Методические указания

Урок 5.2 Коллекции и их методы

**Задачи урока:**

* Глубже познакомиться со списками
* Познакомиться с кортежами

**0. Подготовка к уроку**

До начала урока преподавателю необходимо:

1. Просмотреть, как ученики справились с домашним заданием
2. Прочитать методичку

**1****. Основы работы со списками**

**Учитель:** Напомню, списки представляют собой наборы данных, как правило, но не всегда, однотипных. Сегодня мы пойдем дальше и поговорим о сходстве списков и строк.

Действительно, строки и списки очень похожи, ведь по сути строки — наборы символов, как бы подмножества списков. Но если списки могут содержать абсолютно разные элементы, скажем, целые числа, вещественные числа, строки, да и любой другой тип данных, то строки всегда содержат только символы.

Хорошая новость заключается в том, что, поскольку строки — частный случай списков, то весь функционал для строк доступен и при работе с списками. Это круто, своего рода двукратное использование наших знаний, а точнее, их адаптация под новый материал. Конечно у списков и строк есть и отличия.

**Сходства:**

* функция len() для подсчета длины списка, как у строк;
* оператор принадлежности in, как и у строк;
* индексация, то есть, обращение по индексу. Также начинается с нуля, и возможна отрицательная индексация с конца списка, как и у строк;
* срезы позволяют получать подсписки исходных списков, как у строк;
* операция конкатенации и умножения на число, работает для списков также, как и для строк;
* функции min() и max() со списками работают отлично, и позволяют не писать самостоятельно код поиска наибольшего и наименьшего элемента списка;

**Различия:**

* списки **изменяемы**, иногда их называют мутирующими коллекциями;
* функция sum(), находящая сумму элементов списков, отсутствует у строк.

**2****. Методы списков**

**Учитель**: При изучении строк мы говорили о строковых методах. Напомню, методы это специальные функции, применимые к определенным типам данных. В частности, строковые методы вызываются именно для переменных, обозначающих строки. Метод, по сути, есть функция для выполнения часто встречающейся задачи.

Методы есть и у списков. Если вспомните предыдущие уроки, то, возможно, заметите, что мы всегда работали с уже созданным списком, другими словами, со статическим списком. Но ведь нужно уметь и конструировать списки самостоятельно. К примеру, мы можем считать данные от пользователя и добавить их в список. Или удалить какие-то элементы из списка.

Основные методы списков:

* append - добавить в конец списка
* remove - удалить первый совпадающий элемент списка
* pop - удалить элемент по индексу с возможностью сохранить значение в переменной
* clear - очистить список
* sort - отсортировать список
* extend - объединить списки
* index - найти индекс элемента
* insert - вставить элемент по индексу

Конечно же, это не все методы, которые существуют(о них я вам советую почитать отдельно). Давайте рассмотрим как они работают на практике

1. **append**

| a = [1, 2, 3]  a.append(4)  print(a) |
| --- |

Результат

[1, 2, 3, 4]

1. **remove**

| a = [1, 2, 3]  a.remove(2)  print(a) |
| --- |

Результат

[1, 3]

1. **pop**

| a = [1, 2, 3]  a.pop(2)  print(a) |
| --- |

Результат

[1, 2]

| a = [1, 2, 3]  b = a.pop(2)  print(a)  print(b) |
| --- |

Результат

[1, 2]

3

1. **clear**

| a = [1, 2, 3]  a.clear()  print(a) |
| --- |

Результат

[]

1. **sort**

| a = [2, 1, 3]  a.sort()  print(a) |
| --- |

Результат

[1, 2, 3]

1. **extend**

| a = [1, 2, 3]  b = [3, 5, 6]  a.extend(b)  print(a) |
| --- |

Результат

[1, 2, 3, 3, 5, 6]

1. **index**

| a = [2, 1, 3]  print(a.index(3)) |
| --- |

Результат

2

1. **insert**

| a = [2, 1, 3]  a.insert(0, 4)  print(a) |
| --- |

Результат

[4, 2, 1, 3]

**4****. Кортежи**

**Учитель**: Кортежи очень похожи со списками, единственное кортежи неизменяемы. Т.е мы не можем совершать операции изменяющие элементы кортежа. В остальном работа такая же как и со списками(нахождения индекса элемента, длины кортежа, обращение по индексу и т.д).

