

jenyay.net

Софт, исходники и фото

Поиск:

>>

[Домой](#) [Блог](#) [Контакты](#)[Печать](#) [Править](#)

Блог

Программки

OutWiker (rus)[Плагины](#)[Бета-версии](#)[Локализации](#)[Документация](#)[Предложения и](#)[баги](#)[Исходники](#)**OutWiker (en)**[Plug-ins](#)[Beta versions](#)[Translate](#)[Suggestions and](#)[bugs](#)[Source code](#)[Documentation](#)**Другие...**

Программирование

[Python](#)[Rust](#)[.NET/C#](#)[C++](#)[PHP](#)[Алгоритмы](#)[Инструменты](#)[Остальное](#)

Обзоры книг

Программирование скриптов для Vim. Часть 3. Работа со списками

Предыдущие части

[Часть 1. Запуск скриптов](#)[Часть 2. Переменные](#)

Оглавление

[Создание списков](#)[Получение элементов списков](#)[Срезы](#)[Распаковка](#)[Оператор for](#)[Функция range\(\)](#)[Присоединение списков и удаление элементов](#)[Глубокие копии](#)[Полезные функции для работы со списками](#)[Практика](#)[Комментарии](#)

В [прошлой части](#) мы рассматривали работу с переменными, в основном простейших типов - целые числа, числа с плавающей точкой и совсем немного коснулись строк. Теперь рассмотрим возможности, которые предоставляет Vim для работы с более сложным типом List (сложными в том смысле, что списки хранят переменные других типов). Те из вас, кто знаком с языком Python, будут приятно удивлены тем, насколько работа со списками похожа в Vim и Python.

Создание списков

Студентам

Фото

Животные
Черно-белые
Пейзажи/Природа
Город
Закаты
Панорамы
Спорт
Репортаж
Разное

Контакты

Чтобы создать списочную переменную используются квадратные скобки, в которых через запятую перечисляются значения, которые должны быть включены в список. В следующем примере создается несколько переменных-списков.

```
let foo = [1, 2, 3, 4, 5]
let bar = ["abyrvalg", "spam", "bla-bla-bla"]
let spam = [1, 3.0, "Hello, Vim", [3, 4, 5]]

echo foo
echo bar
echo spam
```

[Исходник](#)

Здесь мы создаем три переменные. Список *foo* хранит целые числа, список *bar* - строки, а список *spam* одновременно целое и дробное числа, строку и даже другой список, состоящий из трех целых чисел. Запустим этот скрипт с помощью команды

```
:source %
```

Vim в ответ на это напишет следующий текст:

```
[1, 2, 3, 4, 5]
['abyrvalg', 'spam', 'bla-bla-bla']
[1, 3.0, 'Hello, Vim', [3, 4, 5]]
Press ENTER or type command to continue
```

Отсюда видно, что Vim разрешает хранить в одном списке объекты разных типов.

Переменные в скрипте намеренно сделаны глобальными и не были удалены в конце скрипта с помощью оператора *unlet*, чтобы мы могли бы с ними поэкспериментировать из командной строки Vim уже после выполнения скрипта.

Получение элементов списков

Чтобы получить элемент списка по его индексу, как и во многих других языках программирования, используются те же квадратные скобки, внутри которых указывается индекс. Индексы при этом начинаются с 0. Можем прямо в командной строке Vim вывести отдельные элементы списков:

```
:echo foo[0]  
:echo bar[1]  
:echo spam[3]
```

[Исходник](#)

При этом мы будем по мере ввода команд получать

```
1  
spam  
[3, 4, 5]
```

При этом в Vim важно, чтобы квадратные скобки шли сразу же после имени переменной **без пробелов**, иначе интерпретатор может решить, что выражение, содержащее фигурные скобки - это новый список, переданный в качестве второго аргумента для какой-либо операции наподобие *echo*. Для примера выполним следующий скрипт:

```
let foo = [0, 1, 2, 3, 4]  
echo foo[0]  
echo foo [0]
```

[Исходник](#)

В результате Vim выведет две абсолютно разные строки:

```
0  
[0, 1, 2, 3, 4] [0]
```

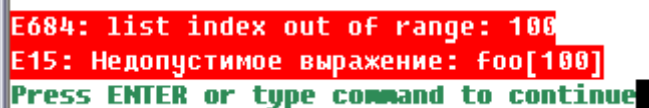
Это произошло из-за того, что в первом случае интерпретатор посчитал выражение `[0]` как взятие индекса, а во втором - как создание нового списка с одним нулевым элементом, и оператор *echo* честно вывела на экран два списка. Поэтому будьте осторожнее с пробелами.

