Программирование скриптов для Vim. Часть 5. Операции ветвления и функции

Предыдущие части

Часть 1. Запуск скриптов
Часть 2. Переменные
Часть 3. Работа со списками
Часть 4. Работа со строками

Оглавление

В прошлых частях мы уже использовали такие операции ветвления как *if* и *for*, на этот раз мы более подробно обсудим логические операции, операторы *if*, *while*, после этого рассмотрим работу с функциями, а затем научимся делать полноценные плагины для Vim, а не просто скрипты.

Оператор if

Студентам

Фото

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

Контакты

Оператор *if*, как и в других языках программирования, выполняет обычное ветвления по условию. Синтаксис оператора *if* по смыслу мало отличается от других языков программирования и выглядит следующим образом:

```
if Условие_1
    Выражение_1
elseif Условие_2
    Выражение_2
elseif Условие_3
    Выражение_3
...
elseif Условие_N-1
    Выражение_N-1
else
    Выражение_N
endif
```

Исходник

Работает это так: Если *Условие_1* истинно, то выполняется *Выражение_1*, а выполнение оператора *if* на этом завершается. Если *Условие_1* ложно, то проверяется *Условие_2*, если оно истинно, то выполняется *Выражение_2*, и снова завершается выполнение оператора *if*. Если ложно и *Условие_2*, то проверяются все остальные ветви *elseif* вплоть до *Выражение_N-1*. Если все условия в ветках *if* и *elseif* оказываются ложными, то выполняется *Выражение_N* из ветки *else*. Все выражения, разумеется, могут состоять из нескольких операторов, при этом все ветки *elseif* и *else* являются необязательными. Поэтому в простейшем случае оператор может выглядеть следующим образом:

```
if Условие_1
    Выражение_1
endif
```

Исходник

Рассмотрим пример:

```
let s:foo = 0

if s:foo == 0
    echo "Is 0"

elseif s:foo == 1
    echo "Is 1"

elseif s:foo == 2
    echo "Is 2"

else
    echo "I don't know :("
endif

unlet s:foo
```

Исходник

Здесь, меняя переменную *s:foo*, мы можем заставить выполняться различные ветки оператора *if*. Думаю, что этот код в особых комментариях не нуждается. Единственное, на что хотелось бы обратить внимание, что произойдет, если будет две ветки с одинаковыми условиями:

```
let s:foo = 1

if s:foo == 0
    echo "Is 0"

elseif s:foo == 1
    echo "Is 1"

elseif s:foo == 2
    echo "Is 2"

elseif s:foo == 1
    echo "Is 1. Again"

else
    echo "I don't know :("
endif

unlet s:foo
```

Исходник

В данном случае у нас две ветки *elseif* с условием *s:foo == 1*. До второго условия выполнение кода просто не дойдет, так как после выполнения оператора *echo "Is 1"* интерпретатор не будет проверять остальные условия и выйдет за пределы оператора *if*.

Также некоторые элементы оператора *if* могут записываться в сокращенном виде:

```
if Условие_1
    Выражение_1
elsei Условие_2
    Выражение_2
elsei Условие_3
    Выражение_3
...
eseli Условие_N-1
    Выражение_N-1
el
    Выражение_N
en
```

Исходник

То есть наш пример может выглядеть вот так:

```
let s:foo = 1

if s:foo == 0
    echo "Is 0"

elsei s:foo == 1
    echo "Is 1"

elsei s:foo == 2
```

```
    echo "Is 2"
el
    echo "I don't know :("
en
unlet s:foo
```

Исходник

Не знаю как вам, а мне лично больше нравится полная запись, пусть она и длиннее, но вот эти *el* и *en*, имхо, сбивают с толку. В дальнейших примерах я буду использовать только полный вариант записи.

Логические операции

Говоря об операторе *if* нельзя не сказать и о логических операциях, коих в Vim предостаточно. Надо сказать, что в Vim нет отдельного булевого типа, вместо него используются целочисленные значения. За "истину" принимается любое ненулевое значение, а 0, соответственно, принимается за "ложь". Сами логические операции возвращают значения 0 и 1. В этом вы можете убедиться на следующем примере:

```
echo 1 == 1
echo 1 > 0
echo 5 < 2
echo 3 >= 4
echo 0 <= 0
```

Исходник

В этом примере мы использовали простейшие математические операторы сравнения. В результате выполнения этого скрипта будет выведено:

```
1
1
0
0
1
```

Но на то, что операторы сравнения и в дальнейшем при истинном значение будут возвращать именно единицу лучше не рассчитывать.

