Информационные технологии

Основы программирования на Python 3

Каф. ИКТ РХТУ им. Д.И. Менделеева
Ст. преп. Васецкий А.М.

Москва, 2018

Лекция 3. Типы данных
Часть 2.
Сложные типы данных

1. Списки (Lists) и Кортежи (Tuples)
2. Словари (Dictionaries)
3. Множества (Sets)
4. Фиксированные множества (Frozen sets)
5. Байты (Bytes)
6. Массивы байтов (Byte Arrays)
7. Прочие типы данных

1. Списки и кортежи

- Кортежи (***tuple***) и списки (***list***) могут содержать ноль или более элементов разных типов.
- Фактически каждый элемент может быть любым объектом Python. Это позволяет создавать структуры любой сложности и глубины.
- Кортежи неизменяемы,
- Списки изменяемы, т.е. в них можно добавлять и удалять элементы.

Примечание. В других языках программирования для подобного типа данных иногда встречается термин "Коллекция"

Создание списков

- Список можно создать из нуля или более элементов, разделенных запятыми и заключенных в **квадратные скобки**

empty_list = [ ]
weekdays = ["Пн", "Вт", "Ср", "Чт", "Пт"]
animals = ["ёж", "уж", "лис"]
names = ["Яна", "Юля", "Яна", "Лена"]

или с помощью функции **list()**

empty_list = list()

создание списка при помощи разбиения:

splitme = "a/b//c/d///e"
splitme.split("/") ***# ["a", "b", "", "c", "d", "", "", "e"]***

Примечание: Если требуется рассмотреть только уникальные значения, то лучше воспользоваться множеством ***(set)***

Запятая в конце списка

- Запятая помогает устранить определенный тип ошибок. Иногда бывает проще писать списки на нескольких строках. Но, затем потребуется переупорядочить предметы. Если написали

```
L1 = [
      1,
      2,
      3,
      4,
      5
] # и хотите перетасовать строки…
L1 = [
      1,
      2,
      3,
      5
      4,
] # то получите ошибку
```

```
L1 = [
      1,
      2,
      3,
      4,
      5,
]
```

Так более оптимально

Функция list()

- Функция ***list()*** преобразует другие типы данных в списки.

Пример: Разбиваем строку

```
list("кот")                  # ['к', 'о', 'т']
mylist = list("one", "two") # ОШИБКА
```

Преобразование кортежа в список

```
my_tuple = ("раз", "два", "три")
list(my_tuple)   # ['раз', 'два', 'три']
```

Получение элементов списков

 Получение одного элемента списка:

```
list_names = ["Юля", "Яна", "Лена"]
list_names[0]  # 'Юля'
list_names[-1]  # "Лена"
list_names[3] # ОШИБКА
```

 Изменение элемента списка:

```
list_names[0] = "Таня"
```

Копирование списков

- Копирование списков
- ***list_names = ["Юля", "Яна", "Лена"]***

```
l_names = list_names   # это будет ссылка!
l_names[1] = "Женя" # изменит оба списка!
list_names     # ['Юля', 'Женя', 'Лена']
l_names        # ['Юля', 'Женя', 'Лена']
l_names = list_names.copy() # копия
l_names[1] = "Женя" # изменит только l_names
list_names     # ['Юля', 'Яна', 'Лена']
l_names        # ['Юля', 'Женя', 'Лена']
```

Альтернативно копирование выполняется:

```
c = list(list_names )
d = list_names[:]
```

- Повторение списка:

```
my_list = [1, 2, 3] * 2 # [1, 2, 3, 1, 2, 3]
```

Вложенные списки

- Списки могут содержать элементы различных типов, включая другие списки.

