GeekBrains

Лекция 3. Коллекции

Погружение в Python

Оглавление

На этой лекции мы

1. Разберёмся, что такое коллекция и какие коллекции есть в Python.
2. Изучим работу со списками, как с самой популярной коллекцией.
3. Узнаем, как работать со строкой в ключе коллекция.
4. Разберём работу с кортежами.
5. Узнаем, что такое словари и как с ними работать.
6. Изучим множества и особенности работы с ними.
7. Познакомимся с классами байт и массив байт.

Дополнительные материалы к лекции

Форматирование строк
https://docs.python.org/3/library/string.html#format-specification-mini-language

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Познакомились со строгой динамической типизацией языка Python.
2. Изучили понятие объекта в Python.
   Разобрались с атрибутами и методами объектов.
3. Рассмотрели способы аннотации типов.
4. Изучили "простые" типы данных, такие как числа и строки.
5. Узнали про математические возможности Python.

Термины лекции

- **Коллекция** — это программный объект который объединяет в себе несколько более простых объектов
- **LIFO** (англ. last in, first out, «последним пришёл — первым ушёл») — способ организации и манипулирования данными относительно времени и приоритетов.
- **Массив** — упорядоченный набор элементов, каждый из которых хранит одно значение, идентифицируемое с помощью одного или нескольких индексов.
- **Список в информатике** — это абстрактный тип данных, представляющий собой упорядоченный набор значений, в котором некоторое значение может встречаться более одного раза.

Подробный текст лекции

Введение

Сегодня на лекции мы поговорим о коллекциях. Напомню, что коллекция — это программный объект, который объединяет в себе несколько более простых объектов. В Python одна коллекция может являться частью другой коллекции. При этом программист может работать с коллекцией как с единым объектом, обращаться к элементам коллекции и пользоваться встроенными в коллекцию методами и атрибутами.

1. Списки, list

List, список является самой часто используемой коллекцией в Python. Прежде чем говорить о списках, я напомню, что такое массив в информатике. Массив - это непрерывная область в оперативной памяти компьютера, поделённая на ячейки одинакового размера хранящие данные одного типа. Массивы могут быть статическими, то есть размер массива нельзя изменить, и динамическими, когда размер массива изменяется при добавлении или удалении элементов. Один из самых больших плюсов в работе с массивами — это доступ к любой из его ячеек за константное время.

Массив — упорядоченный набор элементов, каждый из которых хранит одно значение, идентифицируемое с помощью одного или нескольких индексов. В простейшем случае массив имеет постоянную длину и хранит единицы данных одного и того же типа, а в качестве индексов выступают целые числа.
В информатике, **список (англ. list)** — это абстрактный тип данных, представляющий собой упорядоченный набор значений, в котором некоторое значение может встречаться более одного раза. Экземпляр списка является компьютерной реализацией математического понятия конечной последовательности. Экземпляры значений, находящихся в списке, называются элементами списка (англ. item, entry либо element); если значение встречается несколько раз, каждое вхождение считается отдельным элементом.

Почему list "список", а не "массив"

Если вы уже пользовались питоновскими списками, то могли задуматься о путанице в понятиях. Ведь список по описанию очень похож на динамический массив.
Пройдемся по основным характеристика и сравнить его с массивом и со списком:

- Единый блок в памяти — массив
- Хранит значения любых типов вперемешку — не массив???
- Добавление элементов в конец — динамический массив
- Доступ к элементам по индексу с константной сложностью — массив

Стоп! Доступ к элементам по индексу за O(1) возможен, если каждая ячейка имеет одинаковый размер. Но как в таком случае хранить разные элементы — делать ячейку по максимальному из хранимых элементов? В таком случае будет много

пустоты. А если добавится новый элемент и он больше всех уже хранящихся? Проводить переупаковку?
Правда в том, что массив хранит значения одного типа — указатели. И все указатели занимают фиксированный размер в 1, 2 (в далёких 90-х), 4 или 8 байт. Зависит от ОС и её разрядности, а также от версии Python и его разрядности.
Указатель — это переменная, в которой хранится адрес объекта в памяти.
И в этом месте мы понимаем, где список работает как список. Обращаясь к элементу массива по индексу мы получает указатель на реальных объект, хранящийся отдельно. Благодаря этому приему список может смешивать объекты разных типов и размеров внутри себя.

Работа со списками

Прежде чем разберём основные методы списков, поговорим о способах его создания.
Основной — вызов функции list(). Без аргументов функция возвращает пустой список. Если передать последовательность, функция вернёт список содержащий переданные элементы.
Однако есть более простой и привычный для большинства программистов способ создания списков. Используются квадратные скобки, внутри которых перечисляются элементы списка. Квадратные скобки — синтаксический сахар, который не только короче пишется, но и работает быстрее. https://habr.com/ru/post/671602/

```
list_1 = list()
list_2 = list((3.14, True, "Hello world!"))
list_3 = []
list_4 = [3.14, True, "Hello world!"]
```

Кроме того список можно создать из другой последовательности. Достаточно передать её функции list.

```
new_list = list(other_iterator)
```

🔥 **Важно!** Преобразование одной коллекции в другую имеет линейную асимптотику, т.е. O(n). И по времени, и по памяти конечно же. Ведь мы создаём объект списка и копируем в него данные.

Доступ к элементам списка

Как мы уже разобрались список ведёт себя как динамический массив. А это значит, что к любому элементу списка можно обратиться по индексу. В Python, как и в большинстве языков программирования, индексация начинается с нуля, то есть, первый элемент лежит в ячейке под индексом 0, следующий элемент в ячейке один и так далее.

```
my_list = [2, 4, 6, 8, 10, 12]
print(my_list[0])
print(my_list[6])  # IndexError: list index out of range
```

При попытке обратиться к несуществующему элементу получим ошибку IndexError. Положительный индекс считается слева направо. Если индекс отрицательный вычисление элемента идёт справа налево то есть от конца к началу.