В прошлой части мы говорили о таких операторах сравнения строк как *==?* и *==#*. Напомню, что первый из них (*==?*) сравнивает строки без учета регистра, а второй (*==#*) - с

учетом регистра независимо от состояния настройки *ignorecase*. Кроме этих операторов для строк существуют еще и другие операторы сравнения, про которые мы не говорили в прошлой части. Это операторы сравнения по регулярному выражению, напоминающие операторы из языка Perl. Вот они:

Оператор	Описание
=~	Проверка на удовлетворение регулярному выражению
!~	Проверка на НЕудовлетворение регулярному выражению
=~?	Проверка на удовлетворение регулярному выражению без учета регистра
!~?	Проверка на НЕудовлетворение регулярному выражению без учета регистра
=~#	Проверка на удовлетворение регулярному выражению с учетом регистра
!~#	Проверка на НЕудовлетворение регулярному выражению с учетом регистра

Выглядят некоторые из этих операторов, конечно, страшно, но давайте рассмотрим примеры.

Допустим, что мы работаем с документами планеты Арракис без помощи ментатов, но с использованием старого компьютера, чудом уцелевшего после Бутлерианского Джихада, и редактора Vim, нам нужно убедиться, что очередная строка содержит номер документа. Для этого мы можем воспользоваться следующим кодом:

```
let s:foo = "Арракинский кризис (для секретного пользования,
ВУ, регистрационный номер Ар-8108858)"

if s:foo =~ 'Ар-\d\+'
    echo "Найден документ"
else
    echo "Документ не найден"
endif

unlet s:foo
```

Исходник

С помощью строки *foo =~ 'Ар-\d\+'* мы проверяем содержится ли в очередной строке *s:foo* номер документа, который начинается с символов "Ар-", после которых идет

некоторое количество цифр. Мы воспользовались оператором *=~*, который возвращает истину (значение 1), если строка или переменная, находящаяся слева от него, удовлетворяет регулярному выражению, записанному справа от оператора. Причем заметьте, что совпадать с регулярным выражением может только часть строки, а не обязательна вся строка целиком. В нашем примере Vim скажет "Найден документ".

Тот же самый пример мы можем переписать с использованием обратного оператора !~, который возвращает истину (1), если строка слева от оператора не удовлетворяет регулярному выражению справа от него.

```
let s:foo = "Арракинский кризис (для секретного пользования,
ВУ, регистрационный номер Ар-8108858)"

if s:foo !~ 'Ap-\d\+'
    echo "Документ не найден"
else
    echo "Найден документ"
endif

unlet s:foo
```

Исходник

В этих примерах мы использовали одинарные кавычки, чтобы нам не пришлось удваивать слеши. В использовании логических операторов для регулярных выражений есть еще один интересный момент. Дело в том, что при их использовании по умолчанию считается, что используется магический режим (установлен параметр *magic*), что позволяет не задумываться над тем включен ли этот параметр у конечных пользователей, который будут выполнять наш скрипт, то есть, благодаря этому улучшается переносимость скриптов. Поэтому перед знаком "+" нам пришлось поставить обратный слеш, чтобы он считался управляющим символом, а не литерой. Но несмотря на это, мы можем по-прежнему использовать такие параметры как \m, \M, \v, и \V внутри регулярных выражений. Давайте перепишем пример с использованием особо магического режима (параметр \v)

```
let s:foo = "Арракинский кризис (для секретного пользования,
ВУ, регистрационный номер Ар-8108858)"

if s:foo =~ '\vAp-\d+'
    echo "Найден документ"
else
    echo "Документ не найден"
endif

unlet s:foo
```

Исходник

Результат будет тот же самый.

