Коллекции

Цель занятия

После освоения темы вы:

- узнаете понятие коллекции, поймете назначение коллекций в разработке;
- сможете использовать встроенные в Python коллекции для написания программ;
- сможете сопоставлять и выбирать необходимую структуру данных для конкретной практической задачи.

План занятия

1. Встроенные структуры данных Python.
2. Множества. Операции с множествами.
3. Строки. Индексация строк.
4. Списки.
5. Методы `split()` и `join()`. Списочные выражения.
6. Кортежи.
7. Словари.

Используемые термины

Структура данных — составной тип данных, является контейнером для других объектов.

Множество — неупорядоченная коллекция уникальных и неизменяемых объектов.

Список — упорядоченная изменяемая коллекция объектов произвольных типов.

Кортеж — неизменяемый тип данных, по сути представляет неизменяемый список.

Словарь — изменяемый неупорядоченный тип данных, хранящий пары «ключ: значение».

Конспект занятия

1. Встроенные структуры данных Python

Структура данных — составной тип данных, является контейнером для других объектов. Структура данных позволяет сгруппировать по одному имени несколько объектов одного или разных типов и рассматривать их как единое целое.

Python предоставляет встроенные типы данных для хранения коллекций из объектов. Встроенная структура данных являются частью языка, не нужно импортировать библиотеки или самому ее описывать

На рисунке 1 показаны встроенные в Python структуры данных.

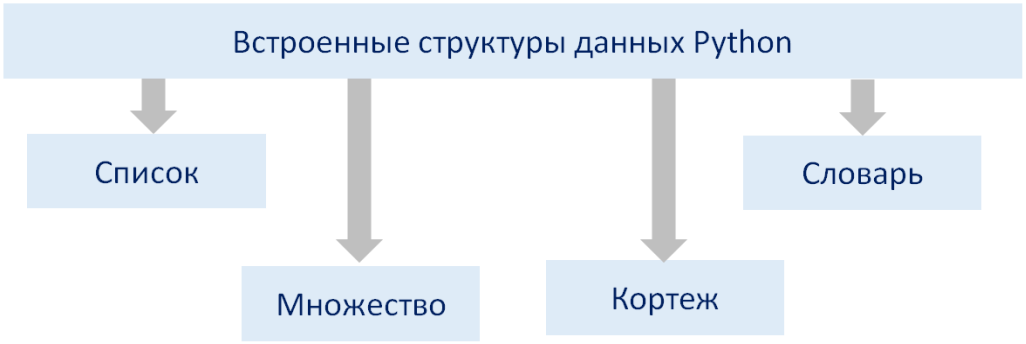

Рисунок 1. Встроенные структуры данных Python.

Строки тоже можно рассматривать как упорядоченную последовательность символов. То есть строка является своего рода контейнером для символов: букв того или иного языка, цифр, спецсимволов.

Важно! В Python не существует прямого понятия символа. Даже если строка состоит из одного символа, такая строка будет считаться строкой, состоящей из одного символа.

2. Множества. Операции с множествами

Множество — неупорядоченная коллекция уникальных и неизменяемых объектов.
В отличии от других языков программирования множества в Python могут содержать в себе объекты различных типов.

Пример создания множества и вывода его на экран:

```
mammals= {'cat','dog','fox','elephant'}
```

```
print(mammals)
```

Множество является неупорядоченной структурой данных, поэтому при выводе множества с помощью команды `print()`, элементы в множестве будут расположены в случайном порядке, например:

```
{'fox', 'elephant', 'cat', 'dog'}
{'elephant', 'fox', 'cat', 'dog'}
{'fox', 'cat', 'dog', 'elephant'}
```

Для создания пустого множества используется функция `set()`:

```
empty= set()
print(empty)
```

Пустое множество будет выведено как: `set()`

Множество может состоять из разных типов. Например, из строк и целых чисел:

```
m_nums= {'cat',5,'dog',3, 'fox', 12, 'elephant',4}
print(m_nums)
```

Элементы множества являются уникальными. Если будет создано множество с двумя одинаковыми элементами, то второй не уникальный элемент уже не будет входить в множество. Например, при выводе множества из пяти элементов, два из которых являются одинаковыми

```
birds = {'raven', 'sparrow', 'sparrow', 'dove', 'hawk'}
print(birds)
```

будет отображено множество из четырех элементов

```
{'sparrow', 'hawk', 'raven', 'dove'}
```

Вычисление количества элементов множества. Для вычисление количества элементов множества используется команда `len()`:

```
my_set= {'a', 'b', 'c', 1, 2, 3}
n = len(my_set)
print(n)
```

После выполнения кода выведется число 6.