Пример:

```
list_a = ["раз", "два", "три"]
list_b = ["0", "1", "2", "3"]
list_c = ["one", "two", "three", "four"]
```

- Вложенный список:

```
lst =[list_a, list_b, "список", list_c, [0, 1, 2], 7]
```

Его элементы:

```
lst[0] # ['раз', 'два', 'три']
lst[0][2] # 'три'
lst[4][0] # 0
lst[2][2] # 'и'
lst[5][2]  # ОШИБКА
```

Повторение вложенных списков

mylist = [[]] * 5 # [[], [], [], [], []]

Однако при этом операция:

mylist[0].append(1) # [[1], [1], [1], [1], [1]]

Несколько забегая вперёд, лучше конструкция вида

mylist = [[] for _ in range(5)] # [[], [], [], [], []]

mylist[0].append(1) # [[1], [], [], [], []]

Бесконечно вложенный список

```
a = [1, 2, 3, 4]
a.append(a)        # [1, 2, 3, 4, [...]]
a[4]               # [1, 2, 3, 4, [...]]
a[4][4][4][4][4][4][4][4][4][4] == a # True
```

Операции со списками

- Разделение списков

```
my_list = [0, 10, 20, 30]
my_list[0:2]    # [0, 10]
my_list[2:]     # [20, 30]
my_list[::2]    # [0, 20]
my_list[::-1]   # [30, 20, 10, 0]
```

Методы списков

- *[…, 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']*

Метод	Что делает
list.append(x)	Добавляет элемент в конец списка
list.extend(L)	Расширяет список list, добавляя в конец все элементы списка L
list.insert(i, x)	Вставляет в *i*-ю позицию значение x
list.remove(x)	Удаляет первый элемент в списке, имеющий значение x. ValueError, если такого элемента не существует
list.pop([i])	Удаляет *i*-й элемент и возвращает его. Если индекс не указан, удаляется последний элемент
list.index(x, [start [, end]])	Возвращает положение первого элемента со значением x (поиск ведется от ***start*** до ***end***)
list.count(x)	Возвращает количество элементов со значением x
list.sort([key=функция][reverse=False])	Сортирует список на основе функции
list.reverse()	Инверсия списка
list.copy()	Поверхностная копия списка
list.clear()	Очищает список

Примеры

```
my_list = [0, 10]
my_list.append(30)          # [0, 10, 30] – добавили
my_list.append([1, 2])      # [0, 10, [1, 2]]
```

 Слияние двух списков

```
my_list.extend([1, 2])      # [0, 10, 1, 2]
my_list+= [1, 2]            # [0, 10, 1, 2]
```

 Вставка списка

```
my_list.insert(0, [1, 2])   # [[1, 2], 0, 10]
my_list.insert(10, [1, 2]) # [0, 10, [1, 2]]
```

 Индекс первого вхождения элемента

```
my_list = [0, 10, 20, 10]
my_list.index(10)           # 1
my_list.index(10, 2)        # 3
```

 Количество включений

```
my_list.count(10)           # 2
```

Примеры

my_list = [0, 10]

- Длина списка

len(my_list) ***# 2***

- Проверка нахождения элемента в списке

10 in my_list ***# True***

- Объединение элементов списка (строки)

my_list = ["a", "b", "c"]

", ".join(my_list) ***# "a, b, c"***

- Сортировка

my_list.sort(reverse=True) ***# ['c', 'b', 'a']***

- Сортировка по ключевой функции

my_list = ["aa", "b", "ccc"]

key – в данном случае отвечает за вычисление длины строки. Используется лямбда-функция.

my_list.sort(key=lambda x: len(x)) ***# ['b', 'aa', 'ccc']***

Примеры сортировки

```
mylist = [0, 2, 1, 9, 7]
mylist.sort()              # [0, 1, 2, 7, 9]
mylist.sort(reverse=True)  # [9, 7, 2, 1, 0]
mylist = [0, 2, 1, 9, 7]
mylist.reverse             # [0, 2, 1, 9, 7]
mylist.reverse ()          # [7, 9, 1, 2, 0]
mylist = [0, "20", 1, "t", 7]
mylist.sort()              # ОШИБКА
mylist = ["0", "20", "1", "9", "70"]
mylist.sort()              # ['0', '1', '20', '70', '9']
mylist = mylist.sort()     # None
```

Удаление элементов из списка

my_list = [0, 10, 20, 30]

 Удаление элемента списка по индексу (***del***)

```
del my_list[1]          # [0, 20, 30]
del my_list[1:3]        # [0, 30]
del my_list[:]      # очистка списка
my_list.clear()         # очистка списка
```

 Удаление по значению

```
my_list.remove(2)  # ОШИБКА
my_list.remove(20)# [0, 10, 30]
```

