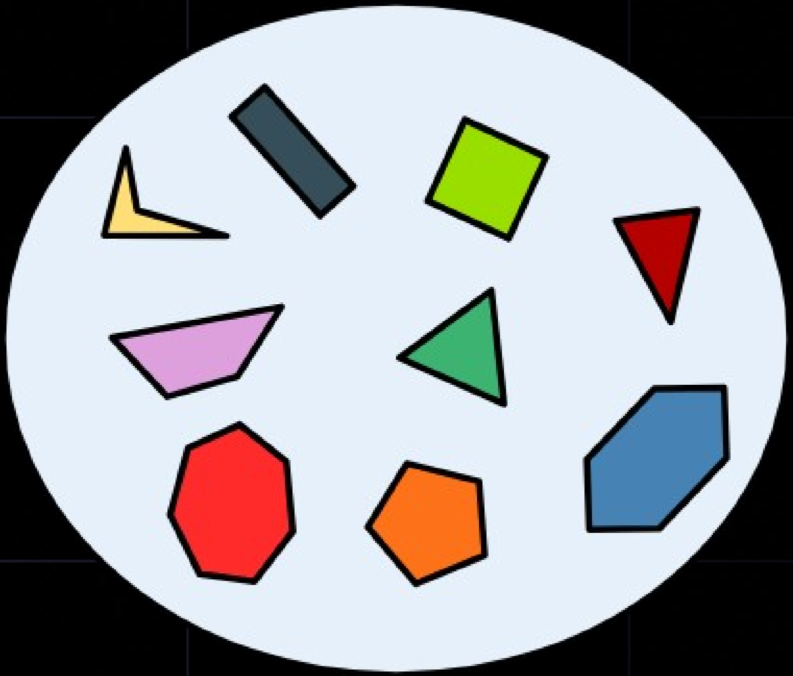
- Before learning Map Data Structure we can think that `PyDictKeysObject *ma_keys` stores all keys, `PyObject **ma_values` stores all vals,

Dict X

- and we can find value in `**ma_values` arrays by values in `*ma_keys` array using `me_hash` from `PyDictKeyEntry` structure.
- Internally Dict is a closed-hash mapping with on-array bucketing.

Set I

- Python's Set is a set from math. Set is a collection of a unique elements. You can add elements to set like a list. In Python set is represented by a `` `set()` ``.



Set II

- Empty dict creation:

```
a = set()
```

- Non-empty list creation:

```
a = {1, 2, 3, 4}
b = {'a', 'b', 'c', 'd'}
```

- Add element to set:

```
a.add(5)\n a.add(4)
print(a) > {1, 2, 3, 4, 5}
```

- Remove element from set:

```
a.discard(1)\n a.discard(5)
print(a) > 2
```

Set III

- Main set operations:
  - Join 2 sets:

```
a = {1, 2}
b = {2, 3}
a | b > {1, 2, 3}
```

- Main set operations (inplace):
  - Join 2 sets:

```
a, b = {1, 2}, {2, 3}
a.update(b)
print(a) > {1, 2, 3}
```

  - Intersect 2 sets:

```
a & b > {2} a.intersecion(b)
```

  - Difference:

```
a - b > {1} a.symmetric_difference(b)
```

  - Symmetric Difference:

```
a ^ b > {1, 3}
```

Type Casting

- You can convert list to tuple, tuple to list, set to list, dict keys to list, etc:

```
list((1, 2, 3)) > [1, 2, 3]
list({1, 2, 3}) > [1, 2, 3]
tuple([1, 2, 3]) > (1, 2, 3)
tuple({1, 2, 3}) > (1, 2, 3)
set([1, 2, 3]) > {1, 2, 3}
set((1, 2, 3)) > {1, 2, 3}
list({1: 2, 3: 4}.keys()) > [1, 3]
list({1: 2, 3: 4}.values()) > [2, 4]
```

Immutable vs Mutable

- Let’s recap **Immutable** (unchangeable) and **Mutable** (changeable) data types:
- **Immutable**:
  - numeric: integer, float, complex;
  - string, bytes;
  - tuple, frozenset.
- **Mutable**:
  - list, bytearray;
  - dict, set;
  - user defined types.

Comprehensions

- Comprehension is a syntax sugar for fast collections building:

```
[n ** 2 for n in range(10) if n % 2]
> [1, 9, 25, 49, 81]
{ch for ch in "abcabcbca"}
> {'a', 'c', 'b'}
{n: n**2 for n in range(10) if n % 2}
> {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
gen = (n ** 2 for n in range(10))
# it’s not a tuple!!!
```

Garbage Collector I

- Now you know how dynamic data types in Python can be created. However, you may want to know how then can be deleted too.
- In classical programming languages like C, Fortran and C++ (not modern C++) you have to release memory after you do not need objects manually. It results in many memory errors:
  - Sometimes you delete object to early and then try to access it → NullPointerException.
  - Sometimes you do not release object's memory, so it takes up memory space til the program end.

