Лекция 3

Списки и кортежи

Словари и множества

Списки и кортежи

- Списки и кортежи это *коллекции*, обладающие свойством *упорядоченности*, то есть это *последовательности*
- Списки *изменяемы* (mutable), класс списка **list**, литерал списка использует квадратные скобки [ ]
- Кортежи *неизменяемы* (immutable), класс кортежа **tuple**, литерал кортежа использует круглые скобки ( )
- Списки и кортежи соотносятся друг с другом приблизительно также, как классы bytearray и bytes
- Полной взаимозаменяемости между списками и кортежами нет, например при форматировании строки оператором % его правым операндом может быть кортеж, но не список
- При записи литерала кортежа единичной длины для разрешения двусмысленности используется запятая:
  - (27) - число в скобках, оператор выделения подвыражения
  - (27,) - кортеж содержащий один элемент

Особенности реализации

- Списки и кортежи реализованы как массив ячеек одинакового размера в которых хранятся *ссылки* на объекты-элементы
  - массив ссылок обеспечивает быстрый доступ к элементу по его индексу
  - малый размер ячейки позволяет с достаточно высокой производительностью осуществлять вставку и удаление элементов
- Количество ячеек и адреса объектов, хранящиеся в ячейках, для *кортежа* величина постоянная
- В *списке* могут изменяться как количество ячеек, так и адреса объектов, хранящиеся в них
- В списках и кортежах могут быть смешаны элементы разных типов, то есть в ячейке может быть ссылка на любой объект

Создание объекта

- Литерал

  Литерал типов *list* и *tuple* это последовательность элементов любого типа, разделенных запятыми и заключенных в скобки

```
m = [ 'abcdef', 14, 27.12, [ 1, 2 ] ] # Литерал списка,  скобки [ ]
c = ( 'abcdef', 14, 27.12, [ 1, 2 ] ) # Литерал кортежа, скобки ( )
```

- Встроенная функция

  Встроенные функции *list* и *tuple* воспринимают единственный параметр-коллекцию с элементами любого типа

```
m1 = list(('abcdef', 14, 27.12, [ 1, 2 ]))
c1 = tuple(['abcdef', 14, 27.12, [ 1, 2 ]])
m2 = list(range(1, 5)) # => [1, 2, 3, 4]
c2 = tuple(range(1, 5)) # => (1, 2, 3, 4)
```

- Для списков, как и для bytearray, реализован метод copy

```
m3 = m1.copy()
```

Подробнее о литералах

- *Литерал списка или кортежа* это заключенная в скобки последовательность элементов *разделенных запятыми*
- Последний элемент также может завершатся запятой
- Элемент литерала это выражение
- Литерал списка или кортежа это элементарное выражение
- Между открывающей и закрывающей скобками литерала везде где есть пробельный символ допустим перенос строки

```
z = [
    'Test',
    a + b,
    math.sin(x) /
    math.cos(y),
    [12, math.sqrt(a ** 2 + b ** 2)]
]
```

- Скобки литерала кортежа можно опустить, если при этом не возникает двусмысленности

Создание объекта, строки и байты

- Строки и байты это коллекции, они могут быть параметром функций list и tuple

```
ms = list('abcdef')   # => ['a', 'b', 'c', 'd', 'e', 'f']
cs = tuple('abcdef')  # => ('a', 'b', 'c', 'd', 'e', 'f')
mb = list(b'abcdef')  # => [97, 98, 99, 100, 101, 102]
cb = tuple(b'abcdef') # => (97, 98, 99, 100, 101, 102)
```

- Обратное преобразование для байт

```
bm = bytes(mb)     # => b'abcdef'
am = bytearray(mb) # => bytearray(b'abcdef')
bc = bytes(cb)     # => b'abcdef'
ac = bytearray(cb) # => bytearray(b'abcdef')
```

- Обратное преобразование для строк не работает

```
sm = str(ms) # Результатом будет удобное для восприятия содержание
sc = str(cs) # списка или кортежа со скобками, кавычками и запятыми
```

Метод join

- Для соединения списка или кортежа, содержащего строки используется метод *join*. Если разделитель не нужен, в качестве разделителя используется пустая строка.

```
sm = ''.join(ms) # => 'abcdef'
sc = ''.join(cs) # => 'abcdef'
```