 Извлечение элемента из списка

```
ex = my_list.pop()  # ex=30, my_list = [0, 10, 20]
ex = my_list.pop(0)     # ex=0, my_list = [10, 20, 30]
```

Можно воспользоваться модулем **collections**

https://docs.python.org/3/tutorial/datastructures.html

Кортежи (Tuples)

- Кортежи, как и списки, являются последовательностями произвольных элементов.
- В отличие от списков кортежи **неизменяемы**

Создание кортежей

```
mytuple = ()          # пустой кортеж
mytuple = ("a", )     # один или более элементов
mytuple = ("a", "b", "c") # больше 1 элемента
mytuple = "a", "b", "a" # скобки не обязательны
mytuple = ("s")       # s – строка!
mytuple = ("s",)      # ('s',) – кортеж
```

- Преобразование в кортеж

```
mytuple = tuple([0, 1, 7]) # (0, 1, 7)
```

- Можно изменять составляющие элементы

```
mytuple = ([1, 2, 3], [3, 2, 1])
mytuple[0][0] = 7 # ([7, 2, 3], [3, 2, 1])
```

Распаковка кортежа

- Распаковка кортежа

mytuple = ("a", "b", "c")
a, b, c = mytuple # a='a', b='b', c='c'
mytuple = ("a", "b", "c", "d", "e", "f")

Если при присваивании значений их окажется больше переменных – можно добавить в начало имени переменной (*) и ей будут присвоены остальные переменные.

first, second,*rest = mytuple
print(first, second, rest) # a b ['c', 'd', 'e', 'f']

Методы кортежей

```
[…'count', 'index']
mytuple = ("a", "b", "a")
mytuple.index("a") # 0 – первое вхождение
mytuple.count("a") # 2 – количество вхождений
```

Особенности кортежей

- Кортежи занимают меньше места, чем списки
- Кортежи можно использовать в качестве ключей словаря (см. дальше)
- Именованные кортежи могут служить более простой альтернативой объектам.
- Аргументы функции передаются как кортежи

Словари (Dictionary)

- Словарь похож на список. Но адресация элементов в нём обеспечивается идентификаторами-ключами.
- Ключ может являться булевой переменной, целым числом, числом с плавающей точкой, кортежем, строкой и другими объектами
- Словарь – изменяемый элемент. Можно добавлять, удалять и изменять его элементы.
- В Python допускается наличие запятой после последнего элемента списка, кортежа или словаря.
- В других языках программирования словари могут называться ключевыми массивами, ассоциативными массивами, хешами или хеш-таблицей

Создание словаря

- Словарь обозначается фигурными скобками {}

```
d = {}          # пустой словарь
d ={"Sub":"Hg", "Property":"Metal"}
```

"Sub", "Property" – ключи

```
d = {"key1": 1, "key2": 2} # {'key1': 1, 'key2': 2}
```

- Использование метода ***setdefault***

```
d.setdefault("key4", 5) # {'key1': 1, 'key2': 2, 'key4': 5}
d.setdefault("key1", 5) # {'key1': 1, 'key2': 2}
e = d.setdefault("key4", 5) # 5
e = d.setdefault("key1", 5) # 1
```

- Создание словаря по ключам

```
d = dict(sh="d", lng="di") # {'sh': 'd', 'lng': 'di'}
```

- С помощью метода ***fromkeys***

```
mydict = dict.fromkeys(["a", "b"], 1) # {'a': 1, 'b': 1}
d = dict.fromkeys (["a", "b"] # {'a': None, 'b': None}
```

Словари из списков и кортежей

- Создание словаря из списка списков

```
rawlist = [["a", "b"], ["c", "d"], ["e", "f"]]
d = dict(rawlist)      # {'a': 'b', 'c': 'd', 'e': 'f'}
```

из списка кортежей:

```
rawlist = [("a", "b"), ("c", "d"), ("e", "f")]
d = dict(rawlist) # {'a': 'b', 'c': 'd', 'e': 'f'}
```

из кортежа списков:

```
rawtuple = (["a", "b"], ["c", "d"], ["e", "f"])
d = dict(rawtuple) # {'a': 'b', 'c': 'd', 'e': 'f'}
```

