Тема 4. СТРУКТУРЫ ДАННЫХ

Оглавление

ТЕМА 4. СТРУКТУРЫ ДАННЫХ

Структуры используются для хранения связанных данных. В *Python* существуют четыре встроенных структуры данных: список, словарь, кортеж и множество. Модуль *collections* содержит очереди, деки и другие коллекции.

4.1 Списки

Список (*list*) – упорядоченный изменяемый набор элементов, каждый из которых имеет свой индекс, позволяющий быстро получить к нему доступ. Нумерация элементов в списке начинается с 0. Во многих языках программирования есть аналогичная структура данных, которая называется массив. Однако в списке одновременно могут храниться данные разных типов.

4.1.1 Создание списка

Создание списка может быть выполнено тремя способами: квадратными скобками [], конструктором *list* () и выражением-генератором.

В случае использования квадратных скобок [] для создания списка, элементы перечисляются через запятую. Без указания элементов будет создан пустой список:

```
numbers = [1, 2, 3, 4, 5]
cities = ['Ростов-на-Дону', 'Новочеркасск', 'Шахты']
my_list = [15, "Автор", None, 34.5, True]
```

Для отображения элементов списка можно использовать стандартную функцию print():

```
print(numbers) # [1, 2, 3, 4, 5]
```

Элементы списка будет выведены на экран внутри квадратных скобок через запятую.

Конструктор *list*() может принимать набор значений, на основе которых создается список. Если использовать его без аргумента, то он вернет нам пустой список:

```
new_list = list()
numbers = [1, 2, 3, 4, 5]
num_list1 = list(numbers)
languages = ['Python', 'C#', 'C++', 'Java']
```

```
str_list1 = list(languages)
print(new_list)
print(num_list1)
print(str_list1)
```

Результат:

```
[]
[1, 2, 3, 4, 5]
['Python', 'C#', 'C++', 'Java']
```

При передаче конструктору итерируемого объекта, например, строки или кортежа, будет сформирован список из элементов этого объекта:

```
str_list2 = list('language')
num_list2 = list((5, 8, 9, 0, 5))
print(str_list2)
print(num_list2)
```

Результат:

```
[5, 8, 9, 0, 5]
['l', 'a', 'n', 'g', 'u', 'a', 'g', 'e']
```

Генераторы списков (списочные выражения) очень похожи на цикл *for*, они позволяют построить новый список, применяя выражение к каждому элементу последовательности. Общий синтаксис конструкции имеет вид:

```
newlist = [expr for item in iterable (if condition)]
```

Синтаксис генератора состоит из следующих компонентов:

- *iterable*: перебираемый источник данных, в качестве которого может выступать список, множество, последовательность, либо даже функция, которая возвращает набор данных, например, *range*();
- *item*: извлекаемый из источника данных элемент;
- *expr*: выражение, которое возвращает некоторое значение. Это значение затем попадает в генерируемый список;
- *condition*: условие, которому должны соответствовать извлекаемые из источника данных элементы. Необязательный параметр.

Пример 1. Сгенерировать список целых чисел в диапазоне от 0 до 9, включительно:

```
my_list = [i for i in range(10)]
print (my_list)
```

Пример 2. Сгенерировать список целых чисел, кратных 30 или 31, в диапазоне от 20 до 180, включительно:

```
my_list = [i for i in range(20,181)
    if i%30 == 0 or i%31 == 0]
print (my_list)
```

Пример 3. Задана строка. Разделить числа и символы, образующие строку на два списка:

```
a = "lsj94ksd2319"
my_list = [int(i) for i in a if '0'<=i<='9']
print (my_list)
my_list = [i for i in a if i.isalpha()]
print (my_list)
```

Стоит заметить, что любой генератор списка можно реализовать через цикл *for*. Более того, в сложных случаях именно так и следует поступать. Но в простейших случаях генераторы списка интуитивно понятнее, работают быстрее цикла *fo*r и занимают чуть места по объему кода.

4.1.2 Обращение к элементам списка

Операции доступа по индексу и получения срезов для списков имеют практически тот же самый синтаксис и смысл, что и для строк.