Создается кортеж с помощью круглых скобок ()

| a = () # для создания пустого кортежа  a = (2,) # для создания кортежа из одного элемента |
| --- |

Если же мы хотим изменить кортеж, мы можем преобразовать его в список, изменить, а потом преобразовать обратно в кортеж

| a = (1, 2)  a = list(a)  a.append(3)  a = tuple(a)  print(a) |
| --- |

**5****. Вывод содержимого списков**

**Учитель:** При использовании встроенной функции **print()** со списками выводится список в квадратных скобках, и все его элементы разделены запятой. Для просмотра содержимого списков такой вывод достаточно удобен, однако бывают ситуации и задачи где вывод должен соответствовать определенным требованиям. Например, я могу попросить вас вывести каждый элемент списка на отдельной строке, или разделить элементы символом пробела, без запятых, и вывести список без квадратных скобок по бокам.

Другими словами, нужно уметь адаптировать вывод содержимого списка под задачу.. Это несложно. Цикл for нам в помощь, как говорится.

| a = [1, 2, 3, ‘a’, True]  for i in a:  print(i) |
| --- |

| a = [1, 2, 3, ‘a’, True]  for i in range(len(a)):  print(a[i]) |
| --- |

| a = [1, 2, 3, ‘a’, True]  print(\*a) # распаковка списка(выводит без квадратных скобок) |
| --- |

1. Отличается ли вывод списка от вывода содержимого строки?

**Ответ:** нет, можно использовать два вида цикла, для вывода содержимого списка. Первый удобно использовать, когда нужны индексы элементов, второй, когда нужны сами элементы. Демонстрация на слайде.

1. А было ли что-нибудь новое, что мы до этих пор не использовали со строками?

**Ответ:** да, выводить содержимое списка можно еще и с помощью так называемой распаковки списка.

**Замечание.** У учеников может возникнуть вопрос, как работает распаковка списков. Ответить пока можно так: это магия Python и на текущем уровне владения языком нам не обязательно понимать подробности ее реализации.

**6****. Строковые методы split и join**

**Учитель:** При изучении **строковых методов** мы намеренно обошли стороной два очень важных метода split() и join(). Обошли мы их по очень простой причине, они работают со строками и списками в паре. Сейчас мы знаем и о строках, и о списках, поэтому можем изучить их.

При создании (конструировании) списков мы считывали построчно элементы списка, а затем добавляли их в список. Но что если начальные данные расположены в одной строке и разделены символом пробела? Такой способ представления данных достаточно удобен и часто встречается в реальной жизни. Как нам сконструировать список из такой строки? Другими словами, как нам разбить такую строку на элементы списка?

**Метод split**

| languages = 'Python C# Java'.split()  print(languages) # ['Python', 'C#', 'Java']  numbers = '1 2 3 4 5'.split()  print(numbers) # ['1', '2', '3', '4', '5']  words = 'To be or not to be that is the question'.split()  print(words) # ['To', 'be', 'or', 'not', 'to', 'be', 'that', 'is', 'the', 'question']  ip = '192.168.1.1'.split('.')  print(ip) # ['192', '168', '1', '1']  terms = '1 + 2 + 3 + 4 = 10'.split(' + ')  print(terms) # ['1', '2', '3', '4 = 10'] |
| --- |

Функция split сканирует всю строку и разделяет ее в случае нахождения разделителя.

В строке должен быть как минимум один разделитель. Им может выступать в том

числе и символ пробела. Пробел — разделитель по умолчанию.

**Метод join**

| languages = ' '.join(['Python', 'C#', 'Java'])  print(languages) # 'Python C# Java'  numbers = ' '.join(['1', '2', '3', '4', '5'])  print(numbers) # '1 2 3 4 5'  words = ' '.join(['To', 'be', 'or', 'not', 'to', 'be', 'that', 'is', 'the', 'question'])  print(words) # 'To be or not to be that is the question'  ip = '.'.join(['192', '168', '1', '1'])  print(ip) # '192.168.1.1'  terms = ' + '.join(['1', '2', '3', '4 = 10'])  print(terms) # '1 + 2 + 3 + 4 = 10' |
| --- |

Метод join в Python отвечает за объединение списка строк с помощью определенного указателя. Часто это используется при конвертации списка в строку. Например, так можно конвертировать список букв алфавита в разделенную запятыми строку для сохранения.