Список строк

```
s = ["ab", "cd", "ef"]
d = dict(s)      # {'a': 'b', 'c': 'd', 'e': 'f'}
```

Кортеж строк

```
s = ("ab", "cd", "ef")
d = dict(s)      # {'a': 'b', 'c': 'd', 'e': 'f'}
```

Бесконечно вложенный словарь

```
a = {}
b = {}
a["b"] = b
b["a"] = a
print(a) # {'b': {'a': {...}}}
```

Методы словаря

- *[… 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']*

clear()	Очищает словарь.
copy()	Возвращает копию словаря.
fromkeys(seq[, value])	Создает словарь с ключами из seq и значением value (по умолчанию ***None***).
get(key[, default])	Возвращает значение ключа, а такого нет, не генерирует исключение, а возвращает ***default*** (по умолчанию ***None***).
items()	Возвращает пары (ключ, значение).
keys()	Возвращает ключи в словаре.
pop(key[, default])	Удаляет ключ и возвращает значение. Если ключа нет, то возвращает ***default*** (по умолчанию бросает исключение).
popitem()	Удаляет и возвращает пару (ключ, значение). Если словарь пуст, бросает исключение KeyError. (примечание: словари неупорядочены)
setdefault(key[, default])	Возвращает значение ключа, но если его нет, не бросает исключение, а создает ключ с значением ***default*** (по умолчанию ***None***).
update([other])	Обновляет словарь, добавляя пары (ключ, значение) из other. Существующие ключи перезаписываются. Возвращает ***None*** (не новый словарь!).
values()	Возвращает значения в словаре.

Примеры

d = {"key1": 1, "key2": 2}

 Замена значения

d["key1"] = 8 # {'key1': 8, 'key2': 2}

 Слияние словарей

e = {"key3": 3, "key1": 9} # второй словарь

d.update(e) # {'key1': 9, 'key2': 2, 'key3': 3}

 Удаление одного элемента словаря

d = {"key1": 1, "key2": 2}

del d["key1"] # {'key2': 2}

 Очистка словаря

d.clear() # {} Внимание! d.clear – не сработает!

 Проверка наличия

d = {"key1": 1, "key2": 2}

"key1" in d # True – такой ключ есть

2 not in d # True – такого ключа нет

Примеры

- Получение значения из словаря

```
d = {"key1": 1, "key2": 2}
d["key1"]                 # 1
d["key4"]                 # ОШИБКА
d.get("key1")               # 1
d.get("key4")               # None
d.get("key1", "default")    # 1
d.get("key4", "default")    # default
```

- Получение всех ключей

```
allkeys = list(d.keys())    # ['key1', 'key2']
```

- Получение всех значений

```
allvalues = list(d.values())  # [1, 2]
```

- Получение всех пар «ключ – значение»

Каждая пара будет возвращена как кортеж:

```
allpairs = list(d.items())  # [('key1', 1), ('key2', 2)]
```

Примеры

- Копирование

```
d = {"key1": 1, "key2": 2}
e = d          # передаётся ссылка!
d["key1"] = 9    # изменяет и словарь e
# d, e: {'key1': 9, 'key2': 2} {'key1': 9, 'key2': 2}
e = d.copy()   # создаём копию словаря
d["key1"] = 9
# d, e: {'key1': 9, 'key2': 2} {'key1': 1, 'key2': 2}
```

- Извлечение элемента из словаря

```
e = d.pop("key1") # d, e: {'key2': 2} 1
e = d.pop("key4", "d") # d,e: {'key1': 1, 'key2': 2} d
d = {"key1": 1, "key2": 2, "key3": 3, "key4": 4}
e = d.popitem()
# d,e: {'key1': 1, 'key2': 2, 'key3': 3} ('key4', 4)
```

Функция defaultdict

- Функция определяет значение по умолчанию для новых ключей при создании словаря
- Аргументом ***defaultdict()*** является функция, возвращающая значение для отсутствующего ключа

```
from collections import defaultdict
def fun():     # Функция для значения по умолчанию
    return "Что?"
d = defaultdict(fun)
d["key1"] = "A"   # добавляем элемент
d["key2"] = "B"   # добавляем элемент
e = d["key3"]   # запрашиваем отсутствующий элемент. В ответ получаем: "Что?"
```

- Допустимо использовать функции ***int(), list(), dict()***, чтобы возвращать пустые значения по умолчанию**: int()** возвращает ***0***, ***list()*** – пустой список ***([])***, а ***dict()*** – пустой словарь ***({})***. Если опустить аргумент, исходное значение нового ключа будет ***None***.

```
e = defaultdict(lambda: "Что?") # лямбда-функция
```

Множества (Set)

Множество похоже на словарь, но имеет только ключи, а значения опущены.