Индексация начинается с нуля. Для обращения к элементам с конца можно использовать отрицательные индексы, начиная с -1. При попытке получить доступ к элементу с индексом, превышающим длину списка, будет выдана ошибка *IndexError*:

```
languages = ['Python', 'C#', 'C++', 'Java']
print (languages[0])        # Python
print (languages[1])        # C#
print (languages[-1])       # Java
print (languages[-5])       # IndexError
```

В результате выполнения операции получения среза всегда возвращается новый список, состоящий из указанных в условии элементов исходного списка.

Пример 1. Определить срез для получения каждого второго элемента (слева направо):

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print('my_list[0:7:2] ->', my_list[0:7:2])
```

Пример 2. Определить срез для получения каждого второго элемента, считая элементы с конца:

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print('my_list[8:2:-2] ->', my_list[8:2:-2])
```

Изменяемость списков означает возможность не только получения доступа к элементам списка, но и изменения их значения, включая добавление и удаление элементов. Изменению подвергается не копия списка, как в случае со строками, а непосредственно сам список.

4.1.3 Методы для работы со списками

В языке *Python* методы для работы со списками можно условно разделить на следующие группы:

- увеличение количества элементов;
- уменьшение количества элементов;
- изменение порядка элементов;
- поиск элемента;
- копирование списка.

В табл. 4.1 представлен перечень доступных для списков методов, вызов осуществляется с помощью конструкции `список.метод()`.

Таблица 4.1

Методы обработки строк

Название метода	Описание
1	2
append(элемент)	Добавляет элемент в конец текущего списка. Если элемент – список, то он появится в а как вложенный
extend(список)	Добавляет в конец текущего списка все элементы из списка-аргумента
insert(индекс, элемент)	Вставляет элемент на заданную индексом позицию
remove(элемент)	Удаляет в текущем списке первый элемент, значение которого аргументу
clear()	Удаляет все элементы из списка а и делает его пустым
index(элемент)	Возвращает индекс элемента списка

Продолжение табл. 4.1

1	2
pop(индекс)	Удаляет элемент с заданным индексом и возвращает его
count(элемент)	Считает, сколько раз элемент повторяется в списке
reverse()	Возвращает обратный итератор списка текущего списка
sort()	Сортирует список. Для сортировки элементов в обратном порядке нужно установить дополнительный аргумент *reverse=True*
copy()	Создает поверхностную копию списка. Для создания глубокой копии используйте метод *deepcopy* из модуля *copy*

Помимо собственных методов к спискам применим и ряд встроенных функций: *min*() – возвращает элемент с минимальным значением, при этом элементы должны быть одного типа с возможностью сравнения; *max*() – возвращает элемент с максимальным значением, при этом элементы должны быть одного типа с возможностью сравнения; s*um*() – возвращает сумму элементов списка, при этом элементы должны быть числами; len() – возвращает количество элементов в списке и т.д.

4.1.4 Списки списков

Список может содержать объекты разных типов: числовые, буквенные, а также списки. Подобные списки можно соотнести с таблицами, где вложенные списки выполняют роль строк.

Например:

```
students = [
    ["Иван", 21],
    ["Арина", 22],
    ["Ангелина", 22]
]
print(students[0])          # ['Иван', 21]
print(students[0][0])       # Иван
print(students[0][1])       # 21
```

Для обращения к элементу вложенного списка необходимо использовать пару индексов: *students*[0][1] – обращение ко второму элементу первого вложенного списка.

Добавление, удаление и изменение общего списка, а также вложенных списков аналогично тому, как это делается с обычными (одномерными) списками

4.2 Кортежи

Кортежи (*tuple*), как и списки, предназначены для хранения последовательностей, состоящих из данных любого типа. Однако, в отличие от списков, относятся к неизменяемым типам данных. По этой причине в них часто хранят информацию, которую необходимо защитить от случайного изменения – например, конфигурационные данные.

Кортеж может быть создан перечислением элементов, конструктором *tuple*() и генератором:

```
tuple1 = ('Петров В.И.', 'студент', '090304-РПИа-о22')
lst = [15, "Автор", None, 34.5, True]
tuple2 = tuple(lst)
tuple3 = tuple('Python')
num = 123456
tuple4 = tuple(str(num))
tuple5 = tuple(i ** 2 for i in range(1, 12, 2))
```

Кортеж *tuple*1 создан перечислением элементов. Элементы структуры данных необязательно перечислять в круглых скобках, т.к. создание (*упаковка*) кортежа происходит автоматически в тех случая, когда *Python* получает более одного значения для переменной:

```
cities = 'Ростов-на-Дону', 'Новочеркасск'
print(cities) # ('Ростов-на-Дону', 'Новочеркасск')
```

Кортежи *tuple*2, *tuple*3, *tuple*4 и *tuple*5 созданы с использованием конструктора; *tuple*2 получается в качестве аргумента список, *tuple*3 – итерируемый объект, элементы которого и образуют структуру данных, аргумент для *tuple*4 явным образом преобразуется из числа в итерируемый объект – строку (напрямую числа использовать нельзя), *tuple*5 использует в конструкторе генератор.

