Инструментальные средства информационных систем

БИН-19-1

Лекция 4 Структуры данных и функции в Python

Содержание

Введение

Мы уже попробовали на практическом занятии использовать операторы и команды Python, поэтому рассмотрим работу со структурами данных Python.

Чтобы понять структуру данных придется упомянуть объекты и классы. Например, если мы присваиваем некоторой переменной **a** значение **5**:

```
>>> a = 5
>>> type(a)
<class 'int'>
```

, то мы создаём объект **a** класса **int**.

В объектно-ориентированном программировании класс объектов имеет свои методы, т.е. некоторые функции, которые предназначены для работы с объектами данного класса.

```
>>> a = 'Hell!o'
>>> print(a)
Hell!o
>>> print(a.replace('!o','o!'))
Hello!
>>> print(a)
Hell!o
```

Класс строка, метод класса замена символов replace().

1 Строки

Строки (string) – это последовательности символов, буквы, цифры, пробелы, переводы строки и многое другое.

Создать строку можно разными способами используя одинарные и двойные кавычки:

```
>>> a = 'String1'
>>> a = "String2"
>>> a = '''Some
... long text'''
>>> a = """Some long
... text 2"""
>>> print(a)
Some long
text 2
```

1.1 Операции над строками

Над строками можно производить некоторые операции, строки можно складывать (конкатенировать) и умножать.

```
>>> a = 'x'+'o'
>>> print(a)
xo
>>> a = a * 3
>>> print(a)
xoxoxo
```

Математические операции, деление, возведение в степень, целочисленное деление не работают.

1.2 Сравнение строк

Строки можно сравнивать между собой. При этом сравнение будет происходить лексикографически. Например:

```
>>> 'qwe' == "qwe"
True
>>> 'abc' < 'abd'
True
>>> 'abc' > 'abd'
False
>>> 'abc' < 'ab'
False
>>> 'abc' > 'ab'
True
```

Цифры меньше букв:

```
>>> 'abc' < 'ab1'
False
```

1.3 Разбираем строки на символы

Как было сказано выше, строка это последовательность символов, таким образом мы можем работать с отдельными символами строки.

Символы в строках имеют свой порядковый номер, индекс, первый символ имеет нулевой индекс.

Например, у нас есть строка:

```
a = 'Python'
```

То символ с нулевым индексом это ‘P’, индекс равный 1 имеет символ ‘y’ и так далее. Обращаться к символу можно по его индексу, например, мы можем получить второй символ по индексу следующим образом:

```
print(a[2])
```

К символам в строке можно обращаться отсчитывая индекс с конца строки, т.е. можно использовать отрицательную индексацию. Индекс -2 означает второй символ с конца.

Например, прямая и обратная индексация:

```
>>> a = 'Python'
>>> print(a[2])
t
>>> print(a[-2])
o
```

Вместо индекса заданного в виде константы, мы, конечно, можем использовать переменную.

```
>>> i = 3
>>> print(a[i])
h
```

Можно получить символ строки по индексу, но изменить символ строки по индексу нельзя. Такой код не работает:

```
>>> i = 3
>>> print(a[i])
h
>>> a[i] = 'H'

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Благодаря тому, что строки это набор символов, мы можем перебирать символы строки в цикле.

```
a = 'Python'
for i in range(6):
   print(a[i])
```

Получим:

```
P
y
t
h
o
n
```

Мы можем немного автоматизировать цикл, добавив функцию len():

```
a = 'Python'
for i in range(len(a)):
   print(a[i])
```

Но гораздо удобнее для строк применять другую конструкцию цикла:

```
a = 'Python'
for c in a:
   print(c)
```

В данном случае `c` принимает по порядку все значения, которые есть в строке.

1.4 Строки и их методы

Рассмотрим для начала задачу, давайте посчитаем сколько раз встречается символ "o" в какой-то строке.

Сначала мы попросим ввести строку. Потом заведем переменную для подсчета символов. Затем в цикли будем перебирать символы и, если текущий символ равен искомому, то будем увеличивать счетчик на единицу. Напишем следующую программу:

```
a = input('Введите строку: ')
n = 0
for c in a:
   if c == 'o':
       n+=1
print(n)
```

На самом деле получить то, что нам нужно в этой задаче можно гораздо короче, используя методы строки. Т.е. используя заранее написанные функции для работы со строками.