Метод принимает итерируемый объект в качестве аргумента, а поскольку список отвечает этим условиям, то его вполне можно использовать. Также список должен состоять из строк. Если попробовать использовать функцию для списка с другим содержимым, то результатом будет такое сообщение: TypeError: sequence item 0: expected str instance, int found.

**7****. Словари**

**Учитель:** Пора познакомиться одним из важнейших типов данных - словари. Словари очень часто применяются в программировании, особенно когда наша программа работает через интернет. Если списки и кортежи достаточно простой тип данных, т.к как хранят значения через запятую, то синтаксис словарей немного другой.

| a = {ключ: значение} |
| --- |

Словарь, как мы видим имеет более сложную структуру и состоит из ключа и значения. Обратиться как в списке или кортеже по индексу в данном случае мы не можем. Создается словарь с помощью фигурных скобок {}

| a = {}  print(a)  print(type(a)) |
| --- |

В данном примере создается пустой словарь, а команда type(), позволяет проверить нам тип данных, к которому относится переменная.

Для создания словаря с уже существующими данными достаточно указать ключ и значение

| a = {‘a’: 1, ‘b’: 2} |
| --- |

Также создать словарь мы можем из существующих списков и кортежей c помощью функции dict(), которая преобразует в словарь.

| a = dict([(1, 2), [4, 5]])  print(a) |
| --- |

Для добавления или изменения значения в словаре мы должны указать ключ, для которого хотим указать значение. Если данного ключа нет, то он будет создан

| a = {'a': 1}  a['a'] = 4  a['b'] = 2  print(a) |
| --- |

Результат

{'a': 4, 'b': 2}

Рассмотрим основные методы словарей:

* **clear**() - очищает словарь.
* **copy**() - возвращает копию словаря.
* **get**(key[, default]) - возвращает значение ключа, но если его нет, не бросает 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, возвращает default (по умолчанию бросает исключение).
* **popitem**() - удаляет и возвращает пару (ключ, значение). Если словарь пуст, бросает исключение KeyError.
* **setdefault**(key[, default]) - возвращает значение ключа, но если его нет, не бросает исключение, а создает ключ со 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.

1. **clear**

| a = {2: 1}  print(a)  a.clear()  print(a) |
| --- |

Результат

{2:1}

{}

1. **copy**

| a = {2: 1}  b = a.copy() # создаем копию  c = a  print(b)  print(b is a) # b не является a  print(c is a) # c является a |
| --- |

Результат

{2: 1}

False

True

1. **get**

| a = {2: 1}  print(a.get(2, 3))  print(a.get(1, 3)) |
| --- |

Результат

1

3

1. **items**

| a = {2: 1, 'a': 4}  print(a.items()) |
| --- |

Результат

dict\_items([(2, 1), ('a', 4)])

1. **keys**

| a = {2: 1, 'a': 4}  print(a.keys()) |
| --- |

Результат

dict\_keys([2, 'a'])

1. **pop**

| a = {2: 1, 'a': 4}  b = a.pop('a')  print(b)  print(a) |
| --- |

Результат

4

{2: 1}

1. **popitem**

| a = {2: 1, 'a': 4}  b = a.popitem()  print(b)  print(a) |
| --- |

Результат

('a', 4)

{2: 1}

1. **setdefault**

| a = {2: 1, 'a': 4}  a.setdefault(3)  print(a) |
| --- |

Результат

{2: 1, 'a': 4, 3: None}

1. **update**

| a = {2: 1}  a.update({'a': 2})  print(a) |
| --- |

Результат

{2: 1, 'a': 2}

1. **values**

| a = {2: 1, 'a': 2}  print(a.values()) |
| --- |

Результат

dict\_values([1, 2])

Словари также можно перебирать с помощью циклов. По умолчанию перебор идет по ключам, но мы можем это изменить с помощью методов values, items

| a = {2: 1, 'a': 2}  for key in a:  print(key)  for key, value in a.items():  print(key, value)  for item in a.items():  print(item) |
| --- |

Результат

2

a

2 1

a 2

(2, 1)

('a', 2)

**8****. Множества**

Множества — это неупорядоченная коллекция уникальных элементов, сгруппированных под одним именем. Множество может быть неоднородным — включать элементы разных типов. Множество всегда состоит только из уникальных элементов (дубли запрещены) в отличие от списков и кортежей в Python.