Попытаемся выйти за пределы массива, например, с помощью такой команды

```
:echo foo[100]
```

[Исходник](#)

Мы получим ошибку:



```
E684: list index out of range: 100  
E15: Недопустимое выражение: foo[100]  
Press ENTER or type command to continue
```

Аналогично языку Python, в Vim мы можем использовать индексацию, начиная с конца списка. Для этого используются отрицательные индексы. Так, для получения последнего элемента в списке используется индекс `-1`, для предпоследнего - `-2` и т.д.

В следующем примере мы последовательно выводим последний, предпоследний и первый элементы списка `foo`.

```
:echo foo[-1]  
:echo foo[-2]  
:echo foo[-5]
```

[Исходник](#)

В результате мы получим:

```
5  
4  
1
```

Если мы напишем что-нибудь вроде `foo[-100]`, то тоже получим ошибку.

Обратите внимание на то, что при использовании отрицательных индексов, индексация начинается с `-1`, потому что для Vim `0` и `-0` - это одно и то же. Хотя, если бы можно было бы использовать индекс `-0`, то было бы удобнее, а так приходится мириться с такой несимметрией.

Срезы

И опять же, как и в Python, в Vim есть возможность взятия срезов списков, то есть выделения отдельных интервалов из их середины, хотя в индексации есть и некоторые различия. Официально в Vim это называется `sublist`, но слово "подсписок" как-то плохо звучит, поэтому я буду пользоваться словом "срез", позаимствованным из Python. Следующие примеры будут демонстрироваться на новом списке `foo`.

```
let foo = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

[Исходник](#)

Выражение для взятия среза выглядит следующим образом:

```
foo [first: last]
```

Здесь *foo* - списочная переменная, а *first* и *last* - соответственно первый и последний индекс интересующей нас части списка. В результате будет создан новый список, первый элемент которого является элементом *foo[first]*, а последний - *foo[last]*. При этом мы можем опускать параметр *first*, *last*, или оба, в этом случае вместо *first* будет использоваться значение 0 (первый элемент), а вместо *last* значение -1 (последний элемент).

Те из вас, кто знаком с языком Python, наверняка заметили несоответствие, заключающееся в том, что там новый список, полученный с помощью аналогичной конструкции закончился бы элементом *foo[last - 1]*, поэтому питонерам (питонистам?) здесь придется быть более осторожными.

Теперь попробуем взять различные срезы списка *foo*:

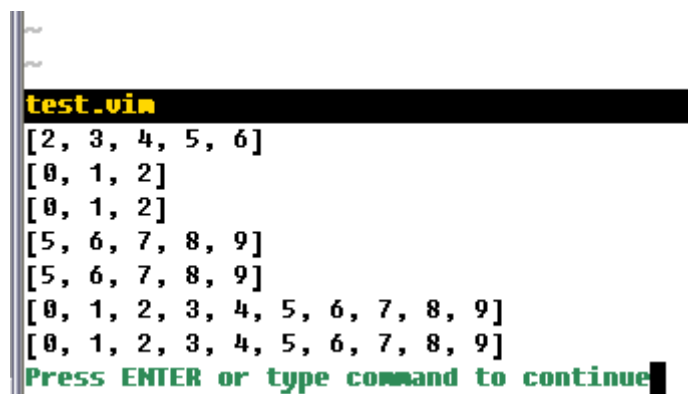
```
let foo = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
echo foo[2: 6]
echo foo[0: 2]
echo foo[: 2]

echo foo[5: -1]
echo foo[5:]

echo foo[0: -1]
echo foo[:]
```

[Исходник](#)

Результат работы этого скрипта выглядит следующим образом:



```
test.vim
[2, 3, 4, 5, 6]
[0, 1, 2]
[0, 1, 2]
[5, 6, 7, 8, 9]
[5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Press ENTER or type command to continue
```

Первый срез создает и сразу же выводит список элементов с индексами от 2 до 6. Затем идут три пары операции взятия срезов, которые попарно дают одинаковый

результат. Первая пара команд *echo* выводит все элементы, начиная с начала списка и до элемента с индексом 2 (до третьего элемента) включительно. Вторая пара выводит элементы, начиная с индекса 5 до конца списка. И, наконец, третья пара команд *echo* выводит весь список от начала и до конца, а если быть более точным, то копирует элементы списка в новый список, который и выводится на экран.

Как вы наверняка заметили, при получении срезов мы можем использовать и отрицательные индексы. В предыдущем примере уже использовался индекс -1 для доступа к последнему элементу списка, а в следующем примере создается срез, заканчивающийся на предпоследнем элементе исходного списка:

```
:echo foo[5: -2]
```

[Исходник](#)

В результате Vim нам покажет следующий результат:

```
[5, 6, 7, 8]
```

У срезов в Vim есть интересная особенность. Если в качестве второй границы будущего среза задать индекс, выходящий за пределы размера списка, то интерпретатор не станет ругаться, как это было при использовании одиночного индекса, а будет считать, что в качестве этого индекса используется наибольший из возможных индексов. Например, следующие примеры дадут одинаковый результат:

```
:echo foo[5:]  
:echo foo[5:9]  
:echo foo[5: -1]  
:echo foo[5: 100]
```

[Исходник](#)

Во всех этих случаях будет получен и выведен на экран список:

```
[5, 6, 7, 8, 9]
```

А вот если мы зададим выходящий за пределы списка индекс в качестве первого элемента среза, то Vim создаст пустой список:

```
:echo foo[100:]
```

[Исходник](#)

В качестве результата получим:

```
[]
```

Распаковка

Кроме перечисленных операций есть также такая удобная штука, как распаковка списков (`unpack`). С помощью этой операции можно разбить список на несколько переменных, которые будут хранить отдельные элементы списка или его часть. Сразу рассмотрим пример:

```
let foo = [0, 1, 2, 3, 4]
let [foo_0, foo_1, foo_2, foo_3, foo_4] = foo
echo foo_0 foo_1 foo_2 foo_3 foo_4
```

[Исходник](#)

После выполнения этого скрипта будут созданы переменные `foo_0`, `foo_1`, `foo_2`, `foo_3` и `foo_4`, каждой из которых будет присвоен по одному элементу списка.

При такой записи обязательным условием является то, чтобы количество перечисленных переменных в квадратных скобках было равно длине списка, в противном случае произойдет ошибка (точнее исключение). Но есть вторая запись, при которой в последней переменной сохраняется не один элемент, а вся оставшаяся часть списка, не поделенная между предыдущими переменными (то, что иногда называют хвостом списка). Чтобы указать интерпретатору, что в последнюю переменную нужно поместить "хвост", хвостовую переменную в квадратных скобках отделяют не запятой, а с помощью точки с запятой:

```
let foo = [0, 1, 2, 3, 4]
let [foo_0, foo_1; tail] = foo
echo foo_0 foo_1 tail
```

[Исходник](#)

В результате на экран будет выведено:

```
0 1 [2, 3, 4]
```

Разумеется, что хвост, может быть только один.

Оператор `for`

В этой части статьи хотелось бы поговорить о таком замечательном операторе как `for`. Не смотря на то, что остальные операторы ветвления типа `while` и `if` мы не пока рассматривали, но именно оператор `for` ближе всего стоит к работе со списками.

Опять же, оператор `for` в Vim аналогичен одноименному оператору в Python или оператору `foreach` в C# и других языках. То есть с помощью этого оператора мы можем последовательно перебирать элементы списка. Существуют два синтаксиса для этого оператора. Первый из них выглядит следующим образом:

```
for foo in foo_list
...
endfor
```

[Исходник](#)

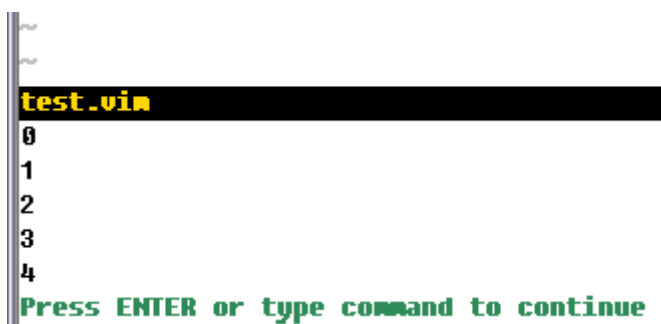
Здесь `foo` - переменная, которая поочередно принимает значение всех элементов списка `foo_list`. При этом обязательным условием является то, чтобы операторы `for`, `endfor` и выражения между ними были бы записаны на отдельных строках так, как указано в синтаксисе. Вместо `endfor` можно использовать сокращенное выражение `endfo`.

Рассмотрим простенький пример:

```
let foo = [0, 1, 2, 3, 4]
for bar in foo
  echo bar
endfor
```

[Исходник](#)

Результат работы, думаю, очевиден:



```
test.vim
0
1
2
3
4
Press ENTER or type command to continue
```

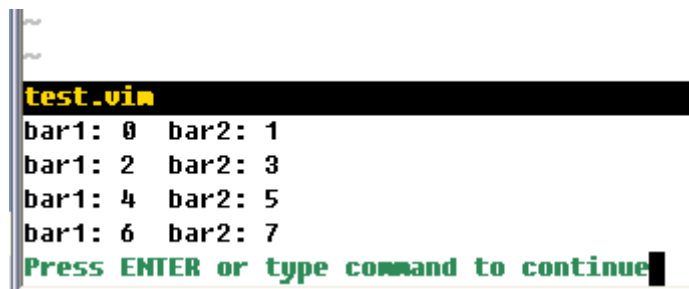

Второй синтаксис для оператора *for* выглядит следующим образом:

```
for [foo_1, foo_2, ...] in foo_list
    ...
endfor
```

[Исходник](#)

Такую запись можно использовать, если каждый элемент списка *foo_list* сам является списком. При этом количество указанных переменных в квадратных скобках должно соответствовать длине внутренних списков-элементов внутри *foo_list*:

```
let foo = [[0, 1], [2, 3], [4, 5], [6, 7]]
for [bar1, bar2] in foo
    echo "bar1:" bar1 " bar2:" bar2
endfor
```

[Исходник](#)

```
test.vim
bar1: 0 bar2: 1
bar1: 2 bar2: 3
bar1: 4 bar2: 5
bar1: 6 bar2: 7
Press ENTER or type command to continue
```

Также как и в других языках, в Vim внутри циклов мы можем использовать операторы *continue* и *break* для начала новой итерации и прерывания цикла. Но про эти конструкции мы поговорим в другой раз, когда будем разбираться с другими операторами ветвления.

Функция `range()`

Те из вас, кто привык программировать на C/C++ могут спросить "А как же нам делать последовательный перебор списка, если нужно знать индекс?" В этом случае приходит на помощь функция *range()*, также знакомая питонерам. Эта функция формирует список, состоящий из последовательности целых чисел. Функция имеет три параметра, из которых обязательным является только первый:

```
range (N, [maxval, [step] ])
```

В зависимости от количества параметров, значение параметра N разное. Так, если задан только один обязательный параметр N , то результирующий список будет выглядеть как последовательность чисел от 0 до $N-1$. Можем набрать в командной строке Vim следующий пример:

```
:echo range (5)
```

[Исходник](#)

В результате на экран будет выведен следующий список:

```
[0, 1, 2, 3, 4]
```

То есть в данном случае N задает количество элементов в списке, а первый элемент является нулевым.

Если мы укажем еще и второй параметр *maxval*, то параметр N приобретет значение первого элемента в формируемом списке, а *maxval* - последнего, и мы получим следующий список:

```
[N, N + 1, N + 2, ..., maxval]
```

Обратите внимание, что, например, в Python такое же использование функции *range()* дало бы на один элемент меньше, там список окончился бы на значении *maxval* - 1. Введем в командной строке следующий пример:

```
:echo range (1, 5)
```

[Исходник](#)

В результате получим список:

```
[1, 2, 3, 4, 5]
```

Если мы зададим все три параметра функции *range()*, то в этом случае первые два параметра будут вести себя аналогично, а последний параметр будет означать шаг приращения между элементами. При этом последний элемент списка будет меньше или равен (не больше) параметра *maxval*.