Остальные операторы для регулярных выражений работают точно так же за исключением того, что можно установить будет ли использоваться регистр символов или нет. Допустим, теперь нам нужно найти упоминание о планете Арракис, она же в некоторых переводах Арраки, она же Дюна. Для этого мы можем воспользоваться следующим кодом, в котором используется проверка на удовлетворение регулярному выражению с учетом регистра, чтобы не реагировать на слово "дюна" в значении "гора песка".

```
let s:foo = "Настоящая его родина - Арракис, планета, более известная под названием Дюна."

if s:foo =~# '\vАрракис?|Дюн[аые]'
    echo "Найдено упоминание Дюны"
else
    echo "Упоминание о Дюне не найдено"
endif

unlet s:foo
```

Исходник

Здесь мы увидим, что Vim найдет упоминание Дюны (аж два раза, но нас в данный момент это не интересует). А вот в следующей строке упоминание планеты не будет найдено:

```
let s:foo = "За  выступами  твердых  пород  уходили  к горизонту дюны"

if s:foo =~# '\vАрракис?|Дюн[аые]'
    echo "Найдено упоминание Дюны"
else
    echo "Упоминание о Дюне не найдено"
endif

unlet s:foo
```

Исходник

Другие операции

Такие булевы операции как конъюнкцию (И), дизъюнкция (ИЛИ) и отрицание (НЕ) в Vim записываются точно так же, как в C/C++ и им подобных языках, то есть соответственно: &&, || и !. Не думаю, что здесь требуются какие-то дополнительные комментарии, поэтому просто пример:

```
echo 1 > 0 && 5 < 10
echo 2 < 6 || 5 > 10
echo !(1 < 5)
```

Исходник

В результате на экран будут выведены строки:

```
1
1
0
```

Теперь поговорим о сравнении сложных объектов. Пусть у нас есть два списка с одинаковым содержимым и мы хотим их сравнить:

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

echo s:foo == s:bar

unlet s:foo s:bar
```

Исходник

Разумеется, в результате выполнения скрипта мы увидим единицу. Теперь рассмотрим следующий код:

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

let s:spam = s:bar

echo s:foo == s:spam

unlet s:foo s:bar s:spam
```

Исходник

Разумеется, ничего не изменилось. Но когда мы говорили про глубокие копии в 3 части, то выяснили, что операция присваивания не копирует список, а просто присваивает переменной указатель на старый список. То есть сейчас переменные *s:spam* и *s:bar* указывают на один и тот же участок в памяти. А можем ли мы узнать указывают ли переменные на один и тот же список или он просто имеет одинаковые элементы? Разумеется, можем, для этого в Vim предусмотрен специальный оператор *is*, который возвращает истину, если переменные слева и справа от него являются указателями на одну и ту же область в памяти, и ложь в противном случае. Воспользуемся этим оператором.

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

let s:spam = s:bar

echo s:bar == s:spam
echo s:bar is s:spam

echo s:foo == s:spam
echo s:foo is s:spam
```

```
unlet s:foo s:bar s:spam
```

Исходник

Сначала мы сравниваем на равенство элементов *s:bar* и *s:spam*, оператор *==* возвращает 1. Затем мы используем оператор *is* и узнаем, что *s:bar* и *s:spam* указывают на один и тот же список (оператор *is* возвращает 1). После этого мы применяем оператор *==* к переменным *s:foo* и *s:spam*. Опять получаем 1. А затем уже применяем к ним же оператор *is* и получаем 0, то есть хоть эти переменные и хранят одинаковые элементы, но они являются двумя разными списками. Общий результат запуска выглядит следующим образом:

```
1
1
1
0
```

Кроме того, существует обратный оператор *isnot*, который возвращает 1, если переменные указывают на разные участки памяти:

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

let s:spam = s:bar

echo s:bar isnot s:spam
echo s:foo isnot s:spam

unlet s:foo s:bar s:spam
```

Исходник

В результате Vim напишет:

```
0
1
```

Можем провести еще один интересный эксперимент, который для практического применения вряд ли пригодится, а вот для понимания работы Vim может быть полезным:

```
let s:foo = 10
let s:bar = 10

echo s:bar is s:foo

unlet s:foo s:bar
```

Исходник

Как вы думаете, что будет выведено в результате? Ну так не честно, вы знали, что 1 :). Несмотря на то, что мы, казалось бы, дважды создали переменную со значением 10, но интерпретатор на самом деле создал только одно значение 10, а затем переменные *s:foo* и *s:bar* стали указывать на одну и ту же ячейку памяти. Действительно, зачем создавать дважды одно и то же целое число, которое все-равно не может быть изменено по указателю. Ведь если мы затем напишем строку вроде *let s:bar = 20*, то будет создана еще одно целое значение 20, и переменная *s:bar* будет указывать уже на нее, и оператор *s:bar is s:foo* станет возвращать 0. При этом значение *s:foo* не изменится и будет равно 10.