Вывод элементов множества на экран. Как и все коллекции, множество может работать с циклом `for`. В примере ниже вместо функции `range()` используется название множества, а после каждой итерации переменная `element` становится равной следующему элементу множества. Количество итерация соответствует количеству элементов в множестве.

```
computer = {'Системный блок', 'Монитор', 'Клавиатура', 'Мышь'}
for element in computer:
    print(element, end = ',')
```

После выполнения примера через запятую будут выведены все элементы множества:

```
Клавиатура, Монитор, Системный блок, Мышь,
```

Проверка наличия элемента в множестве. Данная операция выполняется с помощью команды `in`. Запись элемента должна строго соответствовать записи элемента в множестве. После выполнения примера, если элемент присутствует в множестве будет выведено `True`, иначе — `False`.

```
computer = {'Системный блок', 'Монитор', 'Клавиатура', 'Мышь'}
print('Мышь' in computer)
print('Блок' in computer)
```

После запуска кода будет построчно выведено

```
True
False
```

Программа выведет `False`, не смотря на то, что слово «блок» присутствует в одном из элементов множества.

Добавление элемента в множество. Чтобы добавить элемент в множество используется метод `add()`. В скобках для метода указывается значение добавляемого элемента. В примере в множество вносятся все числа от 0 до 6, а затем выводятся на экран.

```
numbers = set()
for i in range(7):
    numbers.add(i)
print(numbers)
```

Удаление элемента из множества. Чтобы удалить элемент из множества используется одна из трех команд:

- `remove()` — удаляет существующий элемент в множестве, если элемента не существует, программа останавливается с ошибкой;
- `discard()` — удаляет существующий элемент в множестве, если элемента не существует, программа продолжает работать;
- `pop()` — удаляет существующий элемент в множестве и возвращает значение удаленного элемента.

```
numbers = {7, 2, 1, 4, 3, 6, 5}
numbers.remove(1)
numbers.discard(8)
for i in range(len(numbers)):
    print(numbers.pop(), end = ', ')
```

После выполнения примера будет выведено:

```
2, 3, 4, 5, 6, 7,
```

Операции над двумя множествами наглядно представляются с помощью кругов Эйлера (рисунок 2).

