Компьютерные технологии

Лекция № 3.
Структуры данных

Нижний
Новгород
2022 г.

Структуры данных – это структуры, которые могут хранить некоторые данные вместе.

Они используются для хранения связанных данных.

В Python существуют четыре встроенных структуры данных:

- **список**,
- **кортеж**,
- **словарь**,
- **множество**.

Списки

Большинство программ работает не с отдельными переменными, а с набором переменных.

Например, программа может обрабатывать информацию об учащихся класса, считывая список учащихся с клавиатуры или из файла, при этом изменение количества учащихся в классе не должно требовать модификации исходного кода программы.

Список – последовательность элементов, пронумерованных от 0, как символы в строке.

Списки

Список можно задать перечислением элементов списка в квадратных скобках, например, список можно задать так:

Primes = [2, 3, 5, 7, 11, 13]

Rainbow = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']

В списке **Primes** — 6 элементов, а именно, Primes[0] == 2, Primes[1] == 3, Primes[2] == 5, Primes[3] == 7, Primes[4] == 11, Primes[5] == 13.
Список **Rainbow** состоит из 7 элементов, каждый из которых является строкой.

Списки

Также как и символы строки, элементы списка можно индексировать отрицательными числами с конца, например,

Primes[-1] == 13,

Primes[-6] == 2.

Длину списка, то есть количество элементов в нем, можно узнать при помощи функции len, например, len(A) == 6.

Списки

Можно создать пустой список (не содержащий элементов, длины 0), в конец списка можно добавлять элементы при помощи метода **append**. Например, если программа получает на вход количество элементов в списке **n**, а потом **n** элементов списка по одному в отдельной строке, то организовать считывание списка можно так:

```
A = []
for i in range(int(input()):
    A.append(int(input())
```

В этом примере создается пустой список, далее считывается количество элементов в списке, затем по одному считываются элементы списка и добавляются в его конец.

Списки

Для списков целиком определены следующие операции: **конкатенация списков** (добавление одного списка в конец другого) и **повторение списков** (умножение списка на число).

A = [1, 2, 3]

B = [4, 5]

C = A + B

D = B * 3

В результате список **C** будет равен **[1, 2, 3, 4, 5]**, а список **D** будет равен **[4, 5, 4, 5, 4, 5]**.

Списки

Это позволяет по-другому организовать процесс считывания списков:

сначала считать размер списка и создать список из нужного числа элементов,

затем организовать цикл по переменной *i* начиная с числа 0 и внутри цикла считывается i-й элемент списка.

```
A = [0] * int(input())
for i in range(len(A)):
    A[i] = int(input())
```

Списки

Вывести элементы списка A можно одной инструкцией **print(A)**, при этом будут выведены квадратные скобки вокруг элементов списка и запятые между элементами списка.

Такой вывод неудобен, чаще требуется просто вывести все элементы списка в одну строку или по одному элементу в строке.

Приведем два примера, отличающиеся организацией цикла:

```
for i in range(len(A)):
    print(A[i])
```

Здесь в цикле меняется индекс элемента i, затем выводится элемент списка с индексом i.

Списки

```
for elem in A:
    print(elem, end = ' ')
```

В этом примере элементы списка выводятся в одну строку, разделенные пробелом, при этом в цикле меняется не индекс элемента списка, а само значение переменной

Например, в цикле

```
for elem in ['red', 'green', 'blue']
```

переменная elem будет последовательно принимать значения 'red', 'green', 'blue'.

Методы split и join

Элементы списка могут вводиться по одному в строке, в этом случае строку можно считать функцией **input()**. После этого можно использовать метод строки **split**, возвращающий список строк, разрезав исходную строку на части по пробелам. Пример:

```
A = input().split()
```

Если при запуске этой программы ввести строку 1 2 3, то список A будет равен ['1', '2', '3'].

Обратите внимание, что список будет состоять из строк, а не из чисел.