В конце раздела рассмотрим еще один оператор, знакомый С-шникам и им сочувствующим - это оператор *?*. Здесь никаких особенностей нет, синтаксис оператора выглядит следующим образом:

```
Условие ? Выражение_1 : Выражение_2
```

Значение оператора *?* равно значению, возвращаемому *Выражением_1*, если значение *Условия* отлично от нуля, то есть истинно. В противном случае значение оператора *?* равно значению, возвращаемому *Выражением_2*. Рассмотрим несколько примеров:

```
echo 1 > 5 ? "1 > 5" : "1 < 5"

let s:foo = 1 > 5 ? "1 > 5" : "1 < 5"
echo s:foo

unlet s:foo
```

Исходник

В результате обоих вызовов оператора *?* он вернет строку "1 < 5", только в первом случае эта строка будет направлена непосредственно команде *echo*, а во втором - присвоена переменной *s:foo*.

Оператор while

При описании работы со списками мы уже рассмотрели оператор for, пришло время рассмотреть второй оператор для организации цикла - *while*. Оператор *while* опять же полностью копирует одноименный оператор из других языков программирования. Так же как и *if*, *while* имеет полный и

сокращенный синтаксис. Полный синтаксис выглядит следующим образом:

```
while Условие
    Тело цикла
endwhile
```

Исходник

Сокращенный синтаксис выглядит так:

```
wh Условие
    Тело цикла
endw
```

Исходник

При использовании этого оператора тело цикла будет повторяться до тех пор пока *Условие* будет отличаться от 0 (то есть будет истинно). Рассмотрим пример, который выводит текст из текущего буфера в строку Vim:

```
" Узнаем количество строк
let s:count = line("$")

" Номер текущей строки
let s:index = 1

" Пока не дойдем до последней строки
while s:index <= s:count
    " Выведем строку с номером s:index
    echo getline (s:index)

    " Перейдем к следующей строке
    let s:index += 1
endwhile
```

Исходник

Если вы откроете этот скрипт и выполните его с помощью команды *:source %*, то Vim выведет текст самого скрипта.

Функции

Вот и подошло время познакомиться с тем, как в Vim можно выделить участки кода в отдельные функции и что вообще с этими функциями можно делать. Вообще эта тема довольно большая, поэтому в этой статье мы рассмотрим только основы, а более "продвинутые" возможности оставим на следующий раз.

Для объявления функции существует два вида записи: полная и сокращенная, которые ничем не отличаются с точки зрения интерпретатора. Полная запись выглядит следующим образом:

```
function Имя_функции(Аргументы)
    ...
endfunction
```

Исходник

Обязательным требованием является то, что либо имя функции должно начинаться с заглавной буквы, либо область видимости функции должна быть объявлена как *s:*, то есть функция видима только внутри скрипта, в противном случае Vim выдаст ошибку.

Во втором случае объявление функции будет выглядеть следующим образом:

```
function s:имя_функции(Аргументы)
    ...
endfunction
```

Исходник

Имена аргументов функции перечисляются через запятую. Внутри тела функции может встречаться оператор *return*, возвращающий значение функции и прерывающий ее выполнение. Кроме того, после списка аргументов (после закрывающейся скобки) могут указываться дополнительные настройки функции, но о них мы поговорим в другой раз.

Сокращенная форма записи выглядит следующим образом:

```
fu Имя_функции(Аргументы)
    ...
endf
```

Исходник

Для вызова функции используется команда *call*, с которой мы уже встречались в предыдущих частях. Правда, у этой команды есть еще и другая форма, но пока мы ее тоже не будем рассматривать.

Рассмотрим следующий пример. Сразу скажу, что нем есть потенциальная проблема, которую мы потом научимся обходить. Скопируйте следующий пример в окно Vim и запустите его с помощью команды *:source %*.

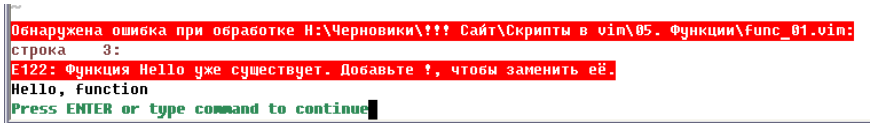
```
function Hello()
    echo "Hello, function"
endfunction

call Hello()
```

Исходник

В результате на экран будет выведена строка "Hello, function".