- Обратное преобразование методом join для байт и байтовых массивов не работает:

```
bm = b''.join(mb) # Ошибка, mb содержит числа, а не байты
```

Таким образом строку, байты или байтовый массивов можно превратить в список или кортеж, произвести над ними нужные действия, и затем сделать обратное преобразование

Операторы [ ], + и *

- Операторы [ ], + и * применимы к последовательностям. Для списков и кортежей они работают также, как для строк и байтов.
- Оператор [ ] позволяет выбирать как отдельные элементы, так и фрагменты, в том числе с шагом
- Оператор [ ] примененный к списку может стоять слева от символа присваивания, *список изменяем*
- Для оператора + *типы list и tuple не являются совместимыми*. Для соединения объектов двух разных типов требуется явное преобразование.

```
m = [ 'abcdef', 14, 27.12, [ 1, 2 ] ]
c = ( 'abcdef', 14, 27.12, [ 1, 2 ] )
m2 = m + m        # Правильно
m2 = m + c        # Ошибка
m2 = m + list(c)  # Правильно, результат list
c2 = c + tuple(m) # Правильно, результат tuple
c3 = c * 3 # Кортеж, в котором три раза повторены элементы из c
```

Создание объекта используя фрагмент

- Фрагмент последовательности в правой части инструкции присваивания создает новый объект
- Копия оригинальной последовательности

```
m = ['abcdef', 14, 27.12, [ 1, 2 ]]
c = ('abcdef', 14, 27.12, [ 1, 2 ])

m2 = m[:] # => ['abcdef', 14, 27.12, [1, 2]]
c2 = c[:] # => ('abcdef', 14, 27.12, [1, 2])
```

- Копия части последовательности

```
m3 = m[1:3] # => [14, 27.12]
c3 = c[1:3] # => (14, 27.12)
```

Переменная как элемент списка

- Список содержит *ссылки* на объекты
- При передаче в список переменной в ячейку списка попадает ссылка на тот же объект, на который ссылается переменная
  - в ячейке списка *не копия объекта*
  - в ячейке списка *не ссылка на переменную*
- Пример:

```
a = 18
b = 'b_string'
c = [ 1, 2 ]
y = [ 'abcdef', a, b, c ]
print(y) # => [ 'abcdef', 18, 'b_string', [1, 2] ]
a = 33
b = 'b_modified'
c[1] = 433
c = [ 8, 9 ]
print(y) # => [ 'abcdef', 18, 'b_string', [1, 433] ]
```

Переменная как элемент кортежа

- Кортеж, как и список, содержит *ссылки* на объекты
- В кортеж может быть помещен объект изменяемого композитного типа. Такой объект *остается изменяемым*.
- Пример

```
a = 18
b = 'b_string'
c = [ 1, 2 ]
y = ( 'abcdef', a, b, c )
print(y) # => ( 'abcdef', 18, 'b_string', [1, 2] )
a = 33
b = 'b_modified'
c[1] = 433
c = [ 8, 9 ]
print(y) # => ( 'abcdef', 18, 'b_string', [1, 433] )
```

- В результате всех операций измененным в кортеже или списке оказался только объект изменяемого композитного типа

Инструкция del

- Инструкция *del* удаляет переменную
- Переменная - это связь объекта с программой,
  это способ обратится к объекту
- Если объект утратил все свои связи, он уничтожается
  - локальные переменные удаляются автоматически
    с завершением их области видимости
- Ячейки списка или кортежа можно рассматривать как переменные ссылающиеся на объекты, и имеющие вместо имени индекс
- Ячейки списка можно удалять инструкцией del, индивидуально или в виде фрагментов. Размер списка при этом уменьшается.
- При удалении списка или кортежа вместе с ним удаляются все содержащиеся в нем ячейки-переменные

Операции изменяющие список

- *Список изменяем*, кортеж нет. Приведенные ниже операции к кортежем неприменимы
- Присваивание элементу списка нового значения

```
m = [ 'abcdef', 14, 27.12, [ 1, 2 ] ]
m[2] = 433 # => [ 'abcdef', 14, 433, [ 1, 2 ] ]
```