```
:echo range (1, 5, 2)
```

[Исходник](#)

В результате получим следующий список:

```
[1, 3, 5]
```

Также мы можем использовать отрицательный шаг:

```
:echo range (5, 1, -1)
```

[Исходник](#)

В результате получим список:

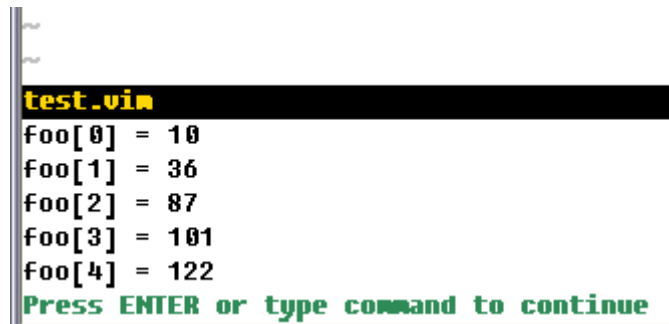
```
[5, 4, 3, 2, 1]
```

Теперь, воспользовавшись оператором *for* и функцией *range()*, мы можем последовательно перебрать все элементы списка:

```
let foo = [10, 36, 87, 101, 122]
for n in range (5)
    echo "foo[" . n . "] =" foo[n]
endfor
```

[Исходник](#)

Результат работы этого скрипта выглядит следующим образом:



```
test.vim
foo[0] = 10
foo[1] = 36
foo[2] = 87
foo[3] = 101
foo[4] = 122
Press ENTER or type command to continue
```

В этом примере в качестве параметра функции *range()* выступает число, равное размеру массива. Это не очень правильно, так как нужно было бы использовать функцию *len()*, которая возвращает размер списка, но мы эту функцию пока еще не проходили :).

Присоединение списков и удаление элементов

До сих пор мы только создавали новые списки (с помощью конструкции [...], с помощью взятия среза или функции *range()*), но разумеется, что должны быть и операции для изменения самих списков. Наиболее важные из них - присоединение элементов к списку и их удаление. Начнем с присоединения элементов.

К спискам можно применять оператор "+" со всеми вытекающими последствиями. Этот оператор, принимающий два списочных параметра (один слева, другой справа от него), создает новый список, элементы которого представляют собой последовательность элементов объединяемых списков. Рассмотрим простой пример:

```
let foo = [1, 2, 3]
let bar = [10, 20, 30, 40]

let spam = foo + bar
echo spam
```

[Исходник](#)

В результате выполнения этого скрипта Vim создаст новый список *spam* и выведет на экран его содержимое:

```
[1, 2, 3, 10, 20, 30, 40]
```

Думаю, что здесь никакие комментарии не требуются. Также к спискам применим оператор "+=". Он дополняет исходный список элементами из другого списка, переданного ему в качестве аргумента:

```
let foo = [1, 2, 3]
let bar = [10, 20, 30, 40]

let foo += bar
echo foo
```

[Исходник](#)

В результате переменная *foo* будет содержать следующий список:

```
[1, 2, 3, 10, 20, 30, 40]
```

Теперь рассмотрим операцию удаления элементов списка. Для этого можно использовать уже знакомую нам по прошлой части операцию *unlet*. Сразу рассмотрим пример:

```
let foo = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
echo foo
```

```
unlet foo[0]
echo foo

unlet foo[-1]
echo foo
```

[Исходник](#)

Здесь мы создаем новый список *foo*, у которого по очереди удаляем сначала первый, а затем последний элемент. После каждой операции выводим полученный список. В результате выполнения этого скрипта мы увидим:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8]
```