```
print(a.count('o'))
```

count – это метод, который считает количества вхождений.

Использование функций не отрицает умение писать код простыми командами. Знание функций и умение их применять, конечно, упрощает и ускоряет разработку, но также важно уметь писать свои собственные аналогичные алгоритмы. Потому что не всегда стандартное решение будет подходящим. И вторая причина, на основании простого можно построить что-то сложное.

Строки имеют следующие методы:

`s.upper()` – возвращает символы строки в верхнем регистре.

`s.lower()` – возвращает символы строки в нижнем регистре.

`s.count('c')` – подсчитывает сколько раз в строке встретился символ 'c'.

`s.find('c')` – возвращает первое вхождение символа 'c'.

`s.replace('c','r')` – возвращает строку с измененными символами 'c' на 'r'.

Пусть у нас будет задано две строки a и b:

```
a = 'Python'
b = 'th'
```

Тогда вышеперечисленные методы:

```
print(a.upper())
print(a.lower())
print(a.count(b))
print(a.find(b))
print(a.replace(b,'TH'))
```

Дадут нам следующий результат:

```
PYTHON
python
1
2
PyTHon
```

При использовании метода строки сама строка не меняется, чтобы сохранить результат изменения нам надо завести другую переменную и присваивать ей результат работы методов.

Методы можно применять один за другим, вызывая их к одной и той же строке. Методы в этом случае будут выполняться последовательно слева направо, выполняя действия в скобках выше по приоритету. Методы в последовательности вызываются через точку.

Рассмотрим пример:

```
a = 'Guido van RosSum'
b = 's'

print(a.upper().count(b.upper()))
```

Результат будет 2, т.е. два раза встречается символ **s** в строке **a**.

Сначала выполняется метод upper строки **a**, затем мы переходим к методу count, но у него в скобках вызывается другой метод к строке b – сначала выполняется он, а затем возвращаемся к count.

1.5 Вырезание частей строки

Рассмотрим различные вариации указывания индексов строки. Таким образом мы сможем вырезать нужные части строки, такой метод называется slicing.

Пример:

```
>>> a = 'GuidoVanRossum'
>>> len(a)
14
>>> a[1]
'u'
>>> a[1:5]
'uido'
>>> a[:5]
'Guido'
>>> a[5:]
'VanRossum'
>>> a[-5:]
'ossum'
>>> a[1:-1]
'uidoVanRossu'
>>> a[1:-1:2]
'udVnos'
>>> a[::-1]
'mussoRnaVodiuG'
```

Есть строка длиной 14 символов. Самое первое получаем один символ.

Второй пример – получаем с первого по пятый символ, пятый символ при этом не включается.

Третий – указана только конечная граница, в данном случае начальная граница считается равной 0.

Четвертый – указана только левая граница, соответственно правая граница считается конец строки.

Пятый – указана левая граница, но она отрицательная, значит первый символ в выборке будет пятый с конца.

Шестой – указаны обе границы, но правая граница отрицательная, значит правая граница это первый символ с конца и его мы по обыкновению не включаем.

Седьмой – кроме двух границ, указан шаг, т.е. берём каждый второй символ, и правую границу выкидываем.

Восьмой – не указаны границы, значит берём всю строку, но указан отрицательный шаг, значит берем строку начиная с конца.

2 Списки

Список (list) – это набор значений, элементами списка могут быть любые значения: строки, числа, другие списки и т.д., каждое значение в списке имеет свой порядковый номер (индекс).

Один из способов создания списков, присвоение значений списка:

```
>>> friends = ['Sasha','Dasha','Pasha','Masha']
```

Элементы списка задаются в квадратных скобках, так Python понимает, что перед ним список. Часто требуется создать пустой список, чтобы потом добавлять в него элементы:

```
>>> a = []
```

Другой вариант создания пустого списка, это использовать конструктор list():

```
>>> a = list()
```

2.1 Разбираем список на элементы

Элементы списка можно извлекать так же как символы из строк. Многие приемы работы со списками похожи на приемы работы со строками, но списки имеют больше возможностей, потому что список изменяемый тип данных в отличие от строк.

Получить элемент списка можно по его индексу:

```
>>> friends[0]
'Sasha'
>>> friends[1]
'Dasha'
>>> friends[2]
'Pasha'
>>> friends[3]
'Masha'
```

Допустима отрицательная индексация:

```
>>> friends[-1]
'Masha'
>>> friends[-2]
'Pasha'
>>> friends[-3]
'Dasha'
>>> friends[-4]
'Sasha'
```

Можно применять slicing:

```
>>> friends[1:3]
['Dasha', 'Pasha']
>>> friends[::-1]
['Masha', 'Pasha', 'Dasha', 'Sasha']
```

Можно обращаться к элементам списка в цикле, как по индексу, так и напрямую извлекая поочередно элементы из списка:

```
friends = ['Sasha', 'Dasha', 'Pasha', 'Masha']
for item in friends:
   print('Hello, ', item)
```

Получим:

```
Hello,  Sasha
Hello,  Dasha
Hello,  Pasha
Hello,  Masha
```

Можно проверять вхождение элемента в список в условиях:

```
if 'Pasha' in friends:
   print('Pasha in list.')

if 'Gosha' not in friends:
   print('Gosha is not in list')
```

И второй способ определения существует ли элемент в списке через метод index(), которая возвращает индекс элемента:

```
>>> friends.index('Pasha')
2
>>> friends.index('Gosha')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'Gosha' is not in list
```

2.2 Операции над списками

К спискам можно применять две основные операции – это сложение и умножение.

Можем прибавить один список к другому:

```
>>> a = [1 , 2, 3]
>>> friends+a
['Sasha', 'Dasha', 'Pasha', 'Masha', 1, 2, 3]
```

Можем умножить список, получим повторение списка:

```
>>> a * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

При этом сами списки не изменяются, чтобы сохранить результат нужно записать его в другой список.

2.3 Изменение списков

Как было сказано выше, списки это изменяемый тип данных. Это значит можно изменять какой-то конкретный элемент списка.

Заменим второй элемент списка:

```
>>> friends = ['Sasha','Dasha','Pasha','Masha']
>>> friends[1] = 'Gosha'
>>> print(friends)
['Sasha', 'Gosha', 'Pasha', 'Masha']
```

Добавление

Добавить элемент в список можно с помощью операции, точнее тут происходит прибавление одного списка к другому и запись результата в первый список:

```
>>> friends += ['Dasha']
>>> print(friends)
['Sasha', 'Gosha', 'Pasha', 'Masha', 'Dasha']
```

Так же добавить элемент в список можно с помощью метода append():

```
>>> friends.append('Masha')
>>> print(friends)
['Sasha', 'Gosha', 'Pasha', 'Masha', 'Dasha', 'Masha']
```

Вставка

Вставить элемент в список можно с помощью метода insert(), при этом все элементы правее вставляемого элемента сдвигаются вправо.

Вставим элемент между вторым и третьим элементом, при этом в параметрах метода мы указываем будущую индекс нового элемента:

```
>>> friends = ['Sasha','Dasha','Pasha','Masha']
```

```
>>> friends.insert(2, 'Misha')
>>> print(friends)
['Sasha', 'Dasha', 'Misha', 'Pasha', 'Masha']
```

Удаление

Удаление элемента из списка возможно с помощью метода remove(), который удаляет элемент по значению, при этом удаляется только первое вхождение элемента:

```
>>> print(friends)
['Sasha', 'Misha', 'Dasha', 'Pasha', 'Masha', 'Dasha']
>>> friends.remove('Dasha')
>>> print(friends)
['Sasha', 'Misha', 'Pasha', 'Masha', 'Dasha']
```

Удалять элемент можно с помощью конструкции del, которая удаляет элемент по индексу:

```
>>> print(friends)
['Sasha', 'Misha', 'Pasha', 'Masha', 'Dasha']
>>> del friends[1]
>>> print(friends)
['Sasha', 'Pasha', 'Masha', 'Dasha']
```

Но если такого элемента или индекса нет в списке, то оба этих метода вызовут ошибку, поэтому перед удалением надо проверять наличие элемента или индекса в списке.

2.4 Сортировка списка

Упорядочить элементы списка можно с помощью функции sorted(), при этом сортировка не повлияет на исходный список и результат сортировки надо будет сохранять в другой список:

```
>>> print(friends)
['Sasha', 'Pasha', 'Masha', 'Dasha']
>>> sorted(friends)
```

```
['Dasha', 'Masha', 'Pasha', 'Sasha']
>>> print(friends)
['Sasha', 'Pasha', 'Masha', 'Dasha']
```

Второй способ упорядочить список это использовать метод sort() списка, при этом изменится сам список:

```
>>> print(friends)
['Sasha', 'Pasha', 'Masha', 'Dasha']
>>> friends.sort()
>>> print(friends)
['Dasha', 'Masha', 'Pasha', 'Sasha']
```

Сортируются списки только с однотипными элементами. Сортировка списка из строки и чисел вызовет ошибку.