Если генераторы кортежей используются без конструктора, нужно выполнить их распаковку:

```
my_tuple = (i for i in range(5))
```

Вызов `print(my_tuple)` завершится выводом следующей информации: `<generator object <genexpr> at 0x7f4a88c282e0>`, результатом `print(*my_tuple)` будут числа от 0 до 4 (включительно).

Индексация и срезы в рассматриваемой структуре данных работают так же, как и в списках:

```
letters = ('P', 'y', 't', 'h', 'o', 'n')
print(letters[1])       # y
print(letters[2: 4])    # ('t', 'h')
```

Кортежи поддерживают большинство методов списков, за исключением удаления элементов и присваивания им новых значений.

В случае однородных кортежи можно сравнивать между собой с помощью операторов >, >=, <, <=, ==, !=. Если элементы принадлежат к разным типам данных, поддерживаются только операторы == и !=.

4.3 Словари

Словарь (*dict*) в *Python* – структура данных, в которой информация хранится в виде хеш-таблицы (ассоциативного массива). В таком массиве каждый ключ сопоставлен с определенным значением. Значения могут повторяться и быть любого типа. Ключи должны быть уникальными и неизменяемых типов.

Словари оптимизированы для извлечения данных. В других языках подобную структуру обычно называются *map, hash* или *object*.

4.3.1 Создание словаря

Словари с данными можно создавать несколькими способами: перечислением ключей и значений (литералы словаря), конструктором *dict*(), комбинацией *dict*() и *zip*(), списком кортежей и кортежем списков, методом *fromkeys*() и генератором.

Первые два способа можно использовать только для создания небольших словарей, перечисляя все их элементы. Кроме того, во втором способе ключи передаются как именованные аргументы конструктора *dict*(), поэтому в этом случае ключи могут быть только строками, причем являющимися корректными идентификаторами:

```
lang_dict = {1: 'Python', 2: 'C#', 3: 'C++'}
empl_dict = dict(name = 'Евгений', age = 28, position
= 'Java-разработчик')
```

Пустой словарь можно создать при помощи конструктора *dict*() или пустой пары фигурных скобок {}.

Функция *zip*() используется для совмещения двух и более списков в один. Она возвращает итератор кортежей, в котором *i*-ый кортеж содержит *i*-ый элемент из каждого из переданных списков. Создание словаря с помощью совместного использования *zip*() и *dict*() предполагает наличие, как минимум, двух списков одинаковой длины: один содержит ключи, другой – значения:

```
numbers = [1, 2, 3, 4]
languages = ['Python', 'C#', 'C++', 'Java']
new_dict = dict(zip(numbers, languages))
```

Список кортежей, в которых первый элемент служит ключом, а второй – значением, можно преобразовать в словарь с помощью *dict*():

```
item = [('имя', 'Евгений'), ('возраст', '28'),
('должность', 'Java-разработчик')]
item_dict = dict(item)
```

Точно так же этот подход работает с кортежем списков:

```
item = (['имя', 'Евгений'], ['возраст', '28'],
['должность', 'Java-разработчик'])
item_dict = dict(item)
```

Метод f*romkeys*() позволяет создать словарь, у всех ключей которого *одинаковые* значения:

```
new_dict = dict.fromkeys(['Python', 'C#', 'C++'], 0)
```

Устанавливаемое значение передается в качестве аргумента.

В языке программирования *Python* наряду со списковыми выражениями (*генераторы списков*) существуют словарные выражения. Они заключаются в фигурные скобки, причем в подвы-

ражении до *for* должны стоять два объекта, разделенных двоеточием:

```
new_dict = {x: x*10 for x in range(4)}
```

В приведенной строке кода номера ключей определяются функцией *range*(), возвращающей диапазон, а значения определяются произведением номера ключа и числа.

