FS12

Week 02
Python Basics II

Mikhail Masyagin
Daniil Devyatkin

Base Collections Types: List I

What we should do for collect multiply elements in memory? There are good data structure – array. You can imagine pill box:

But array can have big count of elements and option – to increase size. In Python (dynamic) array is represented by a \`list()\` or \`[]\`.

Base Collections Types: List II

- Empty list creation

```
>>> lst1 = []
>>> lst1 = list()
```

- Non empty list creation

```
>>> lst2 = [6, 7, 8, 9, 10, 11]
```

`>>> lst2 = list(collection),` where collection is iterable;

- Add element to list
- `>>> lst2.append(4) # inplace method`

Base Collections Types: List II

- Get element

```
>>> lst2[1] # 7
```

- Get multiply elements: list[start: stop: step]

```
>>> lst2[2:6:2] # [8, 10]

>>> lst2[0:len(lst2):2] # [7, 9, 11]

>>> lst2[::-1] # [11, 10, 9, 8, 7, 6]

>>> lst2[:3] # [6, 7, 8]

>>> lst2[3:] # [9, 10, 11]
```

```
lst2[0:len(lst2):2] <=> lst2[::2]
```

Base Collections Types: List III

- len(tuple) – return length of list;
- list.extend(iterable) – add other collection to list, inplace method;
- list.insert(i, x) – insert an item x at a position x;
- list.copy() - return a shallow copy of the list;
- list.index(x) – retrun pos x in list, if existence;
- list.sort(key, reverse) – sorting list inplace by key and reverse order;
- sorted(list) – return sorted list;
- And more...

Base Collections Types: Tuple I

- Empty tuple creation

```
>>> tpl = ()
>>> tpl1 = tuple()
```

- Non empty tuple creation

```
>>> tpl2 = (1, )
>>> tpl2 = (1, 2, 3)
>>> tpl2 = tuple(collection)
```

where collection is iterable;

- Add element to tuple

Nothing can be added to tuple!!!

Base Collections Types: Tuple II

- len(tuple) – return length of list;
- tuple.index(x) – return pos x in list, if existence;

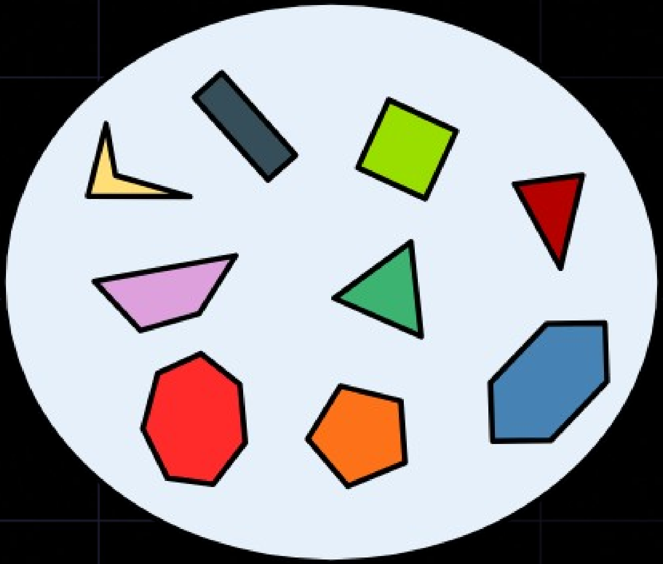
  ...

But tuple is faster than list!

- Element access similar to lists: see slide 4

Base Collections Types: Set I

- 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()` `` or `` `{}` ``.



Base Collections Types: Set II

- Empty set creation

```
>>> s1 = set()
```

- Non empty set creation

```
>>> s2 = {1, 2, 3}
>>> s2 = set(collection)
```

where collection is iterable;

- Add element to set

```
>>> s2.add(4) # inplace method
>>> s2.add(1) # collection is not change
```

Base Collections Types: Set III

- len(set) – return length of list;
- set.discard(x) – remove the specified item;
- set.intersection(set1) – returns a set, that is the intersection of two or more sets;
- set.copy() - returns a copy of the set;
- set.symmetric_difference(x) –returns a set with the symmetric differences of two sets;
- set.issubset(set1) – check, is set1 subset of set;
- And more...