Методы split и join

Если хочется получить список именно из чисел, то можно затем элементы списка по одному преобразовать в числа:

```
for i in range(len(A)):
    A[i] = int(A[i])
```

Используя функции языка **map** и **list** то же самое можно сделать в одну строку:

```
A = list(map(int, input().split()))
```

Если нужно считать список действительных чисел, то нужно заменить тип **int** на тип **float**.

Методы split и join

У метода **split** есть необязательный параметр, который определяет, какая строка будет использоваться в качестве разделителя между элементами списка. Например, метод split('.') вернет список, полученный разрезанием исходной строки по символам '.'.

Используя “обратные” методы можно вывести список при помощи однострочной команды.

Для этого используется метод строки **join**. У этого метода один параметр: список строк.

В результате получается строка, полученная соединением элементов списка (которые переданы в качестве параметра) в одну строку, при этом между элементами списка вставляется разделитель, равный той строке, к которой применяется метод.

Методы split и join

Например программа

```
A = ['red', 'green', 'blue']
print(' '.join(A))
print(''.join(A))
print('***'.join(A))
```

выведет строки

'red green blue',

redgreenblue и red***green***blue.

Если же список состоит из чисел, то придется использовать еще и функцию map. То есть вывести элементы списка чисел, разделяя их пробелами, можно так:

```
print(' '.join(map(str, A)))
```

Генераторы списков

Для создания списка, заполненного одинаковыми элементами, можно использовать оператор повторения списка:

A = [0] * n

Для создания списков, заполненных по более сложным формулам можно использовать ***генераторы***: выражения, позволяющие заполнить список некоторой формулой. Общий вид генератора следующий:

[*выражение* for *переменная* in *список*]

где ***переменная*** — идентификатор некоторой переменной, ***список*** — список значений, который принимает данная переменная (как правило, полученный при помощи функции range), ***выражение*** — некоторое выражение, которым будут заполнены элементы списка, как правило, зависящее от использованной в генераторе переменной.

Генераторы списков

Создать список, состоящий из **n** нулей можно и при помощи генератора:

```
A = [ 0 for i in range(n)]
```

Создать список, заполненный квадратами целых чисел можно так:

```
A = [ i ** 2 for i in range(n)]
```

Если нужно заполнить список квадратами чисел от 1 до n, то можно изменить параметры функции range на range(1, n + 1):

```
A = [ i ** 2 for i in range(1, n + 1)]
```

Генераторы списков

Вот так можно получить список, заполненный случайными числами от 1 до 9 (используя функцию randint из модуля random):

```
A = [ randint(1, 9) for i in range(n)]
```

А в этом примере список будет состоять из строк, считанных со стандартного ввода: сначала нужно ввести число элементов списка (это значение будет использовано в качестве аргумента функции range), потом — заданное количество строк:

```
A = [ input() for i in range(int(input()))]
```

Кортеж

Кортежи служат для хранения нескольких объектов вместе. Их можно рассматривать как аналог списков, но без такой обширной функциональности, которую предоставляет класс списка.

Одна из важнейших особенностей кортежей заключается в том, что они **неизменяемы**, так же, как и строки. Т.е. модифицировать кортежи невозможно.

Кортежи обозначаются указанием элементов, разделённых запятыми;

по желанию их можно ещё заключить в круглые скобки. Кортежи обычно используются в тех случаях, когда оператор или пользовательская

Кортеж

```
zoo = ('питон', 'слон', 'пингвин') # помните, что скобки не обязательны
print('Количество животных в зоопарке -', len(zoo))
new_zoo = 'обезьяна', 'верблюд', zoo
print('Количество клеток в зоопарке -', len(new_zoo))
print('Все животные в новом зоопарке:', new_zoo) print('Животные, привезённые из старого зоопарка:', new_zoo[2])
print('Последнее животное, привезённое из старого зоопарка -', new_zoo[2][2])
print('Количество животных в новом зоопарке -', len(new_zoo)-1+len(new_zoo[2]))
```

Вывод:

Количество животных в зоопарке - 3

Количество клеток в зоопарке - 3

Все животные в новом зоопарке: ('обезьяна', 'верблюд', ('питон', 'слон', 'пингвин'))

Животные, привезённые из старого зоопарка: ('питон', 'слон', 'пингвин')
Последнее животное, привезённое из старого зоопарка - пингвин
Количество животных в новом зоопарке - 5

Кортеж, содержащий 0 или 1 элемент

Пустой кортеж создаётся при помощи пустой пары скобок – “myempty = ()”.