🔥 **Важно!** Вы же знаете что в математике нет разделения на ноль и минус ноль. Самый правый элемент списка, т.е. последний элемент, имеет индекс - 1, Предпоследний - -2 и т.д.

```
my_list = [2, 4, 6, 8, 10, 12]
print(my_list[-1])
print(my_list[-10])  # IndexError: list index out of range
```

Как и в случае с положительными индексами, попытка обратиться к элементу за пределами списка через отрицательный индекс вызывает ошибку.

Метод append

Для добавления нового элемента в конец списка используется метод append. Метод принимает один аргумент — объект который будет добавлен в конец динамически увеличенного списка.

```
a = 42
b = 'Hello world!'
c = [1, 3, 5, 7]
my_list = [None]
my_list.append(a)
print(my_list)
my_list.append(b)
```

```
print(my_list)
my_list.append(c)
print(my_list)
```

Обратите внимание, что при добавлении списка c в список my_list он оказался целиком в одной ячейке списка. Ссылочная система Pyhton позволяет формировать структуры любой глубины вложенности.

Ниже пример плохого кода. Мы добавили в список сам себя.

```
my_list.append(my_list)
print(my_list)
# Вывод: [None, 42, 'Hello world!', [1, 3, 5, 7], [...]]
```

Многоточие в выводе говорит о рекурсивной ссылке. Явный намёк на то, что программист сделал что-то неверно своей программе.
Метод append имеет константную асимптотику O(1).

Метод extend

Метод extend ведёт себя аналогично append, то есть добавляет элементы в конец списка. В качестве аргумента extend принимает последовательность, итерируется по ней слева направо и каждый элемент добавляет в новую ячейку списка.

```
a = 42
b = 'Hello world!'
c = [1, 3, 5, 7]
my_list = [None]
my_list.extend(a)  # TypeError: 'int' object is not iterable
print(my_list)
my_list.extend(b)
print(my_list)
my_list.extend(c)
print(my_list)
my_list.extend(my_list)
print(my_list)
```

- **extend(a)** — если в метод передать не коллекцию, получим ошибку TypeError.
- **extend(b)** — строка воспринимается как коллекция, в результате каждый символ строки помещается в новую ячейку списка.

- **extend(c)** — итерируемся по списку, с последовательно добавляя его элементы в список my_list
- **extend(my_list)** — удваиваем список, добавляя копию всех его элементов.

Метод extend имеет константную асимптотику O(1) на добавление одного элемента и зависит от количества переданных объектов, т.е. O(n), где n - количество элементов в последовательности.

Метод pop

Метод pop позволяет удалить элемент списка. Удаляемый элемент возвращается как результат работы метода.

```
my_list = [2, 4, 6, 8, 10, 12]
spam = my_list.pop()
print(spam, my_list)
eggs = my_list.pop(1)
print(eggs, my_list)
err = my_list.pop(10)  # IndexError: pop index out of range
```

Метод pop может принимать на вход индекс удаляемого элемента. Этот элемент вернётся как результат работы метода, т.е. может быть сохранён в переменную. Индекс может быть отрицательным. Тогда элемент считается от правой части списка, с конца.

Если указать индекс выходящий за границы списка получим ошибку IndexError.
Метод pop без аргумента имеет константную асимптотику O(1). Метод pop с аргументом имеет линейную асимптотику. Список не допускает пустых ячеек, поэтому при удаление элемента из середины списка все элементы находящиеся правее сдвигаются на одну ячейку влево. Получаем O(n), где n - количество элементов правее удаляемого.

Python Style! Переменные spam и eggs используется в языке Python как временные переменные. Эти названия, ветчина и яйца, используют True Python разработчики. Ведь язык назван в честь комедийного шоу Летающий цирк Монти Пайтон. А одно из их видео посвящено шуткам про ветчину и яйца.

Метод count

Метод count подсчитывает вхождение элемента в список.

```
my_list = [2, 4, 6, 2, 8, 10, 12, 2, 4, 14, 2]
spam = my_list.count(2)
print(spam)
eggs = my_list.count(3)
print(eggs)
```

Метод принимает именно объект, а не индекс. Если объект отсутствует в списке, count возвращает ноль — элемент был встречен в списке ноль раз.
Count имеет линейную асимптотику O(n), т.к. для подсчёта метод перебирает все элемента списка и сравнивает их с переданным объектом.

Метод index

Метод index возвращает индекс переданного объекта внутри списка.

```
my_list = [2, 4, 6, 2, 8, 10, 12, 2, 4, 14, 2]
spam = my_list.index(4)
print(spam)
eggs = my_list.index(4, spam + 1, 90)
print(eggs)
err = my_list.index(3)  # ValueError: 3 is not in list
```

Метод работает до первого вхождения, т.е. если элемент встречается несколько раз, возвращается индекс первой встречи. Метод может принимать два дополнительных аргумента: индекс start и индекс stop. В таком случае поиск элемента осуществляется внутри заданного диапазона. Если элемент отсутствует в списке, метод вызывает ошибку ValueError.
Чем ближе элемент к началу списка, тем быстрее работает метод. В лучшем случае получаем константную асимптотику O(1), в худшем линейную O(n).

🔥 **Внимание!** Если правая граница, stop индекс, окажется больше, чем количество элементов в списке, поиск будет осуществляться до конца списка. Ошибку это не вызовет.