Garbage Collector II

- Modern languages like Python, Java, JavaScript, Go etc (some old languages like Lisp too) overcomes these problems via **Garbage Collector**.
- Garbage Collector is a special part of program’s runtime, which detects object, that can be deleted, and deletes them.
- Garbage Collector can be used both in interpreted and compiled programming languages. In interpreted programming languages its a part of interpreter, in compiled – part of a program.

Garbage Collector III

- How does Garbage Collector understand, which object can be cleaned?
- The answer is quite simple: every object, that can not be accessed from the current state of program, can be cleaned.
- Let’s check the following function:

```
def factorial(n: int) → int:
    xs = [x for x in range(1, n+1)]
    res = 1
    for x in xs:\n res *= x
    return x
```

Garbage Collector IV

- When program enters the function and executes it code list `` `xs` `` becomes available for us.
- When program return from the function, we can not access `` `xs` `` anymore, so it can be deleted. Be careful (!!!), on future calls of function `` `factorial` `` the NEW list `` `xs` `` will be created every time, so it is okay to clean it, after finishing function execution.

Garbage Collector V

- So Garbage Collector cleans all objects, that can not be accessed from the current program part.
- It can be done in two ways:
  - via **Reference Counter**;
  - via direct **Garbage Collector.**

Reference Counter I

- **Reference Counter** on each object tracks the number of references to it. For example:

```
a = [1, 2, 3] # [1, 2, 3]_rc=1
b = a # [1, 2, 3]_rc=2
b = [2, 2, 5] # [1, 2, 3]_rc=1
a = 5 # [1, 2, 3]_rc=0
# [1, 2, 3] can clean list's memory
```

- **Reference Counter** works in real-time and it is quite easy to implement, however it always steals time from your runtime.

Reference Counter II

- The biggest problem of **Reference Counter** is that it can not track cyclic dependencies. For example:

```
a, b = [0, 1, 2], [3, 4, 5]
a.append(b)
print(a) > [0, 1, 2, [3, 4, 5, [...]]]
b.append(a)
print(b) > [3, 4, 5, [0, 1, 2, [...]]]
a = 5
print(b) > [3, 4, 5, [0, 1, 2, […]]]
b = 6
# lists still reference each other
```

Reference Counter III

- As lists still reference each other, rcs on both of them are equal to one, so they can not be deleted.

Garbage Collector VI

- Direct **Garbage Collector** will help us!

Garbage Collector VII

- Direct **Garbage Collector** is super helpful in modern programming languages. It can delete all unused objects.
- If you have Reference Counter in your programming language, you have to worry about objects cross-links. With Direct **Garbage Collector** you have to worry about nothing.
- However, Direct **Garbage Collector** has some disadvantages:
  - It is quite complicated.

Garbage Collector VIII

- It has to operate in a separate thread or stop complete program’s runtime sometimes.
- Programming languages with direct garbage collection can not be used in real-time tasks like operating nuclear reactor or managing heart stimulator.
- For today writing efficient garbage collector strategies is one of the biggest tasks for system & compiler programmers.

Garbage Collector in Python

- Python has both **Reference Counter** and Direct **Garbage Collector**.
- Garbage Collector can be disabled for performance optimization. Sometimes developers disable GC in Python program and create a monitor on it, which restarts it every time it dies because of OOM.
- **Reference Counter** can not be disabled.
- Garbage Collector in Python will delete all your unused objects when its possible (ha-ha-ha), but please always think about memory limits and ways to optimize memory usage.

Type Hint’s I

- Python is a dynamically typed programming language, however sometimes we want to have some statically typed features in it. One of ways to have them is usage of Python’s Type Hints.
- Type Hints:
  - help to identify error, while writing code;
  - improve code readability;
  - add IDE hints;
  - are a good style;
  - SUPER useful for LONG term development;

Type Hint’s II

- Basic Type Hints are complete data types, e.g.:
  - Int, float, complex, bool, str,
  - list, tuple, dict, set.
- However, it is better to use Type Hints from typing library, because they provide more readability. Also they can define not complete types but their super classes like Iterable, Sized, Container, Collection, etc:

```
from typing import *
```

  - Tuple, Mapping, Any, Dict, List.

Type Hint’s III

```
def f1(x: int, xs: List[int]) →
List[bool]:
    return [v > x for v in xs]

def f2(x: Tuple[int, int]) → int:
    return x[0] * x[1]

def f3(xs: Tuple[int, …]) → int:
    res = 0
    for x in xs:\n res += x
    return res
```

Type Hint’s IV

```
def f4(m: Mapping[str, Any]) →
List[Any]:
    keys = list(m.keys())
    values = list(m.values())
    return keys + values

def f5(x: int) → Tuple[int, int, int]:
    return x, x**2, x**3
```

Just look here

Bye