Области видимости для функции

Казалось бы, все замечательно, и нам так понравилась предыдущая суперполезная функция, что мы решили ее запустить еще раз, чтобы полюбоваться на результат ее работы. Еще раз запускаем команду *:source %* и получаем следующие ошибки:



В принципе, сообщения об ошибках достаточно ясные. Мы запустили скрипт второй раз, а интерпретатор видит, что функция с таким именем уже была, и сообщает нам об этом. Здесь есть два выхода. Первый - указать интерпретатору, что нет ничего страшного в том, что такая функция уже существует, поставив после ключевого слова *function* (или *fu* в сокращенной записи) символ *!*. Тогда интерпретатор, когда увидит эту строку молча заменит старую функцию на новую и будет счастлив, а мы не получим ошибки. Второй выход заключается в том, чтобы удалить функцию после того как она нам станет ненужной, для чего предусмотрена специальная команда:

```
delfunction Имя_функции
```

Исходник

Или в сокращенном виде:

```
delf Имя_функции
```

Исходник

Команда *delfunction* аналогична команде *unlet* для переменных. Давайте сначала воспользуемся первым способом решения проблемы и перепишем наш скрипт следующим образом:

```
function! Hello()
    echo "Hello, function"
endfunction

call Hello()
```

Исходник

Теперь мы можем запускать скрипт сколько угодно раз.

Или то же самое в сокращенном виде:

```
fu! Hello()
    echo "Hello, function"
endf

call Hello()
```

Исходник

А теперь задумаемся над тем, что же мы тут сделали. Смотрите, мы запустили скрипт в первый раз, все прошло нормально. Запускаем скрипт второй раз, а интерпретатор уже находит нашу функцию, значит где-то она осталась храниться. Так оно и есть, дело в том, что для функции действуют две области видимости, напоминающие те что мы рассматривали для переменных. А именно, для функций, объявленных просто *Foo()* без указания области видимости действует глобальная область видимости, то есть эту функцию можно вызывать из разных скриптов. Если перед именем функции стоит префикс *s:*, то функция видна только внутри скрипта, где она была объявлена.

Хорошо, если мы только что создали глобальную функцию, то ее можно вызывать откуда угодно. Это действительно так и в этом легко убедиться, если после команды *:source %* выполнить в командной строке команду

```
:call Hello()
```

Исходник

В результате мы увидим на экране все ну же надпись "Hello, function". А теперь воспользуемся вторым способом решения того, что функция пытается определяться второй раз. Изменим наш скрипт, убрав символ *!* и добавив новую строку для удаления функции:

```
function Hello()
    echo "Hello, function"
endfunction

call Hello()

delfunction Hello
```

Исходник

Только прежде чем запускать этот пример перезапустите Vim или выполните команду *:delf Hello* или *:delfunction Hello*, чтобы уничтожить функцию, которая до этого уже

была создана. Теперь мы снова можем вызывать наш скрипт с помощью команды *:source %* сколько угодно раз, но теперь мы не сможем вызвать функцию из командной строки, так как после выхода из скрипта функция уже будет уничтожена.

Если наша функция используется, например, только внутри скрипта, то лучше всего и ограничить ее видимость скриптом:

```
function s:Hello()
    echo "Hello, function"
endfunction

call s:Hello()

delfunction s:Hello
```

Исходник

Обратите внимание на то, что такое переопределение функций не есть перегрузка функции, а есть ее переопределение. То есть, если мы напишем следующий скрипт:

```
function! Hello()
    echo "Hello, function"
endfunction

function! Hello(a)
    echo "Hello 2"
endfunction

echo Hello()
```

Исходник

то при его выполнении мы получим сообщение об ошибке, говорящее, что для функции *Hello()* недостаточно параметров. И это правильно, ведь в момент вызова функции существует только вторая функция *Hello()*, ожидающая один параметр.