Метод insert

Метод insert принимает на вход два аргумента — индекс для вставки и объект вставки. Метод добавляет элемент после индекса.

```
my_list = [2, 4, 6, 8, 10, 12]
my_list.insert(2, 555)
print(my_list)
my_list.insert(-2, 13)
print(my_list)
my_list.insert(42, 73)   # my_list.append(73)
print(my_list)
```

Если индекс положительный, элемент добавляется в указанную ячейку, а все последующие элементы списка сдвигаются на ячейку правее.
Если индекс отрицательный, отсчитывается необходимое количество элементов справа для вставки. Например -2 означает, что после вставки справа от добавленного элемента будет находится ещё два.
В том случае, когда индекс оказывается больше, чем количество элементов списка, объект добавляется в конец. В таком случае логичнее использовать метод append, выполняющий добавление элемента в конец списка.

Метод remove

Метод remove принимает на вход объект, производит его поиск в списке и удаляет в случае нахождения.

```
my_list = [2, 4, 6, 8, 10, 12, 6]
my_list.remove(6)
print(my_list)
my_list.remove(3)   # ValueError: list.remove(x): x not in list
print(my_list)
```

Если удаляемый элемент встречается в списке несколько раз, удаляется только один элемент — самый левый.
А если удаляемый элемент отсутствует в списке, будет вызвана ошибка ValueError.

Сортировка списков

Одна из частых операций при работе со списками их сортировка. Python позволяет отсортировать список на месте, inplace, т.е. не создавая копию. А можно создать копию отсортированного списка как отдельный объект.

🔥 **Важно!** При сортировке элементы списка должны быть одного типа. Иначе Python может не понять как сравнивать между собой элементы разных типов и вызовет ошибку.

```
my_list = ['H', 'e', 'l', 'l', 'o', 1, 3, 5, 7]
my_list.sort()  # TypeError: '<' not supported between instances of 'int' and 'str'
res = sorted(my_list)  # TypeError: '<' not supported between instances of 'int' and 'str'
```

Функция sorted()

Функция sorted принимает на вход любую коллекцию по которой можно итерироваться и возвращает отсортированный список.

🔥 **Важно!** Функция sorted может принимать не только списки, но и другие последовательности: строки, множества, кортежи, словари и т.п.. При этом функция всегда возвращает список.

```
my_list = [4, 8, 2, 9, 1, 7, 2]
sort_list = sorted(my_list)
print(my_list, sort_list, sep='\n')
rev_list = sorted(my_list, reverse=True)
print(my_list, rev_list, sep='\n')
```

Переданная в функцию коллекция остаётся неизменной после результата работы функции. Если в функцию передать дополнительный аргумент reverse=True, сортировка происходит по убыванию.

Внутри функции используется алгоритм сортировки Timsort — гибридная устойчивая сортировка с временной асимптотикой O(n log n). Дополнительно тратится O(n) памяти на создание нового отсортированного списка.

Метод sort()

Метод sort осуществляет сортировку элементов списка без создания копии, inplace.

```
my_list = [4, 8, 2, 9, 1, 7, 2]
my_list.sort()
print(my_list)
my_list.sort(reverse=True)
print(my_list)
```

Как и функция sorted метод sort упорядочивает элементы по возрастанию. Если передать дополнительный параметр reverse=True, будет произведена сортировка по убыванию. Внутри метода работает тот же самый алгоритм сортировки Timsort. Но память на создание копии списка мы не тратим.

Разворот списков

Python поддерживает операции по развороту списка. Первый элемент становится последним, второй — предпоследним и так далее.

Функция reversed()

Функция принимает на вход последовательность, которая поддерживает порядок элементов, возвращает функция объект итератор с обратным порядком элементов.

```
my_list = [4, 8, 2, 9, 1, 7, 2]
res = reversed(my_list)
print(type(res), res)
rev_list = list(reversed(my_list))
print(rev_list)
```

Получается, что результат работы функции напрямую не использовать? Если нам нужен новый развёрнутый список, объект итератор стоит обернуть в функцию list. В таком случае мы получим новый развёрнутый список.

🔥 **Важно!** Подобный приём затратен по времени и по памяти.

Обычно функция reversed используется в сочетании с циклом for in. Такой приём позволяет работать с элементами списка внутри цикла в обратном порядке.

```
for item in reversed(my_list):
    print(item)
```

Метод reverse() и синтаксический сахар [::-1]

Если нам нужно развёрнутая версия списка логичные и удобнее использовать встроенный метод reverse.

```
my_list = [4, 8, 2, 9, 1, 7, 2]
my_list.reverse()
print(my_list)
```

Метод разворачивает список на месте не создавая копии.
Кроме этого в Python есть возможность получить развернутую копию через особую запись в квадратных скобках, синтаксический сахар. После имени списка в квадратных скобках слитно записываем два двоеточия и минус один.

```
my_list = [4, 8, 2, 9, 1, 7, 2]
new_list = my_list[::-1]
print(my_list, new_list, sep='\n')
```

Подобная запись аналогична `rev_list = list(reversed(my_list))` рассмотренной выше. Разворот списка с использованием квадратных скобок — частный случай срезов.

Создание копий

Мы уже разобрали способы создания отсортированной копии списка, развернутой копии списка. На этом возможности копирования не заканчиваются.