- Замена фрагмента

```
m[1:3] = [ 1.1, 'two', 3 ] # => ['abcdef', 1.1, 'two', 3, [1, 2]]
```

- Удаление фрагмента

```
del m[1:3]  # Инструкция del
m[1:3] = [] # Присваивание пустого списка
```

- Удаление элемента

```
del m[2]    # верно, элемент удален инструкцией del
m[2] = []   # НЕВЕРНО, элемент заменен пустым списком
m[2:3] = [] # Так правильно, замена фрагмента из одного элемента
```

Удаление всех элементов списка

- Метод clear

```
m = [ 'abcdef', 14, 27.12, [ 1, 2 ] ]
m.clear()
print(m) # => []
```

- Инструкция del

```
m = [ 'abcdef', 14, 27.12, [ 1, 2 ] ]
del m[:]
print(m) # => []
```

Извлечение элемента из списка

- Метод pop() работает для списков точно так же, как для байтовых массивов
  - без параметра извлекается последний элемент
  - параметр является индексом элемента, который необходимо извлечь
  - метод возвращает извлеченный элемент

```
e = m.pop()   # Извлечение последнего элемента
e = m.pop(3)  # Извлечение третьего элемента
e = m.pop(-3) # Извлечение третьего элемента с конца
```

- Извлеченный элемент *удаляется* из списка, список становится на один элемент короче

Добавление элементов в список

- Добавление элемента в конец списка

```
m.append('New') # добавить элемент 'New'
```

- Добавление последовательности в конец списка

```
m.extend('New') # добавить три элемента: 'N', 'e', 'w'
```

- Вставка элемента в позицию с индексом i (i целое число), элементы от позиции i и далее сдвигаются вправо

```
m.insert(i, 'at_i')
```

- *Список не является разреженным массивом,* при обращении к элементу за пределами массива:

```
m[99] = 433       # => Вызывает ошибку
m.insert(99, 433) # => Добавляет элемент в конец списка
```

- Функция len() позволяет определить количество элементов

Стек и очередь

- Сочетание методов append и pop позволяют использовать список как *стек*. Метод pop извлекает элементы из списка в порядке обратном тому, в котором они были помещены в список методом append (буфер LIFO).
- *Очередь* реализована в классе deque модуля collections
- Очередь позволяет извлекать элементы с другой стороны, в этом случае элементы извлекаются в порядке поступления (буфер FIFO)
- Класс *deque* дополняет список новыми методами:

```
appendleft(x)   # Добавить элемент x в начало списка
extendleft(seq) # Добавить последовательность seq в начало списка
popleft()       # Извлечь элемент в начале списка
rotate(n=1)     # Кольцевой сдвиг вправо (n>0) или влево (n<0)
```

- Таким образом deque это *двойная очередь*, где и добавление и извлечение элементов возможно с обоих концов

Поиск и удаление найденного

- Метод count(x) возвращает количество элементов x в списке или кортеже
- Метода find() у списков и кортежей нет
- Метод index() это полный аналог метода index() для строк и байт
- Синтаксис:

```
m.index(x[, start[, end]])
```

  - идет поиск элемента со значением x начиная с индекса start и до индекса end, не включая его
- Метода replace() нет, но есть метод remove()

```
m.remove(x)
```

  - из списка удаляется первый найденный элемент x
- Методы не имеют правосторонних вариантов, производится поиск первого элемента слева
- Если элемент не найден, методы index и remove вызывают ошибку

Оператор in

- Оператор *in* служит для проверки утверждения: *Присутствует ли элемент X в коллекции Y*
- Результат оператора in логическое значение True или False
- Синтаксис:

```
X in Y
```

- Примеры:

```
m = [ 'abcdef', 14, 27.12, [ 1, 2 ] ]
c = ( 'abcdef', 14, 27.12, [ 1, 2 ] )

[ 1, 2 ] in m # => True
   27.13 in m # => False
'abcdef' in c # => True
'abcde'  in c # => False
```