Ключи должны быть уникальными.

Порядок ключей не имеет значения.

- Создание пустого множества

empty_set = set()

Примеры создания множеств:

set("text") ***# {'t', 'e', 'x'}***

- из списка

d = set(["Раз", "Два", "Два", "Три"])

 # {'Два', 'Раз', 'Три'} (порядок может меняться)

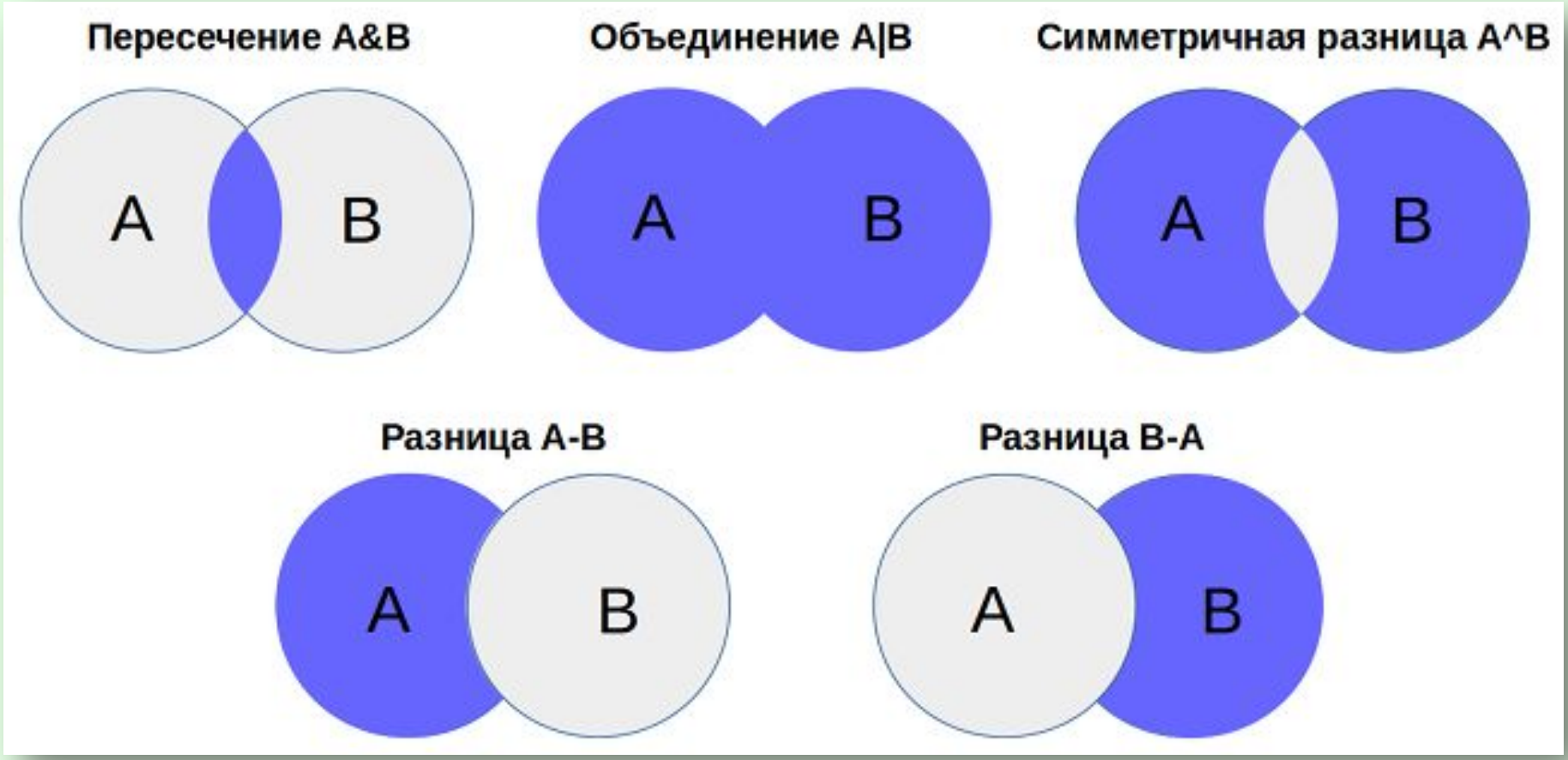
- из кортежа

d= set(("Раз", "Два", "Два", "Три"))

 # {'Два', 'Раз', 'Три'} (порядок может меняться)

d= set({"a":1, "b": 2, "c": 3}) # {'c', 'b', 'a'}

Пересечение, объединение и разница множеств. Графическое представление



Комбинации и операторы

```
d = set(["a", "b", "c"])
e = set(["c", "d", "e"])
```

- пересечение множеств (**&**, **intersection**):

```
f =  d & e              # {'c'}
g = d.intersection(e)   # {'c'}
```

- объединение множеств (**|**, **union**)

```
f =  d | e              # {'b', 'a', 'd', 'c', 'e'}
g = d.union(e)          # {'b', 'a', 'd', 'c', 'e'}
```

- Разность множеств (члены только первого множества, но не второго) (**–**, **difference**)

```
f =  d – e              # {'a', 'b'}
g = d.difference(e)     # {'a', 'b'}
```

Комбинации и операторы

```
d = set(["a", "b", "c"])
e = set(["c", "d", "e"])
```

 исключающее ИЛИ (элементы или первого, или второго множества, но не общие)

(^ , **symmetric_difference()**)

```
f =  d ^ e                          # {'b', 'd', 'e', 'a'}
g = d.symmetric_difference(e)       # {'b', 'd', 'e', 'a'}
```

Свойства множеств

- *[…'add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection', 'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'symmetric_difference', 'symmetric_difference_update', 'union', 'update']*