Однако, с кортежем из одного элемента не всё так просто. Его нужно указывать при помощи запятой после первого (и единственного) элемента, чтобы Python мог отличить кортеж от скобок, окружающих объект в выражении.

Таким образом, чтобы получить кортеж, содержащий элемент 2, вам потребуется указать

“singleton = (2,)”.

Словари

Обычные списки (массивы) представляют собой набор пронумерованных элементов, то есть для обращения к какому-либо элементу списка необходимо указать его номер. Номер элемента в списке однозначно идентифицирует сам элемент. Но идентифицировать данные по числовым номерам не всегда оказывается удобно.

Например, маршруты поездов в России идентифицируются численно-буквенным кодом (число и одна цифра), также численно-буквенным кодом идентифицируются авиарейсы, то есть для хранения информации о рейсах поездов или самолетов в качестве идентификатора удобно было бы использовать не число, а текстовую

Словари

Структура данных, позволяющая идентифицировать ее элементы не по числовому индексу, а по произвольному, называется ***словарем*** или ***ассоциативным массивом***. Соответствующая структура данных в языке Питон называется **dict**.

Рассмотрим простой пример использования словаря. Заведем словарь **Capitals**, где индексом является название страны, а значением — название столицы этой страны. Это позволит легко определять по строке с названием страны ее столицу.

```
# Создадим пустой словать Capitals
Capitals = dict()
# Заполним его несколькими значениями
Capitals['Russia'] = 'Moscow‘
Capitals['Ukraine'] = 'Kiev'
Capitals['USA'] = 'Washington'
```

Словари

```
# Считаем название страны
print('В какой стране вы живете?')
country = input()
# Проверим, есть ли такая страна в словаре Capitals
if country in Capitals:
# Если есть - выведем ее столицу
    print('Столица вашей страны', Capitals[country])
else:
# Запросим название столицы и добавив его в словарь
print('Как называется столица вашей страны?')
    city = input()
    Capitals[country] = city
```

Словари

Итак, каждый элемент словаря состоит из двух объектов: ***ключа*** и ***значения***. В нашем примере ключом является название страны, значением является название столицы. Ключ идентифицирует элемент словаря, значение является данными, которые соответствуют данному ключу. **Значения ключей — уникальны**, двух одинаковых ключей в словаре быть не может.

В жизни широко распространены словари, например, привычные бумажные словари (толковые, орфографические, лингвистические).

В них **ключом** является слово-заголовок статьи, а **значением** — сама статья. Для того, чтобы получить доступ к статье, необходимо указать слово-ключ.

Словари

Другой пример словаря — телефонный справочник. В нем **ключом** является **имя**, а **значением** — **номер телефона**. И словарь, и телефонный справочник хранятся так, что легко найти элемент словаря по известному ключу (например, если записи хранятся в алфавитном порядке ключей, то легко можно найти известный ключ, например, бинарным поиском), но если ключ неизвестен, а известно лишь значение, то поиск элемента с данным значением может потребовать последовательного просмотра всех элементов словаря.

Особенностью ассоциативного массива является его **динамичность**: в него можно **добавлять** новые элементы с произвольными ключами и **удалять** уже существующие элементы. При этом **размер используемой памяти** пропорционален **размеру ассоциативного массива**. Доступ к элементам ассоциативного массива выполняется хоть и **медленнее**, чем к обычным массивам, но в целом

Словари

В языке Питон как ключом **может** быть произвольный неизменяемый тип данных:

целые и действительные числа,

строки,

кортежи.

Ключом в словаре **не** может быть **множество**,

но **может** быть элемент типа **frozenset**: специальный тип данных, являющийся аналогом типа **set**, который нельзя изменять после создания.

Значением элемента словаря может быть любой тип данных, в том числе и изменяемый.