Операторы сравнения

- Списки и кортежи можно сравнивать используя операторы <, >, ==, <= и >=
- Сравнение происходит поэлементно, решение принимается как результат сравнения первых двух несовпадающих элементов
- Элементы имеющие равные индексы должны быть сравнимы, то есть *автоматически* приводятся к одному типу

```
m1 = [ 'abcdef', 14, 27.12, [ 1, 2 ] ]
m2 = [ 'abcdef', 14, 28, [ 1, 2 ] ]
print(m2 > m1) # => True
m3 = [ 'abcdef', 14, '28', [ 1, 2 ] ]
print(m2 > m1) # => Ошибка, нельзя сравнивать строку и число
```

- Нельзя сравнивать список и кортеж, в этом случае требуется явное преобразование типа

```
c1 = ( 'abcdef', 14, 27.12, [ 1, 2 ] )
print(m1 >= c1)        # => Ошибка
print(tuple(m1) >= c1) # => True
print(m1 >= list(c1))  # => True
```

Равенство и идентичность

- Проверку на идентичность осуществляет оператор *is*
- Пример: две переменные указывающие на один и тот же объект

```
m = [ 1, 'two', 3.3 ]
n = m
n[1] = 122
print(m) # => [1, 122, 3.3]
print(n) # => [1, 122, 3.3]
print(n == m, n is m) # => True, True
```

- Пример: две переменные указывающие на разные объекты имеющие одинаковое содержание

```
m = [ 1, 'two', 3.3 ]
n = [ 1, 'two', 3.3 ]
print(n == m, n is m) # => True, False
```

- Пример: то же, но для кортежей

```
m = ( 1, 'two', 3.3 )
n = ( 1, 'two', 3.3 )
print(n == m, n is m) # => True, True - оптимизатор соединил объекты в один
```

Сортировка и инверсия

- Методы *изменяющие* список
  - Сортировка: `m.sort()`
  - Инверсия: `m.reverse()`
- Встроенные функции *создающие новый* список или кортеж
  - Сортировка: `n = sorted(m)`
  - Инверсия: `n = reversed(m)`
- Функция и метод сортировки применимы к последовательностям все элементы которых сравнимы, то есть *автоматически* приводятся к одному типу, например:
  - числа, включая Decimal и Fraction, но не complex
  - строки
- Функция и метод сортировки имеют два именованных параметра
  - key - функция вычисления ключа сортировки
  - reverse - если True, то сортировка в обратном порядке

Строки, списки, кортежи сходство и различие

- Кортежи как и строки неизменяемы
- Кортежи как и списки могут содержать элементы различных типов
- И строки и списки и кортежи - последовательности (sequence), и как следствие - перечислимые типы (enumerable)

Словари и множества

Словари

- *Словарь* это ассоциативный массив или хеш, то есть коллекция содержащая пары "ключ: значение". Класс словаря **dict**.
- Словарь имеет свойства:
  - перечислимость (enumerable)
  - изменяемость (mutable)
- *Словарь это не последовательность*
- Нет операций с фрагментами (slice)
- Словарь подобен списку, но доступ к его элементам осуществляется не по индексу, а по ключу
- Ключ должен иметь тип, допускающий вычисление хеш-функции, то есть обладать свойством хешируемости (hasheable)

Хеш-функция

- *Хеш-функция*, это функция, параметром которой является объект, а результатом целое число, служащее индексом ячейки памяти во внутреннем массиве, где будет размещено связанное с объектом значение, для которого объект является ключом
  - это упрощенная модель, реальные ассоциативные массивы реализованы на базе сложных высокооптимизированных алгоритмов
- Только неизменяемый объект хешируем
  - если объект-ключ изменится, изменится и его индекс во внутреннем массиве и он не сможет быть найден в словаре
- Композитный объект хешируем только в том случае если все его элементы хешируемы

Множества

- *Множество* это частный случай словаря, элементы которого имеют ключ, но не имеют значения. Другими словами, ключ сам по себе является значением. Класс множества **set**.
- Множество как и словарь имеет свойства:
  - перечислимость (enumerable)
  - изменяемость (mutable)
- Множество как и словарь не является последовательностью и не поддерживает операции с фрагментами
- В отличие от словарей, для множеств определены теоретико-множественные операции

