Программирование скриптов для Vim. Часть 2. Переменные

Предыдущие части

Часть 1. Запуск скриптов

Оглавление

В первой части статьи мы разбирали возможные способы запуска скриптов, теперь можно приступить непосредственно к программированию. Когда я разбирался с языком, на котором пишутся скрипты, то меня не покидало ощущение, что этот язык представляет собой какую-то смесь Python и первых реализаций Basic'a, а что-то есть и от Fortran'a (оператор *call*). Конструкции, напоминающие Python, мы будем рассматривать в следующих частях, а здесь поговорим про переменные.

Создание переменных и их типы

Имена переменных, как и во многих других языках программирования, регистрозависимы, то есть переменные с именами *foo*, *Foo* и *FOO* - это три разные переменные. Как и во многих других языках, имена переменных могут состоять из английских букв, цифр и символа подчеркивания. При это имена не могут начинаться с цифр.

Студентам

Фото

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

Контакты

Для создания переменной и присвоения ей значения используется оператор *let* (привет, Basic :)). Вообще, удивляться использованию оператора *let* не стоит, потому что язык скриптов в Vim обязывает в начале строки использовать какой-либо из операторов Vim таких как *let*, *echo*, *call* или какой-нибудь другой.

Как правило, скриптовые языки имеют динамическую типизацию, то есть переменные могут изменять свой тип во время выполнения, а вот в Vim используется немного другой подход. Здесь переменные могут менять свой тип только между родственными типами. То есть, если переменная имеет целочисленный тип, то она его может изменить на тип чисел с плавающей точкой (Float). А вот, например, целочисленная переменная не может стать списком, для этого сначала нужно удалить переменную с помощью оператора *unlet*, а затем с помощью оператора *let* создать переменную с таким же именем, но имеющий тип списка. Интересно, что целочисленная переменная может поменять свой тип на строку без использования оператора *unlet*.

Всего в Vim существует 6 типов:

- Целое число
- Число с плавающей точкой
- Строка
- Список
- Словарь
- Указатель на функцию

Чтобы узнать тип уже созданной переменной, можно воспользоваться функцией *type()*. Она возвращает целое число, обозначающее тип переданной ей в качестве параметра переменной, а именно:

- Целочисленный тип: 0
- Строка: 1
- Указатель на функцию: 2
- Список: 3
- Словарь: 4
- Тип числа с плавающей точкой: 5

Давайте посмотрим как создаются переменные и за одно воспользуемся функцией *type()*. Скопируем следующий скрипт в Vim и запустим его с помощью команды

```
:source %
```

Как уже было сказано в прошлый раз, в статье будут приводиться полные варианты команд, но при частом их использовании удобно пользоваться их сокращенными аналогами, то есть в данном случае:

