**Модуль 2. Базовый**

**Урок 3. Индексы и срезы.**

**Индексы и срезы**

Рассмотрим индексированные коллекции (их еще называют последовательности — sequences) — список (list), кортеж (tuple), строку (string).

Под индексированностью имеется ввиду, что элементы коллекции располагаются в определённом порядке, каждый элемент имеет свой индекс от 0 (то есть первый по счёту элемент имеет индекс не 1, а 0) до индекса на единицу меньшего длины коллекции (т.е. len(mycollection)-1).

**Получение значения по индексу**

Для всех индексированных коллекций можно получить значение элемента по его индексу в квадратных скобках. Причем, можно задавать отрицательный индекс, это значит, что будем находить элемент с конца считая обратном порядке.

При задании отрицательного индекса, последний элемент имеет индекс -1, предпоследний -2 и так далее до первого элемента индекс которого равен значению длины коллекции с отрицательным знаком, то есть (-len(mycollection).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **элементы** | **a** | **b** | **c** | **d** | **e** |
| **индексы** | **0 (-5)** | **1 (-4)** | **2 (-3)** | **3 (-2)** | **4 (-1)** |

my\_str = "abcde"

 print(my\_str[0]) # a - первый элемент

 print(my\_str[-1]) # e - последний элемент

 print(my\_str[len(my\_str)-1])

# e - так тоже можно взять последний элемент

 print(my\_str[-2]) # d - предпоследний элемент

Наши коллекции могут иметь несколько уровней вложенности, как список списков в примере ниже. Для перехода на уровень глубже ставится вторая пара квадратных скобок и так далее.

my\_2lvl\_list = [[1, 2, 3], ['a', 'b', 'c']]

print(my\_2lvl\_list[0])

# [1, 2, 3] - первый элемент — первый вложенный список

print(my\_2lvl\_list[0][0])

# 1 — первый элемент первого вложенного списка

print(my\_2lvl\_list[1][-1])

# с — последний элемент второго вложенного списка

**Изменение элемента списка по индексу**

Поскольку кортежи и строки у нас неизменяемые коллекции, то по индексу мы можем только брать элементы, но не менять их:

my\_tuple = (1, 2, 3, 4, 5)

print(my\_tuple[0]) # 1

my\_tuple[0] = 100

# TypeError: 'tuple' object does not support item assignment

А вот для списка, если взятие элемента по индексу располагается в левой части выражения, а далее идёт оператор присваивания =, то мы задаём новое значение элементу с этим индексом.

**Изменение элемента списка по индексу**

Поскольку кортежи и строки у нас неизменяемые коллекции, то по индексу мы можем только брать элементы, но не менять их:

my\_tuple = (1, 2, 3, 4, 5)

print(my\_tuple[0]) # 1

my\_tuple[0] = 100

# TypeError: 'tuple' object does not support item assignment

А вот для списка, если взятие элемента по индексу располагается в левой части выражения, а далее идёт оператор присваивания =, то мы задаём новое значение элементу с этим индексом.

my\_list = [1, 2, 3, [4, 5]]

my\_list[0] = 10

my\_list[-1][0] = 40

print(my\_list) # [10, 2, 3, [40, 5]]

Для такого присвоения, элемент уже должен существовать в списке, нельзя таким образом добавить элемент на несуществующий индекс.

my\_list = [1, 2, 3, 4, 5]

my\_list[5] = 6 # IndexError: list assignment index out of range

**Срезы**

Очень часто, надо получить не один какой-то элемент, а некоторый их набор ограниченный определенными простыми правилами — например первые 5 или последние три, или каждый второй элемент — в таких задачах, вместо перебора в цикле намного удобнее использовать так называемый срез (slice, slicing).

Следует помнить, что, взяв элемент по индексу или срезом (slice), мы не как не меняем исходную коллекцию, мы просто скопировали ее часть для дальнейшего использования (например добавления в другую коллекцию, вывода на печать, каких-то вычислений). Поскольку сама коллекция не меняется — это применимо как к изменяемым (список) так и к неизменяемым (строка, кортеж) последовательностям.

Синтаксис среза похож на таковой для индексации, но в квадратных скобках вместо одного значения указывается 2-3 через двоеточие:

my\_collection[start:stop:step] # старт, стоп и шаг

**Особенности среза:**

Отрицательные значения старта и стопа означают, что считать надо не с начала, а с конца коллекции.

Отрицательное значение шага — перебор ведём в обратном порядке справа налево.

Если не указан старт [:stop:step]— начинаем с самого края коллекции, то есть с первого элемента (включая его), если шаг положительный или с последнего (включая его), если шаг отрицательный (и соответственно перебор идет от конца к началу).

Если не указан стоп [start:: step] — идем до самого края коллекции, то есть до последнего элемента (включая его), если шаг положительный или до первого элемента (включая его), если шаг отрицательный (и соответственно перебор идет от конца к началу).

step = 1, то есть последовательный перебор слева направо указывать не обязательно — это значение шага по умолчанию. В таком случае достаточно указать [start:stop]

Можно сделать даже так [:] — это значит взять коллекцию целиком

**При срезе, первый индекс входит в выборку, а второй нет!**

То есть от старта включительно, до стопа, где стоп не включается в результат. Математически это можно было бы записать как [start, stop] или пояснить вот таким правилом:

[<первый включаемый> : <первый НЕ включаемый> : <шаг> ]

Поэтому, например, mylist[::-1] не идентично mylist[:0:-1], так как в первом случае мы включим все элементы, а во втором дойдем до 0 индекса, но не включим его!

**Примеры срезов:**

col = 'abcdefg'

print(col[:]) # abcdefg

print(col[::-1]) # gfedcba

print(col[::2]) # aceg

print(col[1::2]) # bdf

print(col[:1]) # a

print(col[-1:]) # g

print(col[3:4]) # d

print(col[-3:]) # efg

print(col[-3:1:-1]) # edc

print(col[2:5]) # cde

**Срез аналоги .append() и insert()**

Можно менять части последовательности — это применение выглядит наиболее интересным, так как решает задачу просто и наглядно.

my\_list = [1, 2, 3, 4, 5]

my\_list[1:3] = [20, 30]

print(my\_list) # [1, 20, 30, 4, 5]

my\_list[1:3] = [0] # нет проблем заменить два элемента на один

print(my\_list) # [1, 0, 4, 5]

my\_list[2:] = [40, 50, 60] # или два элемента на три

print(my\_list) # [1, 0, 40, 50, 60]

Можно просто удалить часть последовательности

my\_list = [1, 2, 3, 4, 5]

my\_list[:2] = [] # или del my\_list[:2]

print(my\_list) # [3, 4, 5]

**Выход за границы индекса**

Обращение по индексу по сути является частным случаем среза, когда мы обращаемся только к одному элементу, а не диапазону. Но есть очень важное отличие в обработке ситуации с отсутствующим элементом с искомым индексом.

Обращение к несуществующему индексу коллекции вызывает ошибку:

my\_list = [1, 2, 3, 4, 5]

print(my\_list[-10]) # IndexError: list index out of range

print(my\_list[10]) # IndexError: list index out of range

А в случае выхода границ среза за границы коллекции никакой ошибки не происходит:

my\_list = [1, 2, 3, 4, 5]

print(my\_list[0:10]) # [1, 2, 3, 4, 5] — отработали в пределах коллекции

print(my\_list[10:100]) # [] - таких элементов нет — вернули пустую коллекцию

print(my\_list[10:11]) # [] - проверяем 1 отсутствующий элемент - пустая коллекция, без ошибки

**Сортировка элементов коллекции**

**Функция sorted()**

Мы может использовать функцию sorted() для вывода списка сортированных элементов любой коллекции для последующее обработки или вывода.

* функция не меняет исходную коллекцию, а возвращает новый список из ее элементов;
* не зависимо от типа исходной коллекции, вернётся список (list) ее элементов;
* поскольку она не меняет исходную коллекцию, ее можно применять к неизменяемым коллекциям;
* Поскольку при сортировке возвращаемых элементов нам не важно, был ли у элемента некий индекс в исходной коллекции, можно применять к неиндексированным коллекциям;

Имеет дополнительные не обязательные аргументы:

reverse=True — сортировка в обратном порядке

key=funcname (начиная с Python 2.4) — сортировка с помощью специальной функции funcname, она может быть как стандартной функцией Python, так и специально написанной вами для данной задачи функцией и лямбдой.

my\_list = [2, 5, 1, 7, 3]

my\_list\_sorted = sorted(my\_list)

print(my\_list\_sorted) # [1, 2, 3, 5, 7]

my\_set = {2, 5, 1, 7, 3}

my\_set\_sorted = sorted(my\_set, reverse=True)

print(my\_set\_sorted) # [7, 5, 3, 2, 1]

Пример сортировки списка строк по длине len() каждого элемента:

my\_files = ['somecat.jpg', 'pc.png', 'apple.bmp', 'mydog.gif']

my\_files\_sorted = sorted(my\_files, key=len)

print(my\_files\_sorted) # ['pc.png', 'apple.bmp', 'mydog.gif', 'somecat.jpg']

**Функция reversed()**

Функция reversed() применяется для последовательностей и работает по другому:

* возвращает генератор списка, а не сам список;
* если нужно получить не генератор, а готовый список, результат можно обернуть в list() или же вместо reversed() воспользоваться срезом [: :-1];
* она не сортирует элементы, а возвращает их в обратном порядке, то есть читает с конца списка;
* из предыдущего пункта понятно, что если у нас коллекция неиндексированная — мы не можем вывести её элементы в обратном порядке и эта функция к таким коллекциям не применима — получим «TypeError: argument to reversed() must be a sequence»;
* не позволяет использовать дополнительные аргументы — будет ошибка «TypeError: reversed() does not take keyword arguments».

my\_list = [2, 5, 1, 7, 3]

my\_list\_sorted = reversed(my\_list)

print(my\_list\_sorted) # <listreverseiterator object at 0x7f8982121450>

print(list(my\_list\_sorted)) # [3, 7, 1, 5, 2]

print(my\_list[::-1]) # [3, 7, 1, 5, 2] - тот же результат с помощью среза

**Методы списка .sort() и .reverse()**

У списка (и только у него) есть особые методы .sort() и .reverse() которые делают тоже самое, что соответствующие функции sorted() и reversed(), но при этом:

* Меняют сам исходный список, а не генерируют новый;
* Возвращают None, а не новый список;
* поддерживают те же дополнительные аргументы;
* в них не надо передавать сам список первым параметром, более того, если это сделать — будет ошибка — не верное количество аргументов.

my\_list = [2, 5, 1, 7, 3]

my\_list.sort()

print(my\_list) # [1, 2, 3, 5, 7]

**Устойчивость сортировки**

Допустим данные нужно отсортировать сначала по столбцу А по возрастанию, затем по столбцу B по убыванию, и наконец по столбцу C снова по возрастанию.

Если данные в столбце B числовые, то при помощи подходящей функции в key можно поменять знак у элементов B, что приведёт к необходимому результату.

А если все данные текстовые? Тут есть такая возможность.

Дело в том, что сортировка sort в Python устойчивая (начиная с Python 2.2), то есть она не меняет порядок «одинаковых» элементов.

**Задание:**

Дан список друзей из 10 фамилий

* Пользователю предлагают ввести любой номер,
* Выводим на печать фамилию с данным номером.
* Задать вопрос - оставить в друзьях?
* Если ответ - да, список оставляем
* Если ответ нет - удаляем фамилию из списка.
* Спрашиваем - повторить?
* Если да- переходим к п. 1
* Если нет - печатаем итоговый список фамилий друзей.