4.3.2 Работа с элементами словаря

В отличие от других типов данных, где для доступа к элементам используется индексация, в словаре используются ключи. Они используются внутри квадратных скобок или в качестве аргумента метода *get*():

```
empl_dict = dict(name = 'Евгений', age = 28, position
= 'Java-разработчик')
print(empl_dict['name'])              # Евгений
print(empl_dict.get('age'))           # 28
print(empl_dict.get('address'))       # None
print(empl_dict['address'])           # KeyError
```

При использовании метода *get*() и отсутствии элемента возвращается *None*. При использовании квадратных скобок и отсутствии элемента в словаре вызывается ошибка *KeyError*.

4.3.3 Методы для работы со словарями

В табл. 4.2 представлен перечень доступных для словарей методов, вызов осуществляется с помощью конструкции `словарь.метод()`.

Таблица 4.2

Методы работы со словарями

Название метода	Описание
1	2
clear()	Удаляет все элементы из словаря
copy()	Возвращает неглубокую копию словаря
fromkeys(*seq*[, *value*])	Возвращает словарь с ключами из итерируемой последовательности *seq* и значениями, равными *value* (по умолчанию *None*)

Продолжение табл. 4.2

get(ключ[,*value*])	Возвращает значение ключа. Если ключ не существует, метод возвращает *value* (по умолчанию *None*)
items()	Возвращает новый объект элементов словаря в формате (ключ, значение)
keys()	Возвращает новый объект с ключами словаря
pop(ключ[,*value*])	Удаляет элемент с заданным ключом и возвращает его значение или *value*, если ключ не найден. Если *value* не было обозначено и ключ не найден, метод вызывает ошибку *KeyError*
popitem()	Удаляет и возвращает произвольную пару (ключ, значение). Вызывает ошибку *KeyError*, если словарь пустой
setdefault(ключ[,*value*])	Если ключ есть в словаре, возвращает соответствующее ему значение. Если ключ отсутствует, в словарь добавляется элемент с указанными ключом и значением, после чего возвращается значение (по умолчанию *None*)
update([*other*])	Обновляет словарь имеющимися парами ключ/значение из *other*, перезаписывая существующие ключи
values()	Возвращает новый объект со значениями словаря

Пример 1. Написать программу, которая получает на вход строку и подсчитывает, сколько раз в ней встречается каждый символ (независимо от регистра). Результат вывести без фигурных скобок:

```
my_str = input('Введите строку -> ').lower()
my_dict = {i: my_str.count(i) for i in my_str}
print(*[str(k) + '-' + str(v)
    for k, v in my_dict.items()])
```

Пример 2. Написать программу, которая получает на вход целое число от 1 до 12, и выводит название месяца, порядковому номеру которого соответствует введенное число. Если пользователь ввел число, которое меньше или больше допустимого – вывести подсказку:

```
months = {1: 'Январь', 2: 'Февраль', 3: 'Март',
          4: 'Апрель', 5: 'Май', 6: 'Июнь',
          7: 'Июль', 8: 'Август', 9: 'Сентябрь',
          10: 'Октябрь', 11: 'Ноябрь', 12: 'Декабрь'}
while True:
    num = int(input('Введите число -> '))
    if 1<=num<=12:
        print(months.get(num))
        break
    else: print('Введите число от 1 до 12')
```

Помимо собственных методов к словарям применим и ряд встроенных функций, например, *len*() – находит длину словаря; *type*() – возвращает сведения о типе данных; *min*() – позволяет получить ключ с минимальным значением; *max*() – находит ключ с максимальным значением.

4.3.4 Вложенные словари

В качестве значения ключа словаря может выступать любой объект: число, строка, список или словарь. Вложенные коллекции часто используются для структурирования данных:

```
students = {
    'Петров В.И.': {
        'phone': '+79281001010',
        'group': '090304-РПИа-о22'
    },
    'Васильева А.Д.': {
        'phone': '+79282002020',
        'group': '090301-ПОВа-о22',
        'praepostor': True
    }
}
```

Для обращения к элементам вложенного словаря соответственно необходимо использовать два ключа:

```
old_group = students['Петров В.И.']['group']
students['Петров В.И.']['group'] = '090301-ПОВа-о22'
print(students['Петров В.И.'])
```