Кроме того, для удаления части списка мы можем использовать конструкцию, напоминающую взятие среза:

```
let foo = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
echo foo

unlet foo[2: 5]
echo foo
```

[Исходник](#)

В результате на экране мы увидим следующие две строки:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 6, 7, 8, 9]
```

Глубокие копии

Давайте начнем сразу с примера,

```
let foo = [0, 1, 2, 3]
let bar = foo

let foo += [10, 20, 30]
echo bar
```

[Исходник](#)

Как вы думаете, что в результате будет выведено на экран? Правильно, список из семи элементов:

```
[0, 1, 2, 3, 10, 20, 30]
```

Такое поведение встречается во многих языках программирования, в том же Python, например. Если вас

все-таки смущает такое поведение, то давайте рассмотрим поподробнее что же происходит при выполнении этого скрипта.

Сначала мы создаем список `[0, 1, 2, 3]`, который затем присваивается переменной `foo`. Уже здесь на самом деле присваивается переменной что-то вроде указателя на созданный список.

Во второй строке мы присваиваем переменной `bar` значение `foo`, но по сути мы присваиваем просто указатель на все тот же созданный список.

Затем мы изменяем список по указателю, хранящемуся в переменной `foo`, а именно добавляем в его конец еще три элемента. Так как список у нас создан один для двух переменных `foo` и `bar`, то и для переменной `bar` список изменился и стал хранить семь элементов вместо четырех.

А теперь рассмотрим другой пример

```
let foo = [0, 1, 2, 3]
let bar = foo

let foo = [10, 20, 30]

echo bar
```

[Исходник](#)

В этом примере на экран будет выведен следующий список:

```
[0, 1, 2, 3]
```

Почему? А все из-за того, что с помощью следующей строки мы для переменной `foo` создаем новый список и она больше не указывает на старый.

```
let foo = [10, 20, 30]
```

[Исходник](#)

В этом примере выполняются следующие операции:

1. Создается безымянный список `[0, 1, 2, 3]`.
2. Переменной `foo` присваивается указатель на этот безымянный список.
3. Переменной `bar` присваивается значение переменной `foo`, то есть тот же указатель на список.

4. Создается новый безымянный список `[10, 20, 30]`. Старый список, как вы понимаете, уже нельзя назвать безымянным (на него указывают аж две переменные).
5. Переменной `foo` присваивается указатель на новый безымянный список.
6. Выводится значение переменной `bar`, которая так и хранит указатель на наш первый список.

Если нужно копировать сам список, а не только указатель на него, то для этого удобно воспользоваться операцией взятия среза без указания начала и конца (`[:]`).

Например:

```
let foo = [0, 1, 2, 3]
let bar = foo[:]

let foo += [10, 20, 30]

echo bar
```

[Исходник](#)

В данном случае на экран будет выведен исходный список:

```
[0, 1, 2, 3]
```

Это происходит из-за того, что в случае взятия среза создается новый список, хоть и состоящий из всех элементов старого. Также можно воспользоваться встроенной функцией `copy()`, которая тоже скопирует все элементы в новый список:

```
let foo = [0, 1, 2, 3]
let bar = copy(foo)

let foo += [10, 20, 30]

echo bar
```

[Исходник](#)

Результат работы скрипта в этом случае будет тот же самый.

Рассмотрим теперь еще один пример. Что будет, если один из элементов списка сам является списком и при этом в ходе выполнения скрипта меняется:

```
let foo = [0, 1, 2, 3]
let bar = [111, 222, foo]

echo bar
```

```
let foo += [10, 20, 30]
echo bar
```

[Исходник](#)

В результате выполнения этого скрипта мы наблюдаем как изменяется список *bar*. В итоге на экран нам будет выведено два списка:

```
[111, 222, [0, 1, 2, 3]]
[111, 222, [0, 1, 2, 3, 10, 20, 30]]
```

Этот пример показывает, что даже при включении переменной в список, все-равно в него включается указатель, а не копируется список целиком. Немного усложним пример:

```
let foo = [0, 1, 2, 3]
let bar = [111, 222, foo]

let spam = bar[:]
echo spam

let foo += [10, 20, 30]
echo spam
```

[Исходник](#)

В результате мы увидим то же самое