- Длина множества

```
d = set(["a", "b", "c"])
len(d) # 3
```

Свойства множеств

a.isdisjoint(b)	Истина, если ***a*** и ***b*** не имеют общих элементов.
a == b	Все элементы ***a*** принадлежат ***b***, все элементы ***b*** принадлежат ***a***.
a.issubset(b) ***a <= b***	Все элементы ***a*** принадлежат ***b***.
a.issuperset(b) ***a >= b***	Все элементы ***b*** принадлежат ***a***.
a.union(b, ...) ***a | b | ...***	Объединение нескольких множеств.
a.intersection(b, ...) ***a & b & ...***	Пересечение.
a.difference(b, ...) ***a – b – …***	Множество из всех элементов ***a***, не принадлежащие ни одному из ***b***.
a.symmetric_difference(b) ***a^ b***	Множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих.
a.copy()	Копия множества.
a.update(b, ...) ***a |= b | ...***	Объединение.

Свойства множеств

a.intersection_update(b, ...); a &= b & ...	Пересечение.
set.difference_update(other, ...); set – = other | ...	Вычитание.
a.symmetric_difference_update(b); set ^= other	Множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих.
a.add(element)	Добавляет элемент в множество.
a.remove(element)	Удаляет элемент из множества. Возникает исключение **KeyError**, если такого элемента не существует.
a.discard(element)	Удаляет элемент, если он находится в множестве.
a.pop()	Удаляет первый элемент из множества. Так как множества не упорядочены, нельзя точно сказать, какой элемент будет первым.
a.clear()	Очистка множества.