Срезы

Используя квадратные скобки можно делать частичные копии списка - срезы.
Базовый синтаксис следующий.
list[start:stop:step]
start указывает на первый индекс, который включается в срез. При отсутствии значения start равен нулю, началу списка.
stop указывает на последний индекс, который не включается в срез. При отсутствии значения stop равен последнему элементу списка и включает его в срез.
step — шаг движения от star до stop. По умолчанию step равен единице, все элементы по порядку.

```
my_list = [2, 4, 6, 2, 8, 10, 12, 14, 16, 18]
print(my_list[2:7:2])
print(my_list[:7:2])
print(my_list[2::2])
print(my_list[2:7:])
print(my_list[8:3:-1])
print(my_list[3::])
print(my_list[:7:])
```

Метод copy()

Метод copy создаёт поверхностную копию списка. Начнём с плохого примера, чтобы понять пользу копий.

```
my_list = [2, 4, 6, 2, 8, 10, 12, 14, 16, 18]
new_list = my_list
print(my_list, new_list, sep='\n')
my_list[2] = 555
print(my_list, new_list, sep='\n')
```

Мы скопировали в переменную `new_list` указатель на список `my_list`. Далее мы изменили элемент в исходном списке. Новый список также оказался изменённым. Как вы помните list — изменяемый тип данных и подобное поведение нормально. Что делать, если нужно менять оригинал, но не затрагивать копию. Верно. Метод copy.

```
my_list = [2, 4, 6, 2, 8, 10, 12, 14, 16, 18]
new_list = my_list.copy()
print(my_list, new_list, sep='\n')
my_list[2] = 555
print(my_list, new_list, sep='\n')
```

Теперь изменяется лишь один список.

Зачем нужна функция copy.deepcopy()

Иногда программисту приходится работать с вложенными друг в друга коллекциями. Например матрица или список списков.

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
new_m = matrix.copy()
print(matrix, new_m, sep='\n')
matrix[0][1] = 555
print(matrix, new_m, sep='\n')
```

Метод copy создал поверхностную копию, копию верхнего уровня. Изменения же вложенных объектов отразится и на оригинале. В таком случае для создания полной копии любой глубины вложенности используют функцию deepcopy из модуля copy.

```
import copy

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
new_m = copy.deepcopy(matrix)
print(matrix, new_m, sep='\n')
matrix[0][1] = 555
print(matrix, new_m, sep='\n')
```

Функция рекурсивно обходит все вложенные объекты создавая их копии. Изменения одной коллекции теперь не затрагивают её копию.

Плюсы и минусы создания копии

При работе со списками важно помнить, что сам список как хранитель указателей на объекты занимает место в памяти. Дополнительно занимают память и сами объекты, на которые список указывает. Создание копии приводит к новым затратам памяти, ведь мы создаём новый объект список. Если вы работает с большими данными, создание копии может быть не лучшей идеей - может не хватить памяти ПК. Кроме того каждая копия требует временных ресурсов на копирование. Прежде чем использовать срезы, копии задумайтесь можно ли решить задачу иначе, экономя время и память.
С другой стороны небольшие списки быстро копируются. И если в вашей задаче важно сохранить оригинал, но нужно модифицировать список для получения результата — копирование вполне допустимо.

Функция len

В финале списков рассмотрим функцию len. На вход она принимает любую коллекцию, в которой можно посчитать количество элементов.

🔥 **Важно!** Функция одинаково работает не только для списков, но и для других коллекций.

```
my_list = [2, 4, 6, 2, 8, 10, 12, 14, 16, 18]
matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
print(len(my_list))
print(len(matrix))
print(len(matrix[1]))
```

Обратите внимание на результат работы функции. Подсчитывается количество элементов первого уровня вложенности. У списка 10 элементов в ячейках с нулевой по девятую. Функция вернула 10.

У матрицы три элемента списка в ячейках с нулевой по вторую. В каждом из вложенных списков по 4 элемента, т.е. суммарно 12. Но функция len вернула число 3 посчитав элементы верхнего уровня.

При обращении к элементу матрицы в ячейке один, т.е. списку [5, 6, 7, 8] функция len возвращает число 4, верно посчитав количество элементов вложенного списка.

Для списков len работает за константное время O(1). Внутри объекта списка всегда хранится количество элементов. Функция получает это значение, а не занимается подсчётом.

Задание

Перед вами список и несколько строк кода, которые его меняют. Напишите что вернёт каждая из строк кода. Попробуйте справится с заданием без запуска кода. У вас 3 минуты.

```
my_list = [2, 4, 6, 2, 8, 10, 12, 14, 16, 18]
print(my_list[2:6:2])
print(my_list.pop())
print(my_list.extend([314, 42]))
print(my_list.sort(reverse=False))
print(my_list)
```

2. Строки, str

И снова строки, но на этот раз как массивы символов. Часть рассмотренной для списков информации аналогична и для строки. Например, обращение к элементу строки по индексу в квадратных скобках, срезы строк и т.п. При этом стоит помнить, что строка неизменяема.

Работа со строками как с массивами

Начнём с квадратных скобок.

```
text = 'Hello world!'
print(text[6])
print(text[3:7])
```

Индексы и срезы работают аналогично спискам.
Если необходимо заменить элемент новым, индексы не подойдут. Для этих целей нужен метод replace

```
new_txt = text.replace('l', 'L', 2)
print(text, new_txt, sep='\n')
```

Первый аргумент — подстрока, которую нужно заменить.
Второй аргумент — подстрока, на которую нужно заменить.
Третий аргумент — максимальное количество замен. Если его не указывать, будут заменены все совпадения.

Методы count, index, find

Как и у списка, строка поддерживает методы count для подсчёта вхождения и index для поиска элемента. Но у строки появился и новый метод find. Он работает аналогично index. Но если искомая подстрока отсутствует, вместо ошибки возвращает -1.

```
text = 'Hello world!'
print(text.count('l'))
print(text.index('l'))
print(text.find('l'))
print(text.find('z'))
```

Реверс строк

Для разворота строки используется обратный срез, как и в случае со списком.

```
text = 'Hello world!'
print(text[::-1])
```

Форматирование строк

Опытные программисты могут назвать 5-7 способов форматирования строк. Разбирать их все нет смысла. Рассмотрим три “главных” на примерах.

Форматирование через %

Форматирование с использованием символа % является старым способом указания формата. Его вы можете встретить в коде, который писали очень давно. В настоящее время он используется лишь в некоторых модулях для задания формата вывода данных.