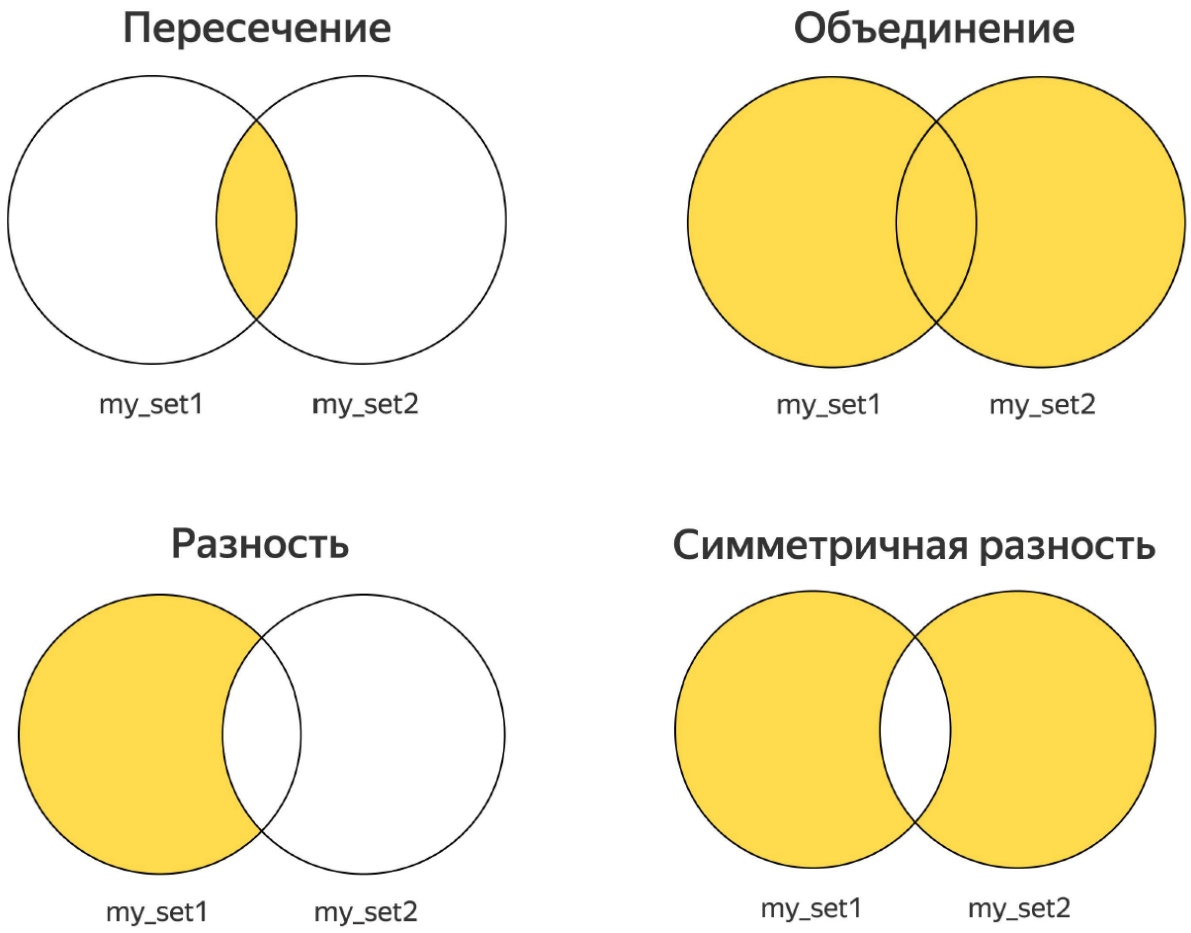

Рисунок 2. Представление операций над двумя множествами.

При **пересечении** (&) двух множеств возвращаются все элементы, состоящие и в первом, и во втором множестве.

Объединение (|) двух множеств состоит в добавлении к элементам первого множества неповторяющихся элементов второго множества.

Примеры реализации пересечения и объединения множеств в Python:

```
firms = {'Apple', 'Acer','Blackberry', 'Samsung'}
fruits = {'Apple','Mandarin', 'Pear','Blackberry'}
print(len(firms & fruits))
print(len(firms | fruits))
```

После выполнения программа возвращает количество элементов, полученных в результате выполнения операций: 2 и 6. Пересечением множества будут два элемента — `' Apple'` и `'Blackberry'`, объединением будут шесть элементов — `'Apple', 'Acer','Blackberry', 'Samsung', 'Mandarin', 'Pear'`.

Разность множеств (-) состоит в том, что из первого множества убирают элементы, состоящие во втором множестве.

В результате **симметричной разности** (^) остаются все элементы, кроме общих элементов исходных множеств.

Примеры реализации разности и симметричной разности множеств в Python:

```
fib_numbers= {1, 2, 3, 5, 8, 13}
odd_numbers= {1, 3, 5, 7, 9, 11, 13}
print(len(fib_numbers-odd_numbers))
print(len(odd_numbers^fib_numbers))
```

После выполнения программа возвращает количество элементов, полученных в результате выполнения операций: 2 и 5.

Множества допускают выполнение операции сравнения. Два множества будут равными, если количество элементов в них и сами элементы совпадают между собой. Порядок элементов во множестве не имеет значения.