Аргументы функции и локальные переменные

Теперь пришло время разобраться с передачей аргументов внутрь функции и возвратом значения из нее. Давайте разберемся в этом на простом примере.

```
function! s:summ(x, y)
    return a:x + a:y
endfunction

echo s:summ(1, 2)
```

```
delfunction s:summ
```

Исходник

Заметьте, что здесь имя функции начинается с нижнего регистра. Это допустимо, так как функция объявлена с областью видимости *s:*. Внутрь функции мы передаем два параметра *x* и *y*; обратите внимание на то, что для доступа к ним внутри функции используется префикс *a:*, который обозначает, что это аргумент функции. Обратите внимание, что в объявлении функции имена аргументов записываются без этого префикса. Значение переменных с таким префиксом нельзя изменить. Функция возвращает сумму переданных чисел с помощью оператора *return*.

Если мы хотим использовать глобальные переменные внутри функции, то мы должны в явном виде указывать область видимости *g:*. Это показано в следующем примере:

```
let z = 10

function! s:summ(x, y)
    return a:x + a:y + g:z
endfunction

echo s:summ(1, 2)

unlet z
delfunction s:summ
```

Исходник

В результате будет выведено число *13*.

Если вместо *g:z* мы напишем просто *z*, то интерпретатор будет искать эту переменную в локальной области видимости внутри функции. Проведем эксперимент и объявим две переменные *z*: вне функции (глобальную) и внутри (локальную):

```
let z = 10

function! s:summ(x, y)
    let z = 200
    return a:x + a:y + z
endfunction

echo s:summ(1, 2)

unlet z
delfunction s:summ
```

Исходник

В данном случае на экран будет выведено число *203*. Если вы помните, то при обсуждении области видимости

переменных была таблица, где упоминалась область видимости *l:* - локальная область. Именно там и была создана переменная *z*, объявленная внутри функции. Мы можем указать область видимости в явном виде:

```
let z = 10

function! s:summ(x, y)
    let z = 200
    return a:x + a:y + l:z
endfunction

echo s:summ(1, 2)

unlet z
delfunction s:summ
```

Исходник

Результат не изменится.

Точно так же внутри функции нужно обязательно указывать область видимости *s:*:

```
let s:z = 10

function! s:summ(x, y)
    return a:x + a:y + s:z
endfunction

echo s:summ(1, 2)

unlet s:z
delfunction s:summ
```

Исходник

До этого раздела мы использовали функцию *Hello()* которая не возвращала значения с помощью оператора *return*. А что же будет, если все-таки попытаться выполнить следующий скрипт:

```
function! Hello()
    echo "Hello, function"
endfunction

echo Hello()

delfunction Hello
```

Исходник

Здесь кроме того, что выводится на экран внутри функции *Hello()*, мы выводим то, что вернет эта функция. В результате мы увидим:

```
Hello, function
0
```

Таким образом, если функция в явном виде не возвращает какое-то значение, то считается, что она возвращает 0. Аналогично мы можем записать функцию с использованием оператора *return* без значения:

```
function! Hello()
    echo "Hello, function"
    return
endfunction

echo Hello()

delfunction Hello
```

Исходник

После запуска скрипта будут выведены все те же строки.

Практика

И завершим эту часть статьи, как обычно, каким-нибудь полезным скриптом. Давайте усовершенствуем пример, созданный в прошлой части. Напомню, что там мы делали скрипт, который в тексте выискивал заголовки в нотации pmWiki и делал по ним оглавление. После написания статьи я немного доработал скрипт, но это чисто косметические исправления.

Для начала обернем весь наш старый скрипт в функцию. Предварительно нам понадобится изменить область видимости всех переменных с *s:* на *l:*. Можно вообще удалить все области видимости, и тогда внутри функции они по умолчанию будут как *l:*, но для ясности будем указывать пространство имен. Замену эту можно произвести с помощью следующей команды (к счастью, скрипт у нас маленький и никаких проблем из-за такой грубой замены у нас не должно быть):

```
:%s/s:/l:/g
```