```
name = 'Alex'
age = 12
text = 'Меня зовут %s и мне %d лет' % (name, age)
print(text)
```

В строке текста используется знак % с символом типа после него. s — строка, d — число и т.п. После строки указывается символ % и перечисляются переменные. Если переменных больше одной, они заключаются в круглые скобки и разделяются запятой — передаётся кортеж.

🔥 **Важно!** Подробнее про используемы литеры вы можете прочитать по ссылке в материалах лекции. Пока же договоримся не использовать данный стиль форматирования, если это не обязывает конкретный модуль.

Метод format

Метод формат является строковым методом и позволяет соединять заранее заготовленный текст с переменными. Долгое время был основным способом форматирования. До версии Python 3.6, если быть точным.

```
name = 'Alex'
age = 12
text = 'Меня зовут {} и мне {} лет'.format(name, age)
print(text)
```

В строке используются фигурные скобки как место для подстановки значений. Далее для строки вызывается метод format. В качестве аргументов метод получает нужное количество переменных.

f-строка

Начиная с Python 3.7 для форматирования текста используют f-строки. Они работают быстрее, чем старые способы форматирования. А некоторые разработчики языка предлагают сделать их строками по умолчанию в одном из будущих релизов.

f-строки похожи на более короткую и читаемую запись метода формат.

```
name = 'Alex'
age = 12
text = f'Меня зовут {name} и мне {age} лет'
print(text)
```

Перед открывающей кавычкой пишут f — указатель на отформатированную строку. Текст внутри фигурных скобок воспринимается как переменная и на печать выводятся из значения.

🔥 **Важно!** Для печати фигурных скобок используется две фигурные скобки слитно.

```
print(f'{{Фигурные скобки}} и {{name}}')
```

Помимо вывода содержимого переменной можно указать дополнительные символы, влияющие на представление.

Уточнение формата

Существую различные способы уточнения способа вывода значения переменной.

```
pi = 3.1415
print(f'Число Пи с точностью два знака: {pi:.2f}')

data = [3254, 4364314532, 43465474, 2342, 462256, 1747]

for item in data:
    print(f'{item:>10}')
```

```
num = 2 * pi * data[1]
print(f'{num = :_}')
```

После указания имени переменной в фигурных скобках ставится двоеточие — указатель на символы задания формата далее.

- :.2f — число пи выводим с точность два знака после запятой
- :>10 — элементы списка выводятся с выравниванием по правому краю и общей шириной вывода в 10 символов
- = — выводим имя переменной, знак равенства с пробелами до и после него и только потом значение.
- :_ — число разделяется символом подчёркивания для деления на блоки по 3 цифры.

Про эти и другие способы форматирования можно почитать в официальной документации. Ссылка в материалах.

Методы строк

Рассмотрим ещё несколько методов, которые есть только у строк.

Метод split

Метод split позволяет разбить строку на отдельные элементы в соответствии с разделителем и поместить результат в список.

```
link = 'https://habr.com/ru/users/dzhoker1/posts/'
urls = link.split('/')
print(urls)

a, b, c = input('Введите 3 числа через пробел: ').split()
print(c, b, a)
```

В первом случае мы взяли ссылку и разделили её на отдельные компоненты по символу “/”. Обратите внимание, что между двойным слешем мы получили пустую строку. И вторую пустую строку после последнего слеша.

Во втором примере метод не получил на вход аргументы и деление происходит по пробельным символам. Три переменные до знака равно примут по одному из переданных значений.

🔥 **Важно!** С одной стороны удобно запросить у пользователя три значения в одной строке кода. Но стоит пользователю ошибится с пробелами и ввести меньше или больше трёх чисел, получим ошибку: `ValueError: too many values to unpack (expected 3)` или `ValueError: not enough values to unpack (expected 3, got 2)`

Один из способов избежать ошибки лишних (но не меньших) данных при распаковке методом split — использовать символ распаковки.

```
a, b, c, *_ = input('Введите не менее трёх чисел через пробел: ').split()
```

Переменная подчёркивание благодаря звёздочке, символу упаковки, превращается в список, который заберет значения начиная с четвёртого.

Метод join

Метод join принимает на вход итерируемую последовательность и соединяет все её элементы в строку, разделяя каждый текстом, к которому применён метод. В некоторой степени join противоположен split.

```
data = ['https:', '', 'habr.com', 'ru', 'users', 'dzhoker1', 'posts']
url = '/'.join(data)
print(url)
```

К строке “/” применили метод, т.е. каждый элемент списка будет разделён слешем. При этом в начале и в конце получившейся строки слеша не будет.

Методы upper, lower, title, capitalize

При работе с текстом можно быстро менять строчные буквы на прописные и наоборот.

```
text = 'однажды в СТУДЁНУЮ зИмнЮЮ ПоРУ'
print(text.upper())
print(text.lower())
print(text.title())
print(text.capitalize())
```

- upper — все символы приводятся к верхнему регистру
- lower — все символы приводятся к нижнему регистру
- title — первый символ каждого слова (разделитель слов - пробел) приводится к верхнему регистру, остальные символы к нижнему
- capitalize — первый символ строки в верхнем регистре, остальные в нижнем

Методы startswith и endswith

Метод startswith проверяет начинается ли строка с заданной подстроки. Метод возвращает истину или ложь. Метод endswith проверяет окончание строки переданной в качестве аргумента подстрокой.

```
text = 'Однажды в студёную зимнюю пору'
print(text.startswith('Однажды'))
print(text.endswith('зимнюю', 0, -5))
```

Оба метода помимо подстроки могут принимать параметры start и stop. В этом случае проверка начала либо конца будет проводиться в указанном диапазоне.

Задание

Перед вами строка текста и несколько строк кода, которые её меняют. Напишите что вернёт каждая из строк кода. Попробуйте справится с заданием без запуска кода. У вас 3 минуты.