Примеры сравнения множеств:

`{1, 2, 3, 4, 5, 6} == {6, 5, 3, 4, 1, 2}` вернет `True`

`{2, 4, 6, 8, 10}!= {10, 2, 6, 4, 8}` вернет `False`

В сравнении множеств могут присутствовать знаки <, <=, >, >=. Знак «<=» означает, что левое множество должно быть подмножеством правого, либо совпадать с ним. Знак «>» означает, что правое множество должно быть строго подмножеством левого множества.

```
{'one', 'three'} <= {'one', 'two', 'three'}
```
вернет `True`

```
{1, 3, 5, 7, 9, 11} > {3, 5, 7, 11, 13}
```
вернет `False`

```
{'let', 'it', 'be'} < {'be', 'it', 'let'}
```
вернет `False`

3. Строки. Индексация строк

Строки в Python можно определить несколькими способами.

Первый способ — присвоить переменной значение, заключенное в одинарные или двойные кавычки.

```
fixed_word = 'опять'
print(fixed_word)
```

Второй способ — с помощью функции `input()`, введя значение в консоль.

```
word = input()
print(word)
```

Третий способ — преобразовать переменную любого другого типа в строку функцией `str()`.

```
number = 25
number_string = str(number)
print(number_string)
```

Четвертый способ — из двух других строк с помощью конкатенации.

```
word_plus_number = fixed_word + number_string
print(word_plus_number)
```

Нумерация символов в строке начинается с 0, каждому символу соответствует свой номер. В Python доступна и обратная индексация.

0	1	2	3	4	5	6	7	8	9	10	11
П	р	и	в	е	т	,		м	и	р	!
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Рисунок 3. Прямая и обратная индексация строки.

Индексация строки позволяет посимвольно обратиться к элементам строки, например, `print(word[5])`. Если элемента с запрашиваемым индексом не существует, выведется сообщение об ошибке.

Обратная индексация удобна, когда необходимо работать с концом строки: `print(word[-1])` — вернет последний символ строки.

Важно! Изменение символа строки по его индексу является недопустимой операцией, выполнение такой программой завершится ошибкой типа данных.

Перебор элементов строки. При работе со строками можно использовать циклы, например, `for`. В примере рассматривается задача нахождения гласных элементов в строке с помощью множества.

```
text = 'Hello, my dear friends!'
vowels = 0
for letter in text:
    if letter in {'a', 'e', 'i', 'o', 'u', 'y'}:
        vowels += 1
print(vowels)
```

Эту же задачу можно решить с помощью посимвольного обращения к строке, состоящей из гласных букв.

```
text = 'Hello, my dear friends!'
vowels = 0
for i in range(len(text)):
    if text[i] in 'aeiouy':
        vowels += 1
print(vowels)
```

Каждый символ в памяти компьютера имеет свой код, узнать который можно с помощью функцию `ord()`. Например, `print(ord('Б'))` вернет значение 1041.

Обратная функция — `chr()` — возвращает символ по его коду. Например, `print(chr(1044))` вернет символ «Д».

Функции `ord()` и `chr()` могут помочь при работе с конкретными символами.

Пример программы для вывода символов английского алфавита.

```
for i in range(26):
    print(chr(ord('A') + i),  end=',')
```

4. Списки

Список связан с другими ранее рассмотренными структурами данных (рисунок 4).

Рисунок 4. Связь списка с другими структурами данных.

Для списков выполняются условия:

- элементы могут быть разных типов,
- элементы упорядочены и проиндексированы,
- элементы можно перебирать, используя цикл `for`.

Для создания пустого списка можно использовать либо пустые квадратные скобки `[]`, либо команду `list()`.

```
empty1= []
empty2 = list()
print('Пустые списки: ', empty1, empty2)
```

Если необходимо создать список со значениями по умолчанию, например, содержащий 5 нулей, можно воспользоваться кодом

```
my_list= [0] * 5
```

У элементов списка есть нумерация или, как иногда называют индексация. Нумерация начинается с 0 (если идём слева направо). Можно обращаться как ко всему списку, так и к отдельным элементам списка.

```
          0  1  2  3  4
primes = [2, 3, 5, 7, 11]
```

Рисунок 5. Индексация списка.

```
print('Весь список:',primes)
print('Сумма первых двух простых чисел:',primes[0] + primes[1])
```

Используя обратную индексацию, можно обратиться к последнему элементу списка.

```
print('Последнее из простых чисел в нашем списке:',primes[-1])
```

Чтобы добавить очередной элемент в конец списка используется метод `append()`.

```
primes.append(13)
```