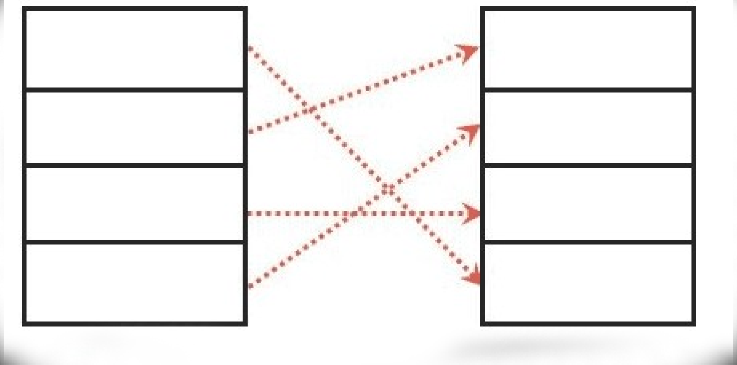
Base Collections Types: Set III

- Also we apply logical operators to sets;
- | - or, & - and, `-` - difference, ^ - xor;

```
>>> x1 = {1, 3}
>>> x2 = {2, 3, 4}
>>> x1 & x2 # {3}
>>> x1 | x2 # {1, 2, 3 ,4}
>>> x1 – x2 # {1}
>>> x1 ^ x2 # {1, 2, 4}
```

Base Collections Types: Dict I

- Dict is a data structure for collecting keys and values, where access to value is by key. Keys should be unique!!! For example we can use people like a key and their pills like a value.



Base Collections Types: Dict II

- Empty dict creation

```
>>> d1 = dict()
>>> d1 = {}
```

- Non empty tuple creation

```
>>> d2 = {“a”: 1, “b”: 2, “c”: 3}
>>> d2 = dict(a=1, b=2, c=3)
>>> d2 = dict(list(tuple))
```

- Add element to set

```
>>> d2[“d”] = 4
>>> d2.update({“e”: 5}) # inplace method
```

Base Collections Types: Dict III

- len(dict) – return amount of keys;
- dict.keys() – returns a list of a dict’s keys;
- dict.values() - returns a list of all the values in the dict;
- dict.items() – returns a list of a tuple for each key value pair;
- dict.setdefault(key, value) - returns the value of the specified key.
- And more...

Type casting

- You can create mapping list → tuple, set → 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}
```

Base Types: immutable or mutable

Mutable

- list
- dict, set
- user defined

Immutable

- int, float, bool
- str, bytes
- tuple, frozenset

Some Python’s Feature

- In Python you can create list, tuple, set and dict of elements of different types:

```
>>> tpl = (1, “a”, True, (76, ), [99])
>>> lst = [1, “a”, True, (76, ), [99]]
>> > s = {1, “a”, True, (76, )}
>>> d = {1: “1”, “2”: 2, }
```

BUT you can’t use mutable types like a key in dict and elements in set!!!

Container’s iterators

```
>>> lst = [1, 3, 2]
>>> dct = {1: 11, 2: 22}
```

Operator `in`:

- `>>> 1 in lst # True`
- `>>> 1 in dct # True`
- `>>> "3" not in "456" # True`

Iteration:

- `>>> for val in lst: pass`
- `>>> for key, val in dct.items(): pass`