```
text = 'Привет, мир!'
print(text.find(' '))
print(text.title())
print(text.split())
print(f'{text = :>25}')
```

3. Кортеж, tuple

Кортежи — это неизменяемые последовательности, обычно используемые для хранения коллекций разнородных данных. Также используются в случаях, когда требуется неизменяемая последовательность однородных данных. Как и строку кортеж нельзя изменить после создания. При этом кортеж как и список является массивом указателей на объекты любого типа.

Способы создания кортежа

Создать кортеж можно четырьмя способами.

```
a = ()
b1 = 1,
b2 = (1,)
c1 = 1, 2, 3,
c2 = (1, 2, 3)
d = tuple(range(3))
print(a, b1, b2, c1, c2, d, sep='\n')
```

1. Пара круглых скобок создаёт пустой кортеж
2. Один элемент с замыкающей запятой в скобках или без них создаёт кортеж с элементом
3. Несколько элементов разделенных запятыми с замыкающей запятой или в круглых скобках
4. Функция tuple(), которой передаётся любой итерируемый объект

🔥 **Важно!** Обратите внимание, что на самом деле кортеж образует запятая, а не круглые скобки. Круглые скобки необязательны, за исключением случая пустого кортежа или когда они необходимы, чтобы избежать синтаксической неоднозначности. Например, `f(a, b, c)` — это вызов функции с тремя аргументами. `f((a, b, c))` — вызов функции с кортежем в качестве единственного аргумента.

Кортежи реализуют все общие операции последовательностей

- Обращение к элементу по индексу
- Срезы
- Методы, которые работают с последовательностью, но не меняют её: count, index, а также функция len()

Задание

Убедимся, что вы сможете провести параллель между кортежами и списками, строками. Перед вами кортеж и несколько строк кода. Напишите, что вернёт каждая из них. Попробуйте справится с заданием без запуска кода. У вас 3 минуты.

```
my_tuple = (2, 4, 6, 2, 8, 10, 12, 14, 16, 18)
print(my_tuple[2:6:2])
print(my_tuple[-3])
print(my_tuple.count(2))
print(f'{my_tuple = }')
print(my_tuple.index(2, 2))
print(type('text',))
```

4. Словарь, dict

В Python есть изменяемый тип данных словарь. В других языках аналогичная структура данных может называться отображение, mapping, именованный массив, ассоциативный массив, сопоставление и т.п. Словарь представляет набор пар ключ-значение. Ключ — любой неизменяемый тип данных. Значение - любой тип данных. Обращаясь к ключу словаря получают доступ к значению.

🔥 **Важно!** Ключ выступает источником для вычисления хеша. Полученный хеш играет роль числового индекса и указывает на ячейку со значением. В Python вычисление хеша возможно лишь у неизменяемых типов данных. Следовательно, ключ словаря обязан быть неизменяемым объектом. Обычно это строка, целое число (вещественные лучше не использовать, вы же помните о точности округления), либо кортеж или неизменяемое множество.

Способы создания словаря

Для создания словаря есть несколько способов. Например:

- передать набор пар ключ-значение в фигурных скобках,
- использовать знак равенства между ключом и значением,
- передать любую последовательность, каждый элемент который пара ключ и значение

```
a = {'one': 42, 'two': 3.14, 'ten': 'Hello world!'}
b = dict(one=42, two=3.14, ten='Hello world!')
c = dict([('one', 42), ('two', 3.14), ('ten', 'Hello world!')])
print(a == b == c)
```

Все три способа создают одинаковые словари.

🔥 **Важно!** Вариант b не допускает использования зарезервированных слов. При этом ключи указываются без кавычек, но в словаре становятся ключами типа str.

Добавление нового ключа

Для добавления в существующий словарь новой пары ключ-значение можно использовать обычную операцию присваивания.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
my_dict['ten'] = 10
print(my_dict)
```

Доступ к значению словаря

Доступ через квадратные скобки []

Для получения доступа к значению необходимо указать ключ в квадратных скобках после или переменной.

```
TEN = 'ten'
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
print(my_dict['two'])
print(my_dict[TEN])
print(my_dict[1])  # KeyError: 1
```

Ключ может быть указан явно или передам как содержимое переменной, константы. При попытке обратиться к несуществующему ключу получаем ошибку: KeyError.
Доступ к ключу позволяет изменять значения. Для этого используем операцию присваивания как и в случае с добавлением новой пары ключ-значение.

🔥 **Важно!** Получить доступ к ключу по значению невозможно.

Доступ через метод get

Если ли мы хотим гарантировать отсутствие ошибки KeyError при обращении к элементу словаря, можно обратиться к значению через метод get, а не квадратные скобки.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
print(my_dict.get('two'))
print(my_dict.get('five'))
print(my_dict.get('five', 5))
print(my_dict.get('ten', 5))
```

При обращении к существующему ключу метод get работает аналогично доступу к через квадратные скобки. Если обратиться к несуществующему ключу, get возвращает None. Метод get принимает второй аргумент, значение по умолчанию. Если ключ отсутствует в словаре, вместо None будет возвращено указанное значение.

Часто используемые методы словарей

Разберем некоторые методы работы со словарями.

Метод setdefault

Метод setdefault похож не get, но отсутствующий ключ добавляется в словарь.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
spam = my_dict.setdefault('five')
print(f'{spam = }\t{my_dict=}')
eggs = my_dict.setdefault('six', 6)
print(f'{eggs = }\t{my_dict=}')
new_spam = my_dict.setdefault('two')
print(f'{new_spam=}\t{my_dict=}')
new_eggs = my_dict.setdefault('one', 1_000)
print(f'{new_eggs=}\t{my_dict=}')
```

При вызове метода с одним аргументом отсутствующий ключ добавляется в словарь. В качестве значения передаётся None. Если указать два аргумента и ключ отсутствует, второй аргумент становится значением ключа и также добавляется в словарь. При обращении к существующему ключу, словарь не изменяется независимо от того указанные один или два аргумента.