Отношения между множествами

- Является ли одно множество подмножеством другого (все члены первого множества являются членами второго) (<=, ***issubset***)
- Проверка, на то, что одно множество является надмножеством второго *(>=, **issuperset()***)

```
d = set(["a", "b", "c"])
h = set(["a", "b"])
g1 = h.issubset(d)# True   То же: g1 = h<=d
g2 = h.issuperset(d)  # False To же: g2 = h>=d
g3 = d.issubset(h)     # False  To же: g3 = d<=h
g4 = d.issuperset(h)  # True   To же: g4 = h<=d
```

- Собственное подмножество (<). Эквивалентно ***A <= B and A != B***
- Собственное множество множеств (>) Эквивалентно ***A >= B and A != B***

Действия над множествами

- Добавление элемента во множество:

```
d = set(["a1", "b2", "c3"])
d.add("d4")        # {'a1', 'b2', 'c3', 'd4'}
```

- Проверка наличия элемента:

```
e = "a" in d       # False
e = "b2" in d      # True
```

- Удаление элемента из множества:

```
d.remove("a1")  # {'b2', 'c3', 'd4'}
d.discard("b2")  # {'c3', 'd4'}
```

- Метод ***discard()*** не выдает ошибку, если элемента нет во множестве в отличие от метода ***remove()***.

Тип frozenset

- ***set*** – изменяемый тип данных, а ***frozenset*** – нет. Примерно схожая ситуация со списками и кортежами.

```
b = frozenset("текст")
b.add(1)          # ОШИБКА
```

Бинарные типы данных

- В Python 3 появились последовательности восьмибитных целых чисел, имеющих возможные значения от **0** до **255**. Они могут быть двух типов:
  - ✔ **bytes** – неизменяем, как кортеж байтов
  - ✔ **bytearray** – изменяем, как список байтов

Тип bytes – неизменяемая последовательность байтов

- объекты типа ***bytes*** являются последовательностями коротких целых чисел, каждое из которых имеет значение в диапазоне от **0** до **255**, которые могут выводиться как символы ASCII.
- Этот тип поддерживает обычные операции над последовательностями и большинство строковых методов, доступных для объектов типа ***str***
- ***bytes*** – можно назвать кортежем байтов

Представление значений типа bytes

- Представление значения типа ***bytes*** начинается с символа ***b*** и кавычки. Далее следуют шестнадцатеричные последовательности или символы ASCII. Завершается конструкция тоже символом кавычки.

Примеры:

```
b"\x61"          # b'a'
b"\x01abc\xff"   # b'\x01abc\xff'
b = [1, 0, 3, 255]
bt = bytes(b)        # b'\x01\x00\x03\xff'
print(bt[2])     # 3
bt[1] = 1        # ОШИБКА!
```

Применение типа bytes

- тип ***bytes*** не поддерживает метод ***format*** и оператор % форматирования, и нельзя смешивать и сопоставлять объекты типов ***bytes*** и ***str***, не выполняя явное преобразование.
- для представления **текстовых** данных в подавляющем большинстве случаев используются объекты типа ***str*** и текстовые файлы, а для представления **двоичных** данных – объекты типа ***bytes*** и двоичные файлы

bytearray – изменяемая последовательность байтов

```
b = [1, 0, 3, 255]
ba = bytearray(b)
ba[1] = 7        # bytearray(b'\x01\x07\x03\xff')
```

При выводе на экран переменных типа ***bytes*** или ***bytearray*** используется формат ***\x xx*** для непечатаемых байтов и их эквиваленты ASCII для печатаемых (за исключением распространенных управляющих последовательностей вроде ***\n*** вместо ***\x0a***).

Библиотеки для работы с бинарными данными

- Стандартная библиотека содержит модуль ***struct***, который обрабатывает данные аналогично структурам в C/C++.
- С помощью этого модуля можно преобразовать бинарные данные в структуры данных Python и наоборот.
- Другие библиотеки для работы с бинарными данными
  - ✔ bitstring (http://bit.ly/py-bitstring);
  - ✔ construct (http://bit.ly/py-construct);
  - ✔ hachoir (http://bit.ly/hachoir-pkg);
  - ✔ binio (http://spika.net/py/binio/).

Прочие типы данных

- ***NoneType*** – объект со значением ***None***
- ***NotImplemented*** **–** объект этого типа возвращается при сравнении несравнимых объектов.
- ***Ellipsis (…)*** **–** это объект, который может появляться в нотации среза многомерного массива (например при использовании библиотеки NumPy).

a[5,:,:,1] эквивалентен ***a[5,...,1]***

См. также:

http://rupython.com/python-ellipsis-706.html

https://habr.com/ru/post/123821/

Спасибо за внимание