Создание объекта

- Литерал

  Литерал типов *dict* и *set* это последовательность элементов, разделенных запятыми и заключенных в фигурные скобки

```
d = { 'name': 'abcd', 'f1': 14, 5: 27.12, 'pair': [ 1, 2 ] }
d = { 'name': 'abcd', 'f1': 14, 5: 27.12, [ 1, 2 ]: 'pair' } # Ошибка
s = { 'abcd', 14, 27.12, ( 1, 2 ) }
s = { 'abcd', 14, 27.12, [ 1, 2 ] } # Ошибка
d = {} # Пустой словарь, пустое множество создается функцией set()
```

- Встроенная функция

```
d2 = dict(d)
d3 = dict(name='abcd', f1=14, digit_5=27.12, pair=[ 1, 2 ])
s2 = set(s)
s3 = set(['abcd', 14, 27.12, ( 1, 2 )])
```

- И словари и множества имеют метод copy.
  Наличие метода copy это свойство изменяемых коллекций.

Доступ к элементу

- Для доступа к элементу словаря используется оператор [ ]. Синтаксис такой же, как для списков, но операндом является ключ, а не индекс.
- Примеры:

```
d = { 'name': 'abcd', 'f1': 14, 5: 27.12, 'pair': [ 1, 2 ] }

# Доступ к элементу
value = d['f1']     # Величина, хранящаяся в словаре по ключу 'f1'
d['f1'] = new_value # Изменение величины по ключу 'f1'
d['f2'] = new_val2  # Создание нового элемента если такого ключа нет

# Если выбирается элемент по несуществующему в словаре ключу
d['ab']          # Ошибка
d.get('ab')      # => возвращает None, ошибки нет
d.get('ab', '?') # => возвращает '?' - значение по умолчанию

# Удаление элемента
del d['f1'] # Удалить элемент с ключом 'f1',
            # ошибка если такого элемента нет
```

Элемент множества

- Для множеств понятие "доступа к элементу" лишено смысла; можно либо проверить принадлежность элемента к множеству, либо удалить элемент из множества.
- Примеры:

```
s = { 'abcd', 14, 27.12, ( 1, 2 ) }

14 in s          # => True, 14 это элемент множества s
s.remove(14)     # Удалить элемент, ошибка если такого элемента нет
s.discard(27.12) # Удалить элемент если такой есть
```

- *Оператор [ ] к множествам неприменим.*
  Добавление элемента к множеству производится методом *add*:

```
s.add('New element') # add() применим к множествам, но не к словарям !
```

- Оператор *in*, примененный к словарю, проверяет наличие *ключа* в словаре

```
d = { 'name': 'abcd', 'f1': 14, 5: 27.12, 'pair': [ 1, 2 ] }
'f1' in d  # => True, словарь d содержит элемент с ключом 'f1'
```

Методы словарей

```
d = { 'name': 'abcd', 'f1': 14, 5: 27.12, 'pair': [ 1, 2 ] }
```

- Преобразование словаря в итерируемый объект. При итерации по коллекции-результату порядок элементов не определен.

```
d.keys()   # Возвращает коллекцию ключей
d.values() # Возвращает коллекцию значений
d.items()  # Возвращает коллекцию пар (ключ, значение)
```

- Добавление элементов в словарь

```
d.update(other_dict) # Добавить элементы из другого словаря
d.update(color='Green', count=27) # или из переданных параметров
```

- Извлечение элементов из словаря

```
d.pop('name')    # => 'abcd' - извлечение элемента с удалением
d.pop('ab')      # => такого ключа нет, вызывает ошибку !
d.pop('ab', '?') # => '?' - извлечение элемента или возврат второго
                 # параметра, если ключ не найден (как в методе get)
```

Методы словарей

- Метод *popitem* возвращает элемент словаря в виде пары (ключ, значение) и удаляет этот элемент из словаря. *Начиная с версии 3.7 popitem извлекает элементы словаря в порядке обратном их поступлению (LIFO)*.

```
d = { 'name': 'abcd', 'f1': 14, 5: 27.12, 'pair': [ 1, 2 ] }
e = d.popitem()
print(e) # => ('pair', [1, 2])
```

  - Для множеств аналогичную операцию выполняет метод *pop*. Метод pop извлекает один заранее не определенный элемент множества. Какой либо порядок не предусмотрен.
- Метод *clear* удаляет все элементы, он применим как к словарям, так и к множествам

```
d = { 'name': 'abcd', 'f1': 14, 5: 27.12, 'pair': [ 1, 2 ] }
s = { 'abcd', 14, 27.12, ( 1, 2 ) }
d.clear()
s.clear()
```