Создать объект множества(set) в Python можно двумя путями:

| a = {1, 2, 3}  a = set() |
| --- |

Нет ограничений на количество элементов в объекте set, но запрещено добавлять элементы изменяемых типов, такие как список или словарь. Если попробовать добавить список (с набором элементов), интерпретатор выдаст ошибку.

| a = {1, 2, 3, [1, 2, 3]}  print(a) |
| --- |

Добавить элемент в множество мы можем с помощью add

| set1 = {1, 3, 4}  set1.add(2)  print(set1) |
| --- |

Добавить несколько элементов с помощью update

| set2 = {1, 2, 3}  set2.update([4, 5, 6])  print(set2) # {1, 2, 3, 4, 5, 6} |
| --- |

Один или несколько элементов можно удалить из объекта set с помощью следующих методов. Их отличие в виде возвращаемого значения.

1. remove()
2. discard()
3. pop()

Метод remove() полезен в тех случаях, когда нужно удалить из множества конкретный элемент и вернуть ошибку в том случае, если его нет в объекте.

| set1 = {1, 2, 3, 4, 'a', 'p'}  set1.remove(2)  print(set1) |
| --- |

Метод discard() полезен, потому что он удаляет конкретный элемент и не возвращает ошибку, если тот не был найден во множестве.

| set1 = {1, 3, 4, 'a', 'p'}  set1.discard('a')  print(set1) |
| --- |

Метод pop() удаляет по одному элементу за раз в случайном порядке. Set — это неупорядоченная коллекция, поэтому pop() не требует аргументов (индексов в этом случае). Метод pop() можно воспринимать как неконтролируемый способ удаления элементов по одному из множеств в Python.

| set1 = {1, 3, 4, ‘p’}  set1.pop() |
| --- |

Ну и самым часто используемым свойством множеств, которыми пользуются, является, то что множество не может хранить дубли

| list1 = [1, 2, 1, 3]  list1 = set(list1)  list1 = list(list1)  print(list1) |
| --- |

**9****. Функции без параметров**

**Учитель:** В предыдущих уроках мы использовали встроенные в Python функции print(), input(), int(), str(), len() и многие другие. Пришло время начать писать свои собственные функции.

Предположим нам надо написать программу, которая рисовала бы звездный прямоугольник размерами 5×7 (5 строк и 7 столбцов).

Наш первый вариант кода выглядел бы примерно так:

| print('\*\*\*\*\*\*\*')  print('\*\*\*\*\*\*\*')  print('\*\*\*\*\*\*\*')  print('\*\*\*\*\*\*\*')  print('\*\*\*\*\*\*\*') |
| --- |

Далее мы изучили оператор умножения строки на число (оператор повторения) и написали бы код:

| print('\*' \* 7)  print('\*' \* 7)  print('\*' \* 7)  print('\*' \* 7)  print('\*' \* 7) |
| --- |

Ну и, наконец, мы изучили циклы, после чего код принял бы вид:

| for \_ in range(5):  print('\*' \* 7) |
| --- |

А теперь представим, что таких прямоугольников нужно изобразить несколько, скажем, три штуки.

Тогда код программы будет иметь вид:

| for \_ in range(5):  print('\*' \* 7)  print()  for \_ in range(5):  print('\*' \* 7)  print()  for \_ in range(5):  print('\*' \* 7) |
| --- |

Результатом выполнения такого кода будет:

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

Этот код полностью решает поставленную задачу, но не лишен недостатков.

* Во-первых, он довольно громоздок, поскольку часть кода отвечающая за вывод прямоугольника дублируется.
* Во-вторых, если понадобится поменять размеры прямоугольника, придется менять их столько раз, сколько раз он выводится.

Во избежание таких недостатков используют функции. Вместо повторения кода, осуществляющего вывод прямоугольника, перенесем его в отдельную функцию, а затем вызовем ее 3 раза.

Для создания функции пишем такой код:

| def draw\_box():  for \_ in range(5):  print('\*' \* 7) |
| --- |

Когда функция создана, чтобы увидеть результат ее работы, нужно ее вызвать по имени, следующим образом:

| draw\_box() |
| --- |

Теперь для того, чтобы изобразить 3 прямоугольника мы напишем код:

| draw\_box()  print()  draw\_box()  print()  draw\_box() |
| --- |

Можно вызывать функцию в цикле столько раз, сколько нужно.