Упомянем здесь также функцию reversed() и метод reverse() для зеркального отображения списка. Функция reversed() сам список не изменяет и возвращает итератор для создания нового списка:

```
>>> print(friends)
['Sasha', 'Dasha', 'Pasha', 'Masha']
>>> a = list(reversed(friends))
>>> a
['Masha', 'Pasha', 'Dasha', 'Sasha']
```

Метод reverse() изменяет порядок элементов в самом списке:

```
>>> print(friends)
['Sasha', 'Dasha', 'Pasha', 'Masha']
>>> friends.reverse()
>>> print(friends)
['Masha', 'Pasha', 'Dasha', 'Sasha']
```

2.5 Присвоение списка другой переменной

При работе со списками важно помнить, что при создании списка переменная ссылается на ячейку памяти, откуда начинается список.

Например:

```
>>> a = [3, 5, 7]
>>> b = a
>>> b
[3, 5, 7]
>>> a[1] = 9
>>> b
[3, 9, 7]
>>> b[0] = 11
>>> a
[11, 9, 7]
```

Получается, что при присвоение знаяения переменной **a** в переменную **b**, переменная **b** тоже начинается ссылаться на тот же список в памяти. Об этом нужно помнить, чтобы случайно не изменить список из другой переменной.

(? если спросят) Чтобы создать список **b** на основании списка **a** нужно использовать конструктор списков list():

```
>>> a = [3, 5, 7]
>>> b = list(a)
>>> b
[3, 5, 7]
>>> a[1] = 9
>>> b
[3, 5, 7]
```

2.6 Генератор списков (List Comprehension) и выражение генератор (Generator Expression)

Списки могут создаваться различными способами, мы можем записывать значения считывая их с клавиатуры, или можем заполнять списки считывая данные из файла или получая из БД. Списки могут быть очень большими.

Для заполнения списка в Python можно использовать:

– генератор списков – list comprehension,

– выражение генератор (более точно, генерирующее выражение) – generator expression.

Рассмотрим как это работает. Генератор списков запускается в квадратных скобках:

```
>>> a = [0 for i in range(3)]
>>> a
[0, 0, 0]
>>> a = [i for i in range(3)]
>>> a
[0, 1, 2]
>>> a = [int(i) for i in input().split()]
7 8 9
>>> a
[7, 8, 9]
```

Сначала объявляем переменную `a`, затем в квадратных скобках, по которым Python понимает, что у нас будет список, пишем цикл нужного нам размера. Интерпретатор Python сгенерирует все значения, которые мы задали через цикл, сохранит их в памяти и запишет получившийся ряд значений в список.

В первом примере переменная i принимает поочередно все значения из range(), т.е. от 0 до 2, и для каждого i срабатывает выражение, указанное перед for, в данном случае выражение это просто константа 0.

А что же делает generator expression. Рассмотрим код:

```
>>> a = (i for i in range(3))
>>> a
<generator object <genexpr> at 0x00000225959503 50>
```

Код выводит совсем не список. Всё верно, потому что этого списка нет в памяти. Выражение генератор будет выдавать нам по элементу, когды мы к нему обратимся, вот так:

```
>>> for i in a:
...     print(i, end=' ')
0 1 2
```

На самом деле просто выражением генератор мы никакой список не заполним, не используя дополнительные функции, например конструктор list().

Использование generator expression там, где это подходит, экономит память и время, потому что не хранит объекты в памяти и не обращается туда.

2.7 Двумерные списки

Иногда требуется использовать таблицы или матрица, в Python матрицы можно хранит с помощью двумерных списков. Например, матрицу:

1 2 3

4 5 6

7 8 9

можно создать следующим образом:

```
>>> a = [[1,2,3],[4,5,6],[7,8,9]]
```

a – это список, каждый элемент которого тоже является списком.