В отличие от строк, списки являются изменяемым типом данных. Поэтому при ошибочном присваивании значения списку, это значение можно изменить.

```
primes= [2, 3, 5, 7, 11, 13]
primes.append(15)     # ошиблись
primes[6] = 17        # ошибка исправлена
print('Дополненный список:',primes)
```

Пример операции сложения двух списков аналогичен операции конкатенации двух строк.

```
primes += [19, 23, 29]
print('Расширенный список:',primes)
```

Количество элементов в списке позволяет определить функция `len()`

```
print('Количество элементов в списке:',len(primes))
```

Команда `in` определяет вхождение того или иного значения в список.

```
print(1 in primes)
```

Для перебора элементов списка существуют два способа: поэлементный и по индексу. В обоих случаях используется цикл `for`.

Пример поэлементного перебора списка и вывода в консоль. Список `primes` перебирается от нулевого элемента до последнего по порядку через переменную `prime`.

```
for prime in primes:
   print(prime)
```

Пример перебора элементов списка по индексу. Значение переменной `i` на каждой итерации равно очередному номеру элемента в списке.

```
for i in range(len(primes)):
    print(i, ':',primes[i])
```

Пример программы «Список покупок». Количество покупок вводится с консоли в переменную `n`. Затем определяется пустой список `purchase_list`, в цикле вводим название очередной покупки и используя метод `append()` добавляем покупку в список. Для вывода значений из списка используется еще один цикл `for`.

```
n = int(input())
purchase_list= []
for i in range(n):
    purchase = input()
    purchase_list.append(purchase)
for i in range(n):
    print(purchase_list[i])
```

Помимо обращения к элементам списка по одному, в списках используются срезы.
В примере ниже рассматривается выделение всех весенних месяцев. В переменную `spring` запишется уже другой список — часть списка `months`.
В квадратных скобках записи `months[2:5]` указываются два значения, обозначающие, что необходимо оставить элементы с индексами со 2 по 5, 5 — не включается. То есть в срез будут входить элементы с индексами 2, 3 и 4.

```
months = ['январь', 'февраль', 'март', 'апрель', 'май', 'июнь',
'июль', 'август', 'сентябрь', 'октябрь', 'ноябрь', 'декабрь']
spring = months[2:5]
for month in spring:
   print(month, end=' ')
```

Чтобы собрать в один список зимние месяца, нужно к последнему месяцу присоединить первые два. Для этого можно использовать обратную индексацию. Если второй элемент в срезе не указан, срез берется до конца списка. Если не указано первое значение в срезе, то срез берется от начала списка до указанного индекса не включительно.

```
winter = months[-1:] + months[:2]
for month in winter:
    print(month, end=' ')
```

В следующем примере в срезе задаются два двоеточия. Срез в этом случае будет выполняться по следующему принципу: начиная от первого числа до второго не включительно с шагом заданным третьим числом. То есть в примере выполняется срез списка от начала и до конца с шагом 2, будут выведены все элементы, стоящие на четных местах.

```
months = ['январь', 'февраль', 'март', 'апрель', 'май', 'июнь',
'июль', 'август', 'сентябрь', 'октябрь', 'ноябрь', 'декабрь']
print(months[::2])
```

Аналогичным образом можно получить элементы, стоящие на нечетных местах.

```
print(months[1::2])
```

Следующий срез выводит элементы с 2 по 9 не включительно с шагом 3.

```
print(months[2:9:3])
```

Срез можно выполнять и в обратном порядке. Но для корректной работы необходимо, чтобы первое число было больше второго. Такой прием используется для того, чтобы развернуть список.

```
print(months[6:1:-1])
```

Важно! Все рассмотренные примеры срезов могут использоваться и со строковым типом данных.

5. Методы `split()` и `join()`. Списочные выражения

Метод `split()` разбивает строку по произвольному разделителю на список «слов». Это может быть удобно, если необходимо разделить предложения на отдельные слова. Есть два способа вызова метода:

- вызывается без аргументов — разбивает по символам пустого пространства,
- вызывается с аргументом-строкой — разбивает по этой строке.

Пример разделения строки на слова, в качестве разделителя выступает символ пробел.

```
s='раз два три'
print(s.split())
```

Также метод `split()` может быть применен к конкретной строке.

```
print(' one two three '.split())
```

Разделителем помимо пробела может выступать другой символ или строка.

```
print('192.168.1.1'.split('.'))
```

Пример разделение строки по символу «а», пробелы останутся в результате. Будет получено следующее: `['р', 'з дв', ' три']`.

```
s = 'раз два три'
print(s.split('а'))
```

В некоторых файлах данных разделителем может выступать решетки «##».

```
print('A##B##C'.split('##'))
```

Метод `join()` собирает из списка слов единую строку через произвольный разделитель (точнее, соединитель). Метод всегда принимает один аргумент — список «слов», которые нужно «склеить» (конкатенировать), соединитель — та строка, для которой вызван метод. Соединителем может быть пустая строка, пробел или другие символы. В таблице 1 показаны примеры использования метода. В качестве исходного списка выступает список `s = ['Тот', 'Кого', 'Нельзя', 'Называть']`.

Таблица 1. Примеры использования метода `join()`.

Пример использования	Результат вывода в консоль
`print(''.join(s))`	ТотКогоНельзяНазывать
`print(' '.join(s))`	Тот Кого Нельзя Называть
`print('-'.join(s))`	Тот-Кого-Нельзя-Называть
`print('! '.join(s))`	Тот! Кого! Нельзя! Называть

На рисунке 6 схематично показано, как работают методы `split()` и `join()`:

- `split()` служит для преобразования строки в список,
- `join()` служит для преобразования списка в строку.

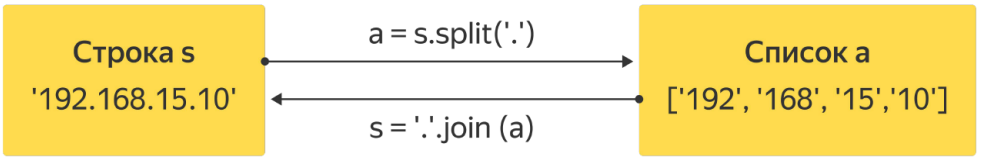

Рисунок 6. Пояснение к работе методов `split()` и `join()`.

Важно! Метод `split()` применяется только к строкам, метод `join()` — только к спискам. Использование этих методов с другими типами коллекций приведет к ошибкам в программе.

Списочные выражения (list comprehensions) позволяют создавать списки в цикле `for`, не записывая цикл целиком

В примере формируется список из квадратов чисел от 0 до 7. Для этого используется цикл `for`, внутри которого содержится метод `append()`, добавляющий очередной элемент в список.

```
squares = []
for i in range(8):
    squares.append(i**2)
print(squares)
```

Рассмотренный пример можно реализовать с помощью списочного выражения. Для этого внутри списка прописывается цикл `for`, то преобразование, которое необходимо сделать с переменной `i` указывается слева от цикла `for`.

```
squares=[i**2  for i in range(8)]
print(squares)
```

Списочные выражения могут работать и с условиями. Например, требуется создать список квадратов четных чисел от 0 до 9. Реализация без использования списочных выражений.

```
even_squares= []
for i in range(10):
    if i % 2 ==  0:
        even_squares.append(i**2)
print(even_squares)
```

Тот же пример, но уже с применением списочного выражения. В пример было добавлено условие справа от цикла `for`.

```
even_squares2 = [i**2 for i in range(10) if i % 2 == 0]
```

```
print(even_squares2)
```

Таким образом, при использовании списочного выражения нужно отталкиваться от цикла `for`: слева записывается действия, которое выполняется с переменной, справа — условие, если это необходимо.