Код стал короче, читабельнее (за счет удачного названия функции), а главное, если нам потребуется изменить размеры прямоугольника, то достаточно будет внести изменения только в самой функции draw\_box().

Имена функциям назначают точно так же, как переменным. Имя функции должно быть достаточно описательным, чтобы любой, читающий ваш код, мог легко догадаться, что именно функция делает. Python требует соблюдения тех же правил, что при именовании переменных. Поскольку функции выполняют действия, большинство программистов предпочитает в их именах использовать глаголы.

Итак, функция – отдельная, независимо действующая часть программы, выполняющая определенную задачу. Функции объявляются с помощью ключевого слова def (от define – определять). За ключевым словом def следует название функции, круглые скобки () и двоеточие.

| def название\_функции():  блок кода |
| --- |

Первая строка объявления функции называется ее заголовком. Начиная со следующей строки идет блок кода - тело функции, набор инструкций, составляющих единое целое. Эти инструкции выполняются каждый раз, когда вызывается функция.

Обратите внимание, каждая строка в теле функции выделена отступом. Последняя выделенная отступом строка - последняя строка тела функции. Для выделения строк блока кода отступом программисты Python обычно используют четыре пробела, в соответствии со стандартом PEP 8.

Рассмотрим объявление функции:

| def print\_message() :  print('Я - Артур,')  print('король британцев. ') |
| --- |

Этот фрагмент кода определяет функцию с именем print\_message(). Тело функции состоит из двух инструкций. Вызов функции приведет к исполнению этих инструкций.

После того, как функция объявлена, чтобы увидеть ее результат, нужно ее вызвать - написать ее название, а затем круглые скобки.

| # объявление функции  def print\_message() :  print('Я - Артур,')  print('король британцев. ')  # вызов функции  print\_message() |
| --- |

**10****. Функции с параметрами**

**Учитель:** До этого мы определили функцию draw\_box(), которая выводит звездный прямоугольник с размерами 5×7:

| def draw\_box():  for i in range(5):  print('\*' \* 7) |
| --- |

Было бы намного удобнее, если бы функция draw\_box() выводила прямоугольник с произвольными размерами. И действительно, функции могут принимать параметры, что делает их более гибкими.

Функции с параметрами объявляют также, как и функции без параметров, только в скобках указывают параметры:

| def название\_функции(параметры):  блок кода |
| --- |

Давайте перепишем предыдущую версию функции draw\_box() так, чтобы она принимала параметры, задающие высоту и ширину прямоугольника:

| def draw\_box(height, width): # функция принимает два параметра  for i in range(height):  print('\*' \* width) |
| --- |

Теперь наша функция draw\_box() принимает два целочисленных параметра height – высота прямоугольника и width – ширина прямоугольника, и для ее вызова нам нужно обязательно их указать.

Чтобы вывести звездный прямоугольник размерами 5 на 7 мы пишем код:

| draw\_box(5, 7) |
| --- |

Результатом такого вызова функции draw\_box(5, 7) будет:

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

Чтобы вывести прямоугольник размерами 10 на 15, мы пишем код:

| draw\_box(10, 15) |
| --- |

Результатом такого вызова функции draw\_box(10, 15) будет:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Теперь с помощью нашей новой версии функции draw\_box() можем в одной программе выводить прямоугольники разных размеров. Следующий программный код:

| draw\_box(3, 3)  print()  draw\_box(5, 5)  print()  draw\_box(4, 10) |
| --- |

выведет:

\*\*\*

\*\*\*

\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

На место параметров мы можем подставлять не только целочисленные константы, но и значения переменных. Следующий программный код:

| n = 3  m = 9  draw\_box(n, m) |
| --- |

выведет:

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

**10****. Решение задач**

**Дополнительно**

Если на уроке остается время, то ученикам можно предложить начать прорешивать домашнее задание.

**Домашняя работа**

Задача 1

Дан список:

a = [1, 2, 3, 4, 5, 6]

Требуется удалить из списка все нечетные элементы, а четные разделить нацело на два. Новый список создавать нельзя

Решение

| a = [1, 2, 3, 4, 5, 6]  i = 0  n = len(a)  while i < n:  if a[i] % 2 == 0:  a[i] //= 2  i += 1  else:  a.pop(i)  n -= 1  print(a) |
| --- |