Таким образом, если извлечь один элемент из **a** , то мы получим сразу три значения, т.е. весь первый элемент:

```
>>> a[1]
[4, 5, 6]
```

Указывая двойную индексацию, где первый индекс – индекс элемента самого списка **a**, а второй индекс – это индекс элемента внутреннего списка, получим одно значение:

```
>>> a[1][2]
6
```

Перебрать все элементы двумерного списка (например, нужно вывести его на экран в виде матрицы) можно с помощью двух вложенных циклов:

```
>>> for i in a:
...     print('\t'.join([str(j) for j in i]))
1       2       3
4       5       6
7       8       9
```

Во внешней цикле мы получаем элемент списка **a**, т.е. список **i**, а во вложенном цикле получаем элемент **j** из подсписка **i**.

2.8 Генерация двумерных списков

Заполнить двумерный список тоже можно с помощью генератора списка. Для этого можно использовать двумерный цикл:

```
>>> a = [[i*3+j+1 for j in range(3)] for i in range(3)]
>>> for i in a:
...     print(i)
...
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
```

Генератор списка, где цикл бежит по `i`, в качестве выражения имеет другой генератор списка, где цикл бежит по `j`, и у этого внутреннего цикла выражение `i*3+j+1` вычисляет значения для списка.

3 Множества (set)

Множество (set) – это набор элементов, является не изменяемым типом данных и содержит только уникальные значения.

Создать множество можно перечислением элементов в фигурных скобках:

```
>>> a = {'milk','butter','bread','cucumber'}
```

Пустое множество создается двумя способами:

```
>>> a = {}
>>> a = set()
```

3.1 Методы множеств

Добавление

В множество можно добавлять элементы, если такой элемент уже существует в множестве, то элемент добавлен не будет.

```
>>> a = {'milk','butter','bread','cucumber'}
>>> a.add('banana')
```

```
>>> a
{'milk', 'banana', 'cucumber', 'butter', 'bread'}
```

Удаление

Из множества можно удалить элемент методом remove(), метод вызовет ошибку, если такого элемента в множестве нет. Чтобы избежать ошибки, следует сначала проверить элемент a вхождение во множество.

```
>>> a.remove('cucumber')
>>> a
{'milk', 'banana', 'butter', 'bread'}
```

Существует метод удаления элемента, который не вызывает ошибки, если элемент отсутствует в множестве – discard():

```
>>> a.discard('tea')
>>> a
{'milk', 'banana', 'butter', 'bread'}
```

Наконец, есть метод очистки множества clear():

```
>>> a.clear()
```

3.2 Перебор элементов множества

Элементы множества можно перебирать в цикле for:

```
>>> for item in a:
...     print(item)

milk
bread
cucumber
butter
```

4 Словари (dictionary)

Словарь (dictionary) – позволяет хранить набор пар ключ-значение, по значению ключа получать значение.

Словари изменяемый тип данных.

Ключи должны быть уникальными, и ключи не изменяемы.

Значения могут быть любыми, в том числе списки значений.

Создать словарь можно перечислением пар через запятую, где ключ и значение разделяются двоеточием, а все пары заключаются в фигурные скобки:

```
>>> a = {'sql':'RDBMS','mongo':'DO','redis':'KV'}
```

В первой паре ‘sql’ – это ключ, а ‘RDBMS’ – значение.

Создать пустой словарь можно с помощью конструктора dict():

```
>>> a = dict()
```

Получают значение по ключу аналогично получению элементы списка по индексу, только вместо индекса в словарях – ключ:

```
>>> a['sql']
'RDBMS'
```

4.1 Операции над словарями

Добавление

Добавить новую пару в словарь можно указав новый ключ и его значение, следующим образом:

```
>>> a['oracle'] = 100
>>> a
{'sql': 'RDBMS', 'mongo': 'DO', 'redis': 'KV', 'oracle': 100}
```

Если указать тот же ключ и новое значение, то значение по ключу изменится.

Извлечение

Извлекать значение по ключу можно разными способами. Первый из них простое указание ключа в квадратных скобках:

```
>>> a['oracle']
100
```

Если такого ключа не существует, то получится ошибка.

Чтобы избежать ошибки, можно использовать метод get():

```
>>> a.get('oracle')
100
```

Удаление

Удалить пару ключ-значение можно с помощью конструкции del, при этом если ключа не существует получим ошибку:

```
>>> del a[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 100
```

4.2 Перебор элементов словаря

Словарь состоит из пар, следовательно из словаря мы можем получить отдельно ключи, отдельно значения или вместо и ключи и значения.