Списочные выражения позволяют использовать несколько циклов, вложенных друг в друга.

```
print([i * j for i in range(3) for j in range(3)])
```

Списочные выражения удобно использовать при считывании чисел, записанных в строку. В примере с помощью метода `split()` получаем отдельные элементы строки, а затем с помощью функции `int()` каждое из них переводится в целочисленный тип.

```
a = [int(x) for x in '976 929 289 809 7677'.split()]
```

Также можно использовать множественное присваивание

```
evil, good= [int(x) for x in '666 777'.split()]
```

Со списочным выражением можно использовать метод `join()`. Цикл `for` перебирает все числа от 1 до 9 включительно, команда слева от цикла выполняет перевод числа в строку, сложение со строкой «`^2`» и строкой представляющей квадрат исходного числа. Метод `join()` объединяет все в строку.

```
print(', '.join(str(i) + '^2='+ str(i**2) for i in range(1, 10)))
```

Результат вывода:

```
1^2=1,  2^2=4,  3^2=9,  4^2=16,  5^2=25,  6^2=36,  7^2=49,  8^2=64,
9^2=81
```

6. Кортежи

Кортеж — неизменяемый тип данных, по сути представляет неизменяемый список. Кортеж во многом схож со списком, но его элементы изменять мы не можем.

Чтобы определить пустой кортеж, достаточно после названия переменной обозначить пустые скобки `()`. Если мы хотим определить кортеж с конкретными элементами, можно указать их явно: `card = ('7', 'пик')`. Если кортеж будет состоять из одного элемента, для корректной работы необходимо после значения элемента поставить знак «,»: `t = (18,)`.

Чтобы определить количество элементов в кортеже достаточно воспользоваться функций `len()`.

Кортежи допускают поэлементное обращение по индексу элемента. Например, чтобы обратиться к первому элементу кортежа `card` нужно записать `card[0]`.

Также допускается добавление одного кортежа к другому:

```
card = ('7', 'пик')
t = (18,)
print(card + t)
```

В результате программа вернет `('7', 'пик', 18)`.

Важно! Попытка поэлементного изменения значения кортежа, например, `card[1] = 'треф'` приведет к ошибке. Кортеж является неизменяемым типом данных.

Кортежи можно сравнивать между собой. Два кортежа равны, если количество элементов в них, значение и порядок элементов совпадают.

`(1, 2, 3) == (1, 3, 2)` вернет `False`

При использовании знака «<» сначала сравниваются два первых элемента, если они равны друг другу. Далее сравниваются следующие элементы. Знак «<=» работает по такому же принципу.

`(1, 2, 3) < (1, 2, 4)` вернет `True`

`(1, 2) <= (5,)` вернет `True`

При сравнении кортежей, состоящих из строк, используется принцип описанный выше. Отличие — учитывается алфавитный порядок.

`('7', 'треф') > ('7', 'червей')` вернет `False`

Условие со знаком «!=» будет выдавать `True` в том случае, когда хотя бы один из элементов не равен другому.

`(1, 2) != ('7', 'пик')` вернет `True`

`(3, 4) < ('5', 'бубен')` вернет `'<' not supported between instances of 'int' and 'str'`

Кортежи широко применяются при множественном присваивании. Слева записывают кортеж, состоящих из переменных, справа — кортеж из значений

```
(n, s) = (10, 'hello')
```

то же самое, что

```
n, s = 10, 'hello'
```

Присваивание может быть более сложным. В примере показан список, каждый элемент которого представляет кортеж. Слева записано значение переменных, справа — значение из списка.

```
cards = [('7', 'пик'), ('Д', 'треф'), ('Т', 'пик')]
value, suit = cards[0]
```

7. Словари

Рассмотрим пример списка, содержащего фамилии известных актеров.

```
actors=['Джонни Депп', 'Эмма Уотсон', 'Билли Пайпер']
```

Для обращения к элементам списка используются последовательности целых чисел: `print(actors[1])`.

Пусть стоит задача хранить не только имя актера, но и его биографию. Решением может быть реализация в виде списка кортежей.

```
actors=[
('Джонни Депп', 'Джон Кристофер Депп Второй родился 9 июня 1963 года в Овенсборо, Кентукки..'),
('Сильвестр Сталлоне', 'Майкл Сильвестр Гарденцио Сталлоне родился в Нью-Йорке. Его отец, парикмахер Фрэнк Сталлоне — иммигрант из Сицилии, ...'),
('Эмма Уотсон', 'Эмма Шарлотта Дуерр Уотсон родилась в семье английских адвокатов. В пять лет переехала вместе с семьей из Парижа в Англию...'),
# ...
]
```

Но такое решение неудобно: для поиска статьи об актере придётся пройтись по всему списку, например с помощью цикла `for`. В подобных задачах удобнее использовать тип данных — словарь.