Словарь как разреженный массив

- С помощью словарей можно имитировать разреженные массивы
  - Одномерный массив - в качестве ключей используются целые числа

```
a = {}
a[20] = 1
a[202020] = 2
print(a[20]) # => 1
print(a) # => {20: 1, 202020: 2}
```

  - Многомерный массив - в качестве ключей используются кортежи

```
b = {}
b[(1,2,3)] = 123
b[4,5,6] = 456
print(b[1,2,3]) # => 123
print(b) # => { (1, 2, 3): 123, (4, 5, 6): 456 }
```

Форматный вывод словаря в строку

- Метод *класса строк* format_map() позволяет сформировать строку, в которую встроены значения величин из словаря
- Для выборки элемента в спецификации формата используется ключ словаря
- Пример

```
d = { 'name': 'abcd', 'f1': 14, 5: 27.12, 'pair': [ 1, 2 ] }
s = 'My name is {name}, Pair value is {pair}'.format_map(d)
print(s) #=> 'My name is abcd, Pair value is [1, 2]'
```

- Ключами в спецификации формата могут быть только строки
- Необязательно использовать все ключи

Словарь это универсальное хранилище информации.
Данные в словаре разделены на части и снабжены именами.
Словарь можно передавать как единое целое и в то же время использовать данные из отдельных его частей индивидуально.

Теоретико-множественные операции

- Операции над множествами (но не над словарями !):

```
s1 & s2 # => Пересечение
s1 | s2 # => Объединение
s1 - s2 # => Разность
s1 ^ s2 # => Симметрическая разность (XOR)
s1 > s2 # => Проверка на надмножество
s1 < s2 # => Проверка на подмножество
```

- Метод словаря keys() возвращает коллекцию, которая имеет свойства множества
- Пример:

```
d = { 'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5 }
blacklist = { 'b', 'c', 'd' }
allowed_keys = d.keys() - blacklist # => {'e', 'a'}
```

Теоретико-множественные методы

- Теоретико-множественные методы подобны операциям, но допускают в качестве параметра объекты произвольного перечислимого типа. *Все элементы объекта-параметра должны быть хешируемы*.
- Примеры:

```
s = { 'abcd', 14, 27.12, ( 1, 2 ) }

s.intersection([27.12, 'abcd', 816]) # => Пересечение
s.union('abc') # => Объединение s с множеством ('a','b','c')
s.difference((14, 'abcd')) # => Разность
s.symmetric_difference((14, 'New')) # => Симметрическая разность XOR
s.issuperset([27.12, 'abcd']) # => Проверка на надмножество
s.issubset((101, 'New')) # => Проверка на подмножество
```

Тип frozenset

- Тип *frozenset* является парным неизменяемым типом к типу set
- Тип frozenset в отличие от set хешируемый (hasheable)
- В состав set нельзя включить элемент типа set, но можно типа frozenset
- Объекты типа frozenset могут быть ключами элементов множества
- Примеры:

```
f = frozenset(('abcd', 18, 27.12, ( 3, 4 )))

f.add('New')        # Ошибка, нельзя добавлять элементы
g = set('abc')
g.add('New')        # Правильно
s = { 1, 'two', f } # Правильно
t = { 1, 'two', g } # Вызывает ошибку !
e = { f: 18 }       # frozenset может быть ключом в словаре
x = { g: 18 }       # Ошибка: unhashable type: 'set'
```

Заключительные замечания

- Списки и кортежи это последовательности, они обеспечивают быстрый доступ к элементу по его порядковому номеру
- Кортеж это неизменяемый вариант списка
- Словари и множества это коллекции, но не последовательности. Они обеспечивают быстрый доступ к элементу по его значению для множеств и по его ключу для словарей.
- Ключи словаря и элементы множества должны быть хешируемыми
- Множество это частный случай словаря, где ключ сам по себе является значением
- Тип frozenset это неизменяемый вариант множества
- Тип frozendict (PEP 416) ... есть в репозитории PyPi
- В контексте итерации словарь это коллекция ключей