```
# tuple and set similarly
```

Comprehension

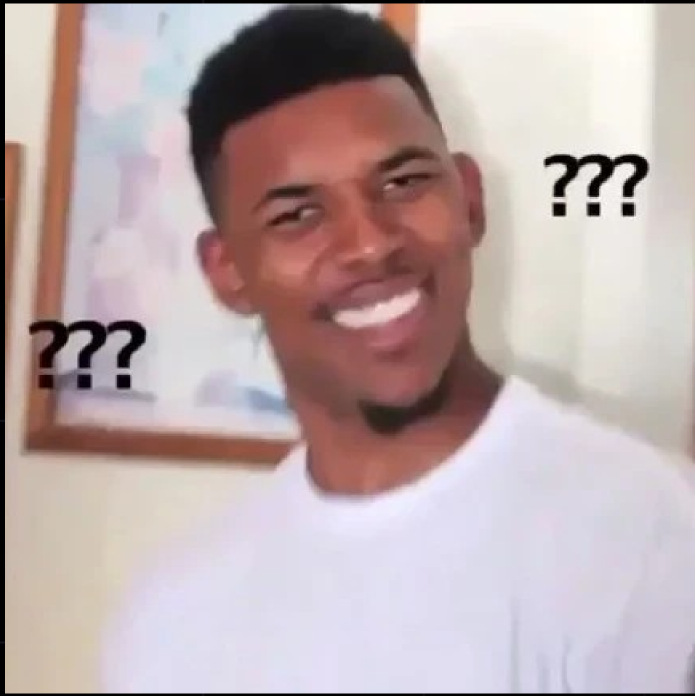
Comprehension is a syntax sugar for fast collections building

```
>>> lst = [n ** 2 for n in range(10) if n % 2]
# [1, 9, 25, 49, 81]

>>> s = {ch for ch in "abcabcbca"}
# {'a', 'c', 'b'}

>>> d = {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) if n % 2)
# it’s not a tuple!!!
```

Type hint’s I

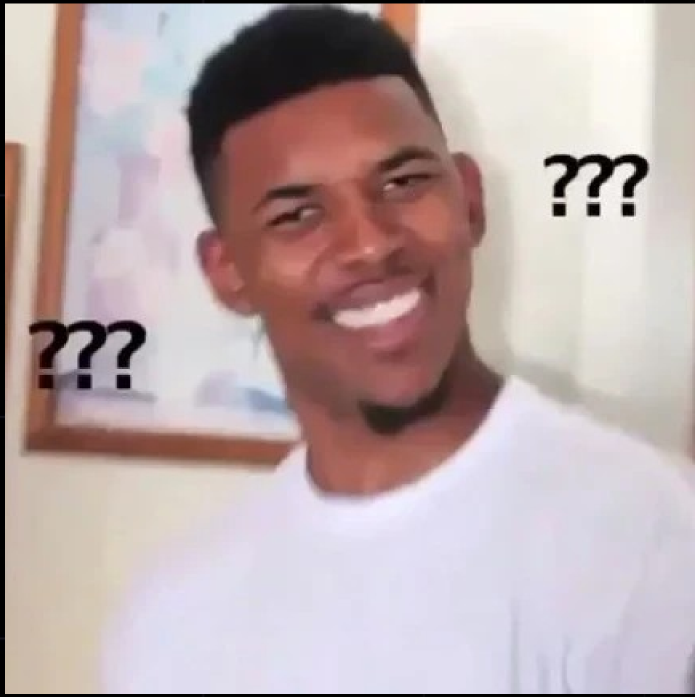
For what??? Python is a dynamic PL…

- Identifying errors, when you write code;
- readability, comprehensibility, maintainability for code;
- IDE hints;
- It’s a good style
- VERY useful for LONG time development;



Type hint’s II

- int, float, str, bool;
- list, tuple, dict, set;
- from typing import *
- NamedTuple, NamedDict



Type hint’s Example I

```
def add(x: float, y: float) → float:
    return x + y
```

Type hint’s Example I

```
def get_keys(
    x: dict,
    is_sort: bool,
) → list:
    keys = x.keys()
    return sorted(keys) if is_sort else keys
```

Bad practice

```
from typing import Mapping, List
def get_keys(
    x: Mapping[str, int],
    is_sort: bool,
) → List[str]:
    keys = x.keys()
    return sorted(keys) if is_sort else keys
```

Best practice