Допустим, у нас есть словарь:

```
>>> a = {'sql': 'RDBMS', 'mongo': 'DO', 'redis': 'KV', 'oracle':
100}
```

Получить список ключей можно двумя способами. Не указав, что нам нужно получить из словаря, по умолчанию получим ключи. Кроме того, существует метод key(), который говорит, что нам нужны именно ключи:

```
>>> for k in a:
...     print(k, end=' ')
sql mongo redis oracle

>>> for k in a.keys():
...     print(k, end=' ')
```

```
sql mongo redis oracle
```

Чтобы получить значения используем другой метод – values():

```
>>> for v in a.values():
...     print(v, end=' ')
RDBMS DO KV 100
```

Чтобы получить и ключи и значения используется метод items():

```
>>> for k, v in a.items():
...     print(k, v, end=', ')
sql RDBMS, mongo DO, redis KV, oracle 100,
```

5 Функции

В любом языке программирования есть механизм функций. Функции помогают структурировать код, полезно выделять какое-то одно действие, которое делает программа, в отдельную функцию. Функции можно переиспользовать чтобы избежать повторения кода.

Функции могут принимать на вход какие-то аргументы или параметры, а затем могут возвращать какие-то значения.

5.1 Объявление функции

Объявление функции обычно помещается в самом начале файла скрипта. Python понимает, что перед ним объявление функции по ключевому слову def. Затем пишется произвольное имя функции и в скобках указываются параметры, которые получает на вход функция, затем ставится двоеточие и со следующей строки начинается описание тела функции, т.е. просто пишется ее код.

Если запустить PyCharm, то он по умолчанию создаст пример кода, в котором будет объявлена функция с именем print_hi():

```
def print_hi(name):
```

```
    print(f'Hi, {name}')
```

5.2 Вызов функции

Функцию можно вызывать из тела программы или из другой функции, указав ее имя и необходимые параметры.

```
print_hi('PyCharm')
```

Выполнив свои действия функция возвращает управление вызвавшей ее процедуре.

5.3 Виды функций

Рассмотрим, какие бывают функции.

1. Не имеют параметров
   т.е. на вход ничего не получают, например:
   ```
   def func_hi ():
         print(‘Hi’)
   ```

2. Имеют произвольное число параметров
   Например, функция print(), ей можно передать множество значений, которые надо вывести в консоль.

3. Не возвращают значение
   Функции обычно возвращают значение, результат своих действий, но бывают функции, которые этого не делают, это скорее процедуры, а не функции.

4. Возвращают значение
   Они вычисляют какой-то результат и возвращают его в вызвавшую процедуру, например, функция перемножает два числа и возвращает произведение:
   ```
   def func_mp (a, b):
   ```

```
        return a*b
```

5. Имеют параметры со значениями по умолчанию
   Например, функция range(), ее параметр step имеет значение по умолчанию равный 1, даже если мы ничего не передали в этом параметре, мы получаем множество значений с шагом 1.

5.4 Функции и видимость переменных

Если переменная объявлена внутри функции, она является локальной переменной. Даже если ее имя совпадает с какой-то другой переменной из программы, это разные переменные и их значения не мешают друг другу.

```
def func(a, b):
   x = 6
   return a*b

if __name__ == '__main__':
   a, b, x = 2 ,3, 4
   print(func(a,b))
   print(x) → 4
```

Переменная **x** останется равна 4.

Если в функцию передается список, то изменение списка внутри функции меняет исходный список, так как список это ссылка на ячейку памяти и мы меняем значение в том же месте памяти.

```
def func(a):
   a.append(5)

if __name__ == '__main__':
   a = []
   func(a)
   print(a) → [5]
```

Список **a** инициализировался значением 5.

Если в функции используется переменная, которая в ней не объявлена, то Python ищет эту переменную среди глобальных, т.е. объявленных в родительском скрипте. Изменение глобальной переменной внутри функции меняет ее значение и в вызвавшей процедуре.

```
def func(a):
   print(a) → Glob

if __name__ == '__main__':
   a = 'Glob'
   func(a)
```

6 Python и Базы данных

Построение приложения которое работает с Базами данных как правило имеет несколько уровней.

На верхнем уровне описывается сущности и какие атрибуты будут у сущностей, описывается бизнес-логика. На этом уровне не интересует какая будет СУБД и какой будет способ хранения. Получаем доменную модель.