Когда нужно использовать словари

- Подсчет числа каких-то объектов. В этом случае нужно завести словарь, в котором ключами являются объекты, а значениями — их количество.
- Хранение каких-либо данных, связанных с объектом. Ключи — объекты, значения — связанные с ними данные. Например, если нужно по названию месяца определить его порядковый номер, то это можно сделать при помощи словаря Num['January'] = 1; Num['February'] = 2; ....
- Установка соответствия между объектами (например, “родитель—потомок”). Ключ — объект, значение — соответствующий ему объект.
- Если нужен обычный массив, но при этом масимальное значение индекса элемента очень велико, но при этом будут использоваться не все возможные индексы (так называемый “разреженный массив”), то можно использовать ассоциативный массив для экономии

Создание словаря

Пустой словарь можно создать при помощи функции dict() или пустой пары фигурных скобок {} (вот почему фигурные скобки нельзя использовать для создания пустого множества). Для создания словаря с некоторым набором начальных значений можно использовать следующие конструкции:

```
Capitals = {'Russia': 'Moscow', 'Ukraine': 'Kiev', 'USA': 'Washington'}
Capitals = dict(Russia = 'Moscow', Ukraine = 'Kiev', USA = 'Washington')
Capitals = dict([("Russia", "Moscow"), ("Ukraine", "Kiev"), ("USA", "Washington")])
Capitals = dict(zip(["Russia", "Ukraine", "USA"], ["Moscow", "Kiev", "Washington"]))
```

Работа с элементами словаря

Основная операция: получение значения элемента по ключу, записывается так же, как и для списков:

A[key].

Если элемента с заданным ключом не существует в словаре, то возникает исключение **KeyError**.

Другой способ определения значения по ключу — метод get:

A.get(key).

Если элемента с ключом get нет в словаре, то возвращается значение None.

В форме записи с двумя аргументами A.get(key, val) метод возвращает значение val, если элемент с ключом key отсутствует в словаре.

Проверить принадлежность элемента словарю можно операциями in и not in, как и для множеств.

Работа с элементами словаря

Для добавления нового элемента в словарь нужно просто присвоить ему какое-то значение:

A[key] = value.

Для удаления элемента из словаря можно использовать операцию del A[key] (операция возбуждает исключение KeyError, если такого ключа в словаре нет.

Вот два безопасных способа удаления элемента из словаря.

- Способ с предварительной проверкой наличия элемента:

  if key in A: del A[key]

- Способ с перехватыванием и обработкой исключения:

  try: del A[key] except KeyError: pass

Работа с элементами словаря

Еще один способ удалить элемент из словаря: использование метода **pop**:

A.pop(key).

Этот метод возвращает значение удаляемого элемента, если элемент с данным ключом отсутствует в словаре, то возбуждается исключение.

Если методу **pop** передать второй параметр, то если элемент в словаре отсутствует, то метод **pop** возвратит значение этого параметра.

Это позволяет проще всего организовать безопасное удаление элемента из словаря:

A.pop(key, None).

Перебор элементов словаря

Можно легко организовать перебор ключей всех элементов в словаре:

```
for key in A: print(key, A[key])
```

Следующие методы возвращают ***представления* элементов** словаря. Представления во многом похожи на множества, но они изменяются, если менять значения элементов словаря.

Метод **keys** возвращает представление ключей всех элементов, метод **values** возвращает представление всех значений, а метод **items** возвращает представление всех пар (кортежей) из ключей и значений.

Перебор элементов словаря

Соответственно, быстро проверить, если ли значение **val** среди всех значений элементов словаря A можно так:

```
val in A.values(),
```

а организовать цикл так, чтобы в переменной **key** был ключ элемента, а в переменной **val** было его значение можно так:

```
for key, val in A.items():
    print(key, val)
```

Множества

Множество в языке Питон — это структура данных, эквивалентная множествам в математике. Множество может состоять из различных элементов, порядок элементов в множестве неопределен. В множество можно добавлять и удалять элементы, можно перебирать элементы множества, можно выполнять операции над множествами (объединение, пересечение, разность). Можно проверять принадлежность элементу множества.