Словарь — изменяемый неупорядоченный тип данных, хранящий пары «ключ: значение». Словарь хранит, как и список, много данных, но индексом («ключом») может быть строка.

Пример решения задачи хранения имени актера и его биографии с помощью словаря. Каждый элемент словаря состоит из двух пары значений, разделенных знаком двоеточия. Ключом является имя актера.

```
actors={
'Джонни Депп': 'Джон Кристофер Депп Второй родился 9 июня 1963 года в Овенсборо, Кентукки..',
'Сильвестр Сталлоне': 'Сильвестр Гарденцио Сталлоне родился в Нью-Йорке. Его отец, парикмахер Фрэнк Сталлоне —иммигрант из Сицилии, ...',
'Эмма Уотсон': 'Эмма Шарлотта Дуерр Уотсон родилась в семье английских адвокатов. В пять лет переехала вместе с семьей из Парижа в Англию...',
# ...
}
```

Чтобы определить пустой словарь, необходимо использовать функцию `dict()` или пустые фигурные скобки `{}`.

```
d = dict()
# или так
d = {}
```

Для обращения к элементу словаря нужно знать его ключ.

```
print(actors['Эмма Уотсон'])
```

Также можно обратиться к ключу использованием конкатенации

```
first_name ='Сильвестр'
last_name ='Сталлоне'
print(actors[first_name+' '+last_name])
```

Если ключа в словаре нет, возникнет ошибка

```
print(actors['Несуществующий ключ'])
KeyError: 'Несуществующий ключ'
```

Добавление записи или редактирование существующей записи. Указывается имя словаря, ключ и присваиваемое ключу значение. При изменении записи указывается уже существующий ключ.

```
actors['Брэд Питт'] = 'Уильям Брэдли Питт, более известный как Брэд Питт—американский актёр и продюсер. Лауреат премии «Золотой глобус» за 1995 год, ...';
```

Удаление записи. Для удаления записи используется команда `del` после которой через пробел записывается название словаря с соответствующим ключом.

```
del actors['Джонни Депп']
# больше в словаре нет ни ключа 'Джонни Депп', ни соответствующего ему значения
print(actors['Джонни Депп'])
KeyError: 'Джонни Депп'
```

Для удаления можно использовать функцию `pop()`.

```
actors.pop('Джонни Депп')
```

Функция `pop()` возвращает удаляемое значение, поэтому можно записать:

```
deleted_value=actors.pop('Джонни Депп')
```

Если в словаре нет ключа `'Джонни Депп'` возникнет ошибка `KeyError`.

Чтобы ошибки о несуществующем ключе не возникало, можно в функцию `pop()` передать второй аргумент. В случае отсутствия нужного ключа `pop()` вернёт этот аргумент.

```
deleted_value = actors.pop('Джонни Депп', None)
```

Если ключа `'Джонни Депп'` в словаре не оказалось, в переменной `deleted_value` окажется `None`, и ошибки не возникнет.

Для поиска по словарю используется `in`.

```
if 'Джонни Депп' in actors:
    print('У нас есть статья про Джонни Деппа')
```

Существует аналогичный оператор `not in`.

```
if 'Сергей Безруков' not in actors:
    print('У нас нет статьи о Сергее Безрукове')
```

Взять значение в словаре можно с помощью метода `get()`.

```
article = actors.get('Джонни Депп')
```

Метод `get()` может принимать второй аргумент — значение, которое вернется, если заданного ключа нет.

```
article = actors.get('Джонни Депп','Статья о Джонни Деппе не найдена')
```

Получить список всех ключей словаря можно с помощью метода `keys()`

```
actors_names= list(actors.keys())
```

Получить список всех значений словаря можно с помощью метода `values()`

```
all_articles= list(actors.values())
```

Получить пары ключ–значение можно с помощью метода `items()`. Пары ключ-значения используют при переборе элементов словаря с помощью цикла `for`.

```
for key, valind.items():
```

При каждой итерации цикла будут соответствующие ключ и значение.

Дополнительные материалы для самостоятельного изучения

1. Data Structures — Python 3.11.4 documentation