Метод keys

Метод keys возвращает объект-итератор dict_keys.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
print(my_dict.keys())
for key in my_dict.keys():
    print(key)
```

Обычно объект не используют напрямую. Метод keys применяется в связке с циклом for для перебора ключей словаря.

🔥 **Важно!** Запись цикла `for key in my_dict:` отработает аналогично. По умолчанию словарь возвращает ключи для итерации в цикле.

💡 **Внимание!** В отличии от списков, кортежей и строк доступ к элементу-значению осуществляется не по индексу, а по ключу. При этом начиная с версии Python 3.7 словарь сохраняет порядок добавления ключей. В

каком порядке ключи были добавлены, в том порядке они будут возвращены в случае итерации по словарю.

Метод values

Метод values похож на keys, но возвращает значения в виде объекта итератора dict_values, а не ключи.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
print(my_dict.values())
for value in my_dict.values():
    print(value)
```

Метод items

Если в цикле необходимо работать одновременно с ключами и значениями, как с парами, используют метод items.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
print(my_dict.items())
for tuple_data in my_dict.items():
    print(tuple_data)
for key, value in my_dict.items():
    print(f'{key = } value before 100 - {100 - value}')
```

Метод возвращает объект итератор `dict_items`. Если создать цикл for с одной переменной между for и in, получим кортеж из пар элементов — ключа и значения. Обычной используют две переменные в цикле: первая принимает ключ, а вторая значение. Такой подход облегчает чтение кода и позволяют использовать ключ и значение по-отдельности.

Метод popitem

Для удаления пары ключ значение из словаря используют метод popitem.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
spam = my_dict.popitem()
print(f'{spam = }\t{my_dict=}')
eggs = my_dict.popitem()
print(f'{eggs = }\t{my_dict=}')
```

Так как словари сохраняют порядок добавления ключей, удаление происходит справа налево, по методу LIFO. Элементы удаляются в обратном добавлению порядке.

🔥 **Важно!** Если измените значение у существующего ключа, положение ключа в очереди не меняется, он не считается последним добавленным.

Метод pop

Метод pop удаляет пару ключ-значение по переданному ключу.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
spam = my_dict.pop('two')
print(f'{spam = }\t{my_dict=}')
err = my_dict.pop('six')  # KeyError: 'six'
err = my_dict.pop()  # TypeError: pop expected at least 1 argument, got 0
```

Если указать несуществующий ключ, получим ошибку KeyError. В отличии от метода pop у списков list, dict.pop вызовет ошибку TypeError. Для удаление последнего элемента нужен метод popitem.

Метод update

Для расширение словаря новыми значениями используют метод update.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
my_dict.update(dict(six=6))
print(my_dict)
my_dict.update(dict([('five', 5), ('two', 42)]))
print(my_dict)
```

На вход метод получает другой словарь в любой из вариаций создания словаря. Если передать существующий ключ, значение будет заменено новым.

Ещё один способ создать словари из нескольких других, который появился в новой версии Python — вертикальная черта.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
new_dict = my_dict | {'five': 5, 'two': 42} | dict(six=6)
print(new_dict)
```

При перезаписи совпадающих ключей приоритет отдаётся словарю, расположенному правее.

Задание

Перед вами словарь и несколько строк кода. Напишите что вернёт каждая из строк. Попробуйте справится с заданием без запуска кода. У вас 3 минуты.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
print(my_dict.setdefault('ten', 555))
print(my_dict.values())
print(my_dict.pop('one'))
my_dict['one'] = my_dict['four']
print(my_dict)
```

5. Множества set и frozenset

Ещё одна коллекция из коробки — множества. Множество — набор уникальных неиндексированных элементов. В Python есть два вида множеств: set — изменяемое множество, frozenset — неизменяемое множество. Неизменяемое множество позволяет вычислять хеш и может использоваться там, где разрешён

лишь хешированный тип данных, например в качестве ключа словаря.

```
my_set = {1, 2, 3, 4, 2, 5, 6, 7}
print(my_set)
my_f_set = frozenset((1, 2, 3, 4, 2, 5, 6, 7,))
print(my_f_set)
not_set = {1, 2, 3, 4, 2, 5, 6, 7, ['a', 'b']}  # TypeError: unhashable type: 'list'
```

Обратите внимание, что двойка передавалась в множества дважды, но хранится в единственном экземпляре, как один из уникальных элементов

🔥 **Важно!** Элементом множества могут быть только неизменяемые типы данных.

Методы множеств

Рассмотрим некоторые методы множеств на примере изменяемого множества set. Все методы, которые не изменяют оригинал, работают аналогично и для множества frozenset.

Метод add

Метод add работает аналогично методу списка append, т.е. добавляет один элемент в коллекцию.

```
my_set = {3, 4, 2, 5, 6, 1, 7}
my_set.add(9)
print(my_set)
my_set.add(7)
print(my_set)
my_set.add(9, 10)    # TypeError: set.add() takes exactly one argument (2 given)
my_set.add((9, 10))
print(my_set)
```

Если попытаться добавить уже существующий элемент, множество не изменится. При попытке добавить несколько элементов получим ошибку TypeError.

🔥 **Внимание!** Если передать в метод add неизменяемую коллекцию, например кортеж, коллекция будет добавлена как один целостный объект.