```
[111, 222, [0, 1, 2, 3]]
[111, 222, [0, 1, 2, 3, 10, 20, 30]]
```

Здесь мы завели новый список *spam*, куда поэлементно скопировали список *bar*, однако это нам не помогло избежать изменения списка *foo*, после добавления новых элементов в который, список *spam* тоже изменился. Здесь даже не поможет функция *copy()*, результат останется тем же.

Как же быть? На помощь нам приходит другая функция *deepcopy()*, которая копирует списки (и словари, о которых мы пока не говорили) рекурсивно. Изменим предыдущий пример следующим образом:

```
let foo = [0, 1, 2, 3]
let bar = [111, 222, foo]

let spam = deepcopy (bar)
echo spam
```



```
let foo += [10, 20, 30]

echo spam
```

[Исходник](#)

И в результате мы, наконец, получим следующий результат:

```
[111, 222, [0, 1, 2, 3]]
[111, 222, [0, 1, 2, 3]]
```

Как видите, внутренний список тоже скопировался поэлементно. Максимальная глубина рекурсии или вложения для функции `deersory()` составляет 100 уровней.

Полезные функции для работы со списками

Давайте теперь кратко рассмотрим некоторые встроенные функции, которые могут пригодиться при работе со списками. Разумеется, здесь перечислены не все функции. Чтобы увидеть все встроенные функции, введите в командной строке Vim следующую команду:

```
:help function-list
```

Название функции	Описание
<code>len()</code>	Возвращает количество элементов в списке.
<code>insert()</code>	Вставить новые элементы в заданную позицию списка.
<code>remove()</code>	Удалить элемент или группу элементов списка. Функция возвращает удаленные элементы.
<code>sort()</code>	Сортировать элементы. Есть возможность задавать функцию для сравнения элементов.
<code>reverse()</code>	Перевернуть элементы списка таким образом, чтобы первый элемент стал последним и наоборот.
<code>get()</code>	Получить элемент списка по индексу, но если индекс не укладывается в размер списка, то можно задать значение по умолчанию, которое будет возвращено в этом случае.

index()

Осуществляет поиск в списке и возвращает индекс найденного элемента.

Практика

Теперь мы можем написать какой-нибудь более жизненный пример. А именно. Допустим, у нас есть открытый файл, тогда при запуске нашего скрипта (пусть он называется `linenumbers.vim`) должен создаваться новый буфер с именем нашего открытого файла, в начале которого должна быть добавлена строка `"numbers_"`. Содержимое нового буфера должно в точности совпадать с содержимым текущего буфера, но в начале каждой строки должен быть добавлен ее номер.

Исходник скрипта с комментариями:

```
" Получим количество строк в текущем буфере
let s:count = line("$")

" Здесь будем хранить все новые строки с нумерацией
let s:newlines = []

" Проходимся по всем строкам в буфере.
for n in range(1, s:count)
    " Получим содержимое строки по ее номеру
    " Строки в буфере нумеруются с 1
    let s:currline = getline(n)

    " Добавим номер строки
    let s:newline = printf("%d: %s", n, s:currline)

    " Добавим новую строку в общий список строк
    call add(s:newlines, s:newline)
endfor

" Имя для нового буфера
let s:bufname = "numbers_" . bufname("%")

" С помощью второго параметра укажем сделать новый буфер
call bufnr(s:bufname, 1)

exe 'sbuffer ' . s:bufname

" Переместимся на начало буфера
normal gg

" Удалим в нем все до конца
normal dG

call append(0, s:newlines)

" Снова переместимся на начало буфера
normal gg

unlet s:count s:newlines s:bufname
unlet! s:currline s:newline
```

[Исходник](#)

И пример работы скрипта:

```

1 1: " Получим количество строк в текущем буфере
2 2: let s:count = line ("$_")
3 3:
4 4: " Здесь будем хранить все новые строки с нумерацией
5 5: let s:newlines = []
6 6:
7 7: " Проходимся по всем строкам в буфере.
8 8: for n in range (1, s:count)
9 9: " Получим номер строки по ее номеру
10 10: " Строки в буфере нумеруются с 1
11 11: let s:currline = getline (n)
12 12:
13 13: " Добавим номер строки
14 14: let s:newline = printf ("%d: %s", n, s:currline)
15 15:
16 16: " Добавим новую строку в общий список строк
17 17: call add (s:newlines, s:newline)
18 18: endfor
19 19:

```

Не будем досконально разбирать каждую строку. Рассмотрим только операторы и функции, которые раньше не встречались.

Все до окончания оператора *for*, думаю, понятно благодаря комментариям в тексте. С помощью функции *bufname()* мы узнаем имя текущего буфера. Ее параметр "%" как раз и определяет, что нас интересует именно текущий буфер. С помощью этой же функции мы можем узнать имя буфера по другим признакам, например по его номеру.

Функция *bufnr()* возвращает номер буфера по его имени. Также она может создать буфер, если он не существует, для этого в качестве второго параметра необходимо передать значение, отличное от 0, например 1. Нас в данном случае не интересует сам номер буфера, мы эту функцию используем только для того, чтобы создать буфер с именем "numbers_*", если он не существует.

После этого мы используем оператор *exe*, позволяющий выполнить команды, которые запускаются через командную строку Vim. В данном случае мы вызываем команду *:sbuffer* (у нее есть короткое имя *sb*), которое разделяет наш буфер на два. В качестве параметра для *sbuffer* мы передаем имя нового буфера. Заодно новый буфер становится текущим.

Затем мы используем оператор *normal*, который позволяет запускать команды Vim, выполняемые в нормальном режиме. Нам нужно очистить текущий буфер "numbers_*", для этого с помощью команды *gg* мы переходим на начало файла, а затем с помощью *dG* удаляем все содержимое до конца буфера.

После этого мы используем функцию *append()*, чтобы добавить в буфер строки с их номерами, а затем снова выполняем операцию *gg*, чтобы курсор переместился на начало буфера.

В конце скрипта используется оператор *unlet!*. Отличие его от оператора *unlet* состоит в том, что *unlet!* не вызывает ошибку (исключение), если удаляемой переменной не существует. Ведь возможно ситуация, когда внутри цикла *for* не будет ни одной итерации, соответственно, переменные *s:currline* и *s:newline* не будут созданы.

Вот, собственно и все.

В следующей части мы рассмотрим работу со строками.

Часть 4. Работа со строками

Вы можете подписаться на новости сайта через [RSS](#), [Группу Вконтакте](#) или [Канал в Telegram](#).



Рейтинг 5.0/5. Всего 38 голос(а, ов)

☐ Плохо ☐ Так себе ☐ Неплохо ☐ Хорошо ☐ Отлично

Голосовать

02.05.2009 - 08:32

Всё здорово, но: 1) %s/Раскаповка/Распаковка 2) подсветка в последнем примере хромает – может, в конце комментов ещё по кавычке поставить? (костыльно, конечно) PS: жду ещё статей :-)

Jenyay 02.05.2009 - 08:47

Спасибо, опечатки исправил :)

По поводу подсветки, да я видел, но почему-то раскрашивалка решила, что комментарий после отступа - это строка. Пока не знаю как с этим бороться.

Lynn 18.05.2009 - 01:44

Очепятки

teil – это липа, а хвост – tail

Jenyay 18.05.2009 - 09:08

Действительно :) Спасибо, исправил.

asn 02.12.2009 - 10:48

comment

В последнем скрипте (тот, что идет ниже строчки "Исходник скрипта с комментариями") есть комментарий для getline

" Получим номер строки по ее номеру

предлагаю заменить на

" Получим значение строки по ее номеру

Jenyay 02.12.2009 - 22:07

asn, спасибо, сейчас исправлю.

Алексей 26.01.2012 - 10:43

Спасибо за статью

Поправьте: "ни одной итерации" -> "ни одной итерации"

Jenyay 26.01.2012 - 11:58

Спасибо, поправил.

Коля 18.04.2012 - 14:16

Спасибо за статью.



[Подписаться на комментарии](#)

Автор:

Тема:

Ваш комментарий



Введите код
745

Послать

© Евгений Ильин 2008-2024 (jenyay.ilin@gmail.com)