В отличии от массивов, где элементы хранятся в виде последовательного списка, в множествах порядок хранения элементов неопределен (более того, элементы множества храняться не подряд, как в списке, а при помощи хитрых алгоритмов). Это позволяет выполнять операции типа “проверить принадлежность элемента множеству” быстрее, чем просто перебирая все элементы множества.

Множества

Элементами множества может быть любой неизменяемый тип данных: числа, строки, кортежи.

Изменяемые типы данных не могут быть элементами множества, в частности, нельзя сделать элементом множества список (но можно сделать кортеж) или другое множество.

Требование неизменяемости элементов множества накладывается особенностями представления множества в памяти компьютера.

Задание множеств

Множество задается перечислением всех его элементов в фигурных скобках. Например:

A = {1, 2, 3}

Исключением является пустое множество, которое можно создать при помощи функции **set()**.

Если функции set передать в качестве параметра список, строку или кортеж, то она вернет множество, составленное из элементов списка, строки, кортежа. Например:

A = set('qwerty') print(A)

выведет {'e', 'q', 'r', 't', 'w', 'y'}.

Задание множеств

Каждый элемент может входить в множество только один раз, порядок задания элементов не важен.

Например, программа:

A = {1, 2, 3} B = {3, 2, 3, 1}

print(A == B)

выведет

True,

так как A и B — равные множества.

Каждый элемент может входить в множество только один раз. set('Hello') вернет множество из четырех элементов: {'H', 'e', 'l', 'o'}.

Работа с элементами множеств

Узнать число элементов в множестве можно при помощи функции len.

Перебрать все элементы множества (в неопределенном порядке!) можно при помощи цикла for:

```
C = {1, 2, 3, 4, 5}
for elem in C:
    print(elem)
```

Проверить, принадлежит ли элемент множеству можно при помощи операции in, возвращающей значение типа bool:

```
i in A
```

Работа с элементами множеств

Проверить, принадлежит ли элемент множеству можно при помощи операции in, возвращающей значение типа bool:

i in A

Аналогично есть противоположная операция **not in**.

Для добавления элемента в множество есть метод **add**:

A.add(x)

Для удаления элемента x из множества есть два метода: **discard** и **remove**. Их поведение различается только в случае, когда удаляемый элемент отсутствует в множестве. В этом случае метод **discard** не делает ничего, а метод **remove** генерирует исключение **KeyError**.

Наконец, метод **pop** удаляет из множетсва один случайный элемент и возвращает его значение. Если же множество пусто, то генерируется исключение **KeyError**.

Из множества можно сделать список при помощи функции **list**.

Перебор элементов множества

При помощи цикла for можно перебрать все элементы множества:

```
Primes = {2, 3, 5, 7, 11}
for num im Primes:
    print(num)
```

Операции с множествами

С множествами в питоне можно выполнять обычные для математики операции над множествами.

A | B A.union(B)	Возвращает множество, являющееся объединением множеств A и B.
A |= B A.update(B)	Добавляет в множество A все элементы из множества B.
A & B A.intersection(B)	Возвращает множество, являющееся пересечением множеств A и B.
A &= B A.intersection_update(B)	Оставляет в множестве A только те элементы, которые есть в множестве B.
A - B A.difference(B)	Возвращает разность множеств A и B (элементы, входящие в A, но не входящие в B).
A -= B A.difference_update(B)	Удаляет из множества A все элементы, входящие в B.

Операции с множествами

A ^ B A.symmetric_difference(B)	Возвращает симметрическую разность множеств A и B (элементы, входящие в A или в B, но не в оба из них одновременно).
A ^= B A.symmetric_difference_update(B)	Записывает в A симметрическую разность множеств A и B.
A <= B A.issubset(B)	Возвращает true, если A является подмножеством B.
A >= B A.issuperset(B)	Возвращает true, если B является подмножеством A.
A < B	Эквивалентно A <= B and A != B
A > B	Эквивалентно A >= B and A != B