На следующем уровне происходит соединение доменной модели со способами хранения данных. Некоторый мэппинг.

На самом нижнем уровне происходит общение с СУБД, тут отправляются запросы, а полученные результаты преобразуются к виду, который воспринимает предыдущий уровень.

Для работы с БД используются драйверы и фреймворки. Мы не будем вдаваться в подробности их работы. Как правило приложения строят так: есть некий сервер приложения, который установил соединение с БД и постоянно держит его открытым, он обрабатывает обращения пользователей и, если нужны данные из БД, обращается к БД по уже установленному соединению.

6.1 Соединение с Redis

Драйверы для работы с БД заключены в библиотеках Python. Например, для работы с Redis потребуется библиотека, которая называется Redis. Чтобы установить ее для интерпретатора в командной строке запускается команда:

```
python -m pip install redis
```

Код программы простейшего обращения в БД:

```
import redis

if __name__ == '__main__':
   r = redis.Redis()
   r.mset({'sql': 'RDBMS', 'mongo': 'DO', 'redis': 'KVRAM',
'oracle': 300})
   print(r.mget('oracle'))
```

Функция mset() записывает в БД множество пар ключ-значение.

Функция mget() извлекает из БД и возвращает нам значение по ключу.

При установке соединения строкой:

`r = redis.Redis()`

мы не указывали ни адрес БД, ни порт. Потому что стандартные значения соединения с БД заданы по умолчанию в этом классе. Если посмотреть описание самого класса, видим:

Python

```
# From redis/client.py
class Redis(object):
    def __init__(self, host='localhost', port=6379,
                 db=0, password=None, socket_timeout=None,
                 # ...
```

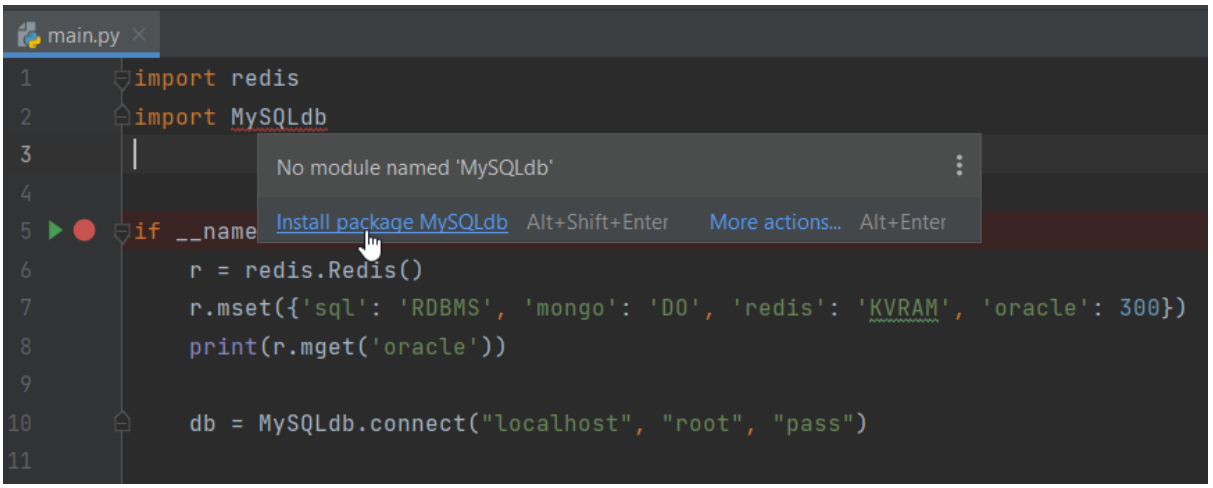
Если проверить содержимое БД, мы увидим наши записанные значения.

```
C:\>docker exec -it redis-port redis-cli
127.0.0.1:6379> keys *
1) "oracle"
2) "mongo"
3) "sql"
4) "redis"
```

Обращаем внимание, что в строке приглашения в клиенте Redis отображается порт, если бы нам требовалось в соединении с БД указать порт, то нам надо было бы его знать таким образом, зайдя в клиент Redis.

6.2 Библиотека MySQL

Библиотека для работы с MySQL называется MySQLdb. Следующий скриншот показывает, как установить ее из PyCharm. Если навести курсор мыши на отсутствующую библиотеку, то PyCharm предложит установить ее.



PyCharm сам скачает и установит нужный пакет.