Исходник

Затем добавим объявление функции (сделаем ее глобальной с именем *PmTitles()*). В результате получим код:

```
function! PmTitles()
    " Регулярное выражение для нахождения якоря
    let l:pattern_anchor = '\m^\[\[#\([a-zA-Z0-9_]*\)\]\]'

    " Регулярное выражение для нахождения заголовка
    let l:pattern_title = '\m^!!\s*\(.*\)$'

    " Количество строк в файле
    let l:count = line ("$")
```

```
    " Сюда будут добавляться строки оглавления
    let l:result = []

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

        " Для каждой строки проверим соответствует ли она
шаблону с якорем
        let l:anchor = matchlist (l:currline,
l:pattern_anchor)

        if len (l:anchor) != 0
            " Если строка соответствует, то список не будет
пустым

            " Теперь проверим соответствует ли следующая
строка шаблону заголовка
            let l:nextline = getline (n + 1)
            let l:title = matchlist (l:nextline,
l:pattern_title)

            if len (l:title) != 0
                " Если и заголовок найден, создадим строку
оглавления
                let l:resline = printf ("* [[#%s | %s]]",
l:anchor[1], l:title[1])
                call add (l:result, l:resline)
            endif
        endifpmtitlex_02.vim

    endfor

    " Добавим ссылку на комментарии
    call add (l:result, "* [[#comments | Комментарии]]")

    " Получить положение курсора в виде списка:
    " [номер буфера,
    " номер строки,
    " номер столбца,
    " параметр при использовании опции virtualedit]
    let l:cursor = getpos(".")

    " Вставим полученное оглавление в ту строку, где сейчас
стоит курсор
    call append (l:cursor[1], l:result)

    " Удалим все переменные
    unlet l:pattern_anchor l:pattern_title l:count
    unlet! l:resline l:nextline l:title l:anchor l:currline
endfunction
```

Исходник

Содержание функции не изменилось.

Если теперь я скопирую этот скрипт, который назову pmtitles.vim в папку с этой статьей и запущу скрипт с помощью команды

```
:source pmtitles.vim
```

Исходник

Сначала ничего не произойдет, но зато потом можно будет его вызывать сколько угодно раз с помощью команды *:call PmTitles()*, которая будет вставлять оглавление в то место, где стоит курсор.

Пока это удобства в использовании не прибавило, даже наоборот теперь надо предварительно выполнять скрипт, а затем только вызывать функцию. Хотелось бы сделать так, чтобы эта функция всегда загружалась автоматически. А сделать это очень просто, достаточно скопировать файл *pmtitles.vim* в папку *plugin* редактора Vim. Перезапускаем Vim, и теперь у нас всегда под рукой оказывается функция *PmTitles()* и вручную скрипт pmtitles.vim загружать не надо, так как он будет загружаться автоматически при старте. И теперь не надо помнить где лежит этот скрипт или копировать его в папку со статьей.

Но на этом мы не остановимся. Зачем нам писать длинное выражение *:call PmTitles()*, если его можно сократить до *:PmTitles*. Для этого в конец скрипта добавим всего одну строку:

```
command PmTitles call PmTitles()
```

Исходник

Она означает, что когда мы вызываем команду *:PmTitles*, то она будет вызывать *:call PmTitles()*. Все, теперь одной этой командой можно вставлять содержание статьи в нотации pmWiki. И теперь нас можно поздравить с тем, что мы только что создали полноценный плагин для Vim. Здесь лежит архив с самим скриптом и исходником этой статьи в нотации pmWiki, чтобы вы могли проверить как работает плагин.

На этом пока все. В следующей части мы продолжим разбираться с функциями и рассмотрим более "продвинутые" способы их использования.

Часть 6. Продвинутое использование функций

Вы можете подписаться на новости сайта через RSS, Группу Вконтакте или Канал в Telegram.

★★★★★

Рейтинг 5.0/5. Всего 22 голос(а, ов)

○Плохо ○Так себе ○Неплохо ○Хорошо ○Отлично

Голосовать

Михаил 06.07.2009 - 21:31

Спасибо. Жду ещё!

Огромное спасибо. Жду продолжения.

Jenyay 06.07.2009 - 21:44
И Вам спасибо. Продолжение все никак не допишу. Надеюсь выложить его через неделю-другую.

Александр 09.11.2012 - 10:35
Выбрать текст
Как выбрать текст от начала документа до слова в верхнем регистре в минимум 5 букв для последующего удаления? Само это слово и текст после него до конца документа не должно попадать в выделение.

Jenyay 09.11.2012 - 14:48
По памяти не помню команду именно для выделения текста. А вот команда для удаления текста по Вашему описанию выглядит так:

```
:%s/\_.\{-}\(\u\{5,}\)\@=//
```

Подписаться на комментарии

Автор:

Тема:

Ваш комментарий

Введите код
204

Послать