Результат:

```
{'phone': '+79281001010', 'group': '090301-ПОВа-о22'}
```

При попытке получить значение по ключу, который отсутствует в словаре, *Python* сгенерирует исключение *KeyError*:

```
praep_stud = students['Петров В.И.']['praepostor']
```

Для избежания ошибок можно проверять наличие ключа в словаре:

```
key = 'praepostor'
if key in students['Петров В.И.']:
    print(students['Петров В.И.']['praepostor'] )
else:
    print("Данные не найдены")
```

В качестве альтернативы предотвращения возникновения исключения можно использовать метод *get*():

```
praep_stud = students['Петров В.И.'].get('praepostor',
    'Данные не найдены')
print(praep_stud)
```

Словари и генераторы словарей помогают в решении задач, связанных с подсчетом, множественным выбором, хранением и обработкой значений, описывающих свойства объектов. Во многих случаях словари позволяют избежать использования многоуровневых условий *if – elif – else*.

4.4 Множества

Множество (*set*) – неупорядоченная коллекция уникальных элементов неизменяемых типов, которая часто используется для проверки вхождения элементов, удаление дубликатов из последовательностей, а также для выполнения математических операций пересечения, объединения, разности и т.д.

Для объявления множества используются фигурные скобки, в которых через запятую перечисляются значения элементов, или конструктор *set*():

```
a = {5, 2, 3, 1, 4}
languages = ['Python', 'C#', 'C++', 'Java']
new_set = set(languages)
```

Если аргумент конструктора является итерируемым объектом, множество будет состоять из его уникальных элементов. Также допускается использовать функцию *range*() в качестве аргумента *set*():

```
set1 = set("abrakadabra")
set2 = set(range(7))
print(set1) #   {'k', 'r', 'd', 'a', 'b'}
print(set2) #   {0, 1, 2, 3, 4, 5, 6}
```

Элементы во множествах хранятся в неупорядоченном виде, к ним нельзя обратиться по индексу, следовательно, механизм срезов также не работает.

В табл. 4.3 представлен перечень доступных для множеств методов.

Таблица 4.3

Методы обработки строк

Название метода	Альтернативная запись	Описание
1	2	3
A.union(*B*)	*A* | *B*	Возвращает множество, являющееся объединением множеств *A* и *B*
A.update(*B*)	*A* |= *B*	Добавляет во множество *A* все элементы из множества *B*
A.intersection(*B*)	*A* & *B*	Возвращает множество, являющееся пересечением множеств *A* и *B*
A.intersection_update(*B*)	*A* &= *B*	Оставляет в множестве *A* только те элементы, которые есть в множестве *B*
A.difference(*B*)	*A* - *B*	Возвращает разность множеств *A* и *B* (элементы, входящие в *A*, но не входящие в *B*)
A.difference_update(*B*)	*A* -= *B*	Удаляет из множества *A* все элементы, входящие в *B*
A.symmetric_difference(*B*)	*A* ^ *B*	Возвращает симметрическую разность множеств *A* и *B* (элементы, входящие в *A* или в *B*, но не в оба из них одновременно)

Продолжение табл. 4.3

A.symmetric_difference_update(*B*)	*A* ^= *B*	Записывает в *A* симметрическую разность множеств *A* и *B*
A.issubset(*B*)	*A* <= *B*	Возвращает *True*, если *A* является подмножеством *B*
A.issuperset(*B*)	*A* >= *B*	Возвращает *True*, если *B* является подмножеством *A*

Пример 1. Написать программу, которая получает на вход две строки с перечислением интересов и хобби двух пользователей, и вычисляет процент совпадения:

```
hobbies1 = input('Первый пользователь -> ').split()
hobbies2 = input('Второй пользователь -> ').split()
hobbies1 = set(hobbies1)
hobbies2 = set(hobbies2)
set_inter = hobbies1.intersection(hobbies2)
set_un = hobbies1.union(hobbies2)
result = len(set_inter) / len(set_un) * 100
print(f'Совпадение интересов: {result:.2f}%')
```

Переменные *set_inter* и *set_un* хранят множества общих и всех интересов и хобби, соответственно.

В *Python* два вида множеств – обычное, изменяемое *set* и замороженное, неизменяемое *frozenset*. Замороженные множества нельзя изменить после создания, но все операции по объединению, пересечению и разности они поддерживают: результатом этих операций тоже будут *frozenset*. Неизменяемое множество *frozenset* может входить в изменяемое множество *set*.