Метод remove

Для удаления элемента множества используют метод remove.

```
my_set = {3, 4, 2, 5, 6, 1, 7}
my_set.remove(5)
print(my_set)
my_set.remove(10)  # KeyError: 10
```

При передаче несуществующего объекта получим ошибку KeyError.

Метод discard

Метод discard работает аналогично remove — удаляет один элемент множества.

```
my_set = {3, 4, 2, 5, 6, 1, 7}
my_set.discard(5)
print(my_set)
my_set.discard(10)
```

В отличии от remove при попытке удалить несуществующий элемент discard не вызывает ошибку. При этом множество не изменяется.

Метод intersection

Для получения пересечения множеств, т.е. множества с элементами, которые есть и в левом и в правам множестве используют метод intersection

```
my_set = {3, 4, 2, 5, 6, 1, 7}
other_set = {1, 4, 42, 314}
new_set = my_set.intersection(other_set)
print(f'{my_set = }\n{other_set = }\n{new_set = }')
```

Новая версия Python позволяет получить пересечение множеств в следующей записи с использованием символа &

```
my_set = {3, 4, 2, 5, 6, 1, 7}
other_set = {1, 4, 42, 314}
new_set = my_set & other_set
print(f'{my_set = }\n{other_set = }\n{new_set = }')
```

🔥 **Внимание!** Исходные множества при пересечении не изменяются.

Метод union

Для объединения множеств используется метод union.

```
my_set = {3, 4, 2, 5, 6, 1, 7}
other_set = {1, 4, 42, 314}
new_set = my_set.union(other_set)
print(f'{my_set = }\n{other_set = }\n{new_set = }')
new_set_2 = my_set | other_set
print(f'{my_set = }\n{other_set = }\n{new_set_2 = }')
```

На выходе получаем множество уникальных элементов из левого и правого множеств. Более короткая запись объединения возможна при помощи вертикальной черты.

Метод difference

Метод difference удаляет из левого множества элементы правого.

```
my_set = {3, 4, 2, 5, 6, 1, 7}
other_set = {1, 4, 42, 314}
new_set = my_set.difference(other_set)
```

```
print(f'{my_set = }\n{other_set = }\n{new_set = }')
new_set_2 = my_set - other_set
print(f'{my_set = }\n{other_set = }\n{new_set_2 = }')
```

На выходе получаем множество элементов встречающихся только в левом множестве. Более короткая запись возможно при помощи знака минус. Вычитаем из левого элементы правого.

Проверка на вхождение, in

Для проверки входит ли элемент в множество используют зарезервированное слово in.

```
my_set = {3, 4, 2, 5, 6, 1, 7}
print(42 in my_set)
```

🔥 **Внимание!** Слово in позволяет сделать проверку на вхождение и в других коллекциях. Входит ли объект в list, tuple, является ли подстрока частью строки str, встречается ли ключ в словаре. Для list, tuple, str проверка на вхождение работает за линейное время O(n). Для dict, set, frozenset проверка работает за константное время O(1).

Задание

Перед вами множество и несколько строк кода. Напишите что вернёт каждая из строк. Попробуйте справится с заданием без запуска кода. У вас 3 минуты.

```
my_set = frozenset({3, 4, 1, 2, 5, 6, 1, 7, 2, 7})
print(len(my_set))
print(my_set - {1, 2, 3})
print(my_set.union({2, 4, 6, 8}))
print(my_set & {2, 4, 6, 8})
print(my_set.discard(10))
```

6. Классы bytes и bytearray

Уделим несколько строк лекции неизменяемым байтам и их изменяемой версии — массиву байт. Для отправки информации по каналам связи объекты не подойдут. Даже текст не отправить. А вот пересылать байты — легко.

```
text_en = 'Hello world!'
res = text_en.encode('utf-8')
print(res, type(res))

text_ru = 'Привет, мир!'
res = text_ru.encode('utf-8')
print(res, type(res))
```

Строковой метод encode получает в качестве аргумента указание кодировки. На выходе получаем строку байт. Функция print возвращает строковое представление байт, сами ячейки памяти с электронами невозможно увидеть невооруженным глазом.

```
b'Hello world!'
b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82,
\xd0\xbc\xd0\xb8\xd1\x80!'
```

Префикс b говорит о том, что перед нами не строка, а байты. Если байт может быть представлен как символ, т.е. он есть в семибитной кодировке ASCII. отображается символ. В остальных случаях указывается приставка \x и слитно с ней шестнадцатеричное представление байта.

Для получения набора байт можно использовать функцию bytes. А если необходимо изменять байт, использую функцию bytearray.

```
x = bytes(b'\xd0\x9f\xd1\x80\xd0\xb8')
y = bytearray(b'\xd0\x9f\xd1\x80\xd0\xb8')
print(f'{x = }\n{y = }')
```

В качестве аргумента передаётся строковое представление нужным байт.

Классы байт и массив байт обладают практически всеми методами строк. Кроме того для массива байт доступны методы модификации списка list.

7. Вывод

На этой лекции мы:

1. Разобрали, что такое коллекция и какие коллекции есть в Python
2. Изучили работу со списками, как с самой популярной коллекцией
3. Узнали, как работать со строкой в ключе коллекция
4. Разобрали работу с кортежами
5. Узнали, что такое словари и как с ними работать
6. Изучили множества и особенности работы с ними
7. Познакомились с классами байт и массив байт

Краткий анонс следующей лекции

1. Разберёмся с созданием собственных функций в Python
2. Изучим работу встроенных функций

Домашнее задание

1. Поработайте со справочной информацией в Python, функцией help(). Попробуйте найти дополнительную информацию о изученных на уроке коллекциях. Если вы плохо читаете на английском, воспользуйтесь любым онлайн переводчиком.
2. Проведите несколько экспериментов с классами bytes и bytearray. Посмотрите на их поведение как у строки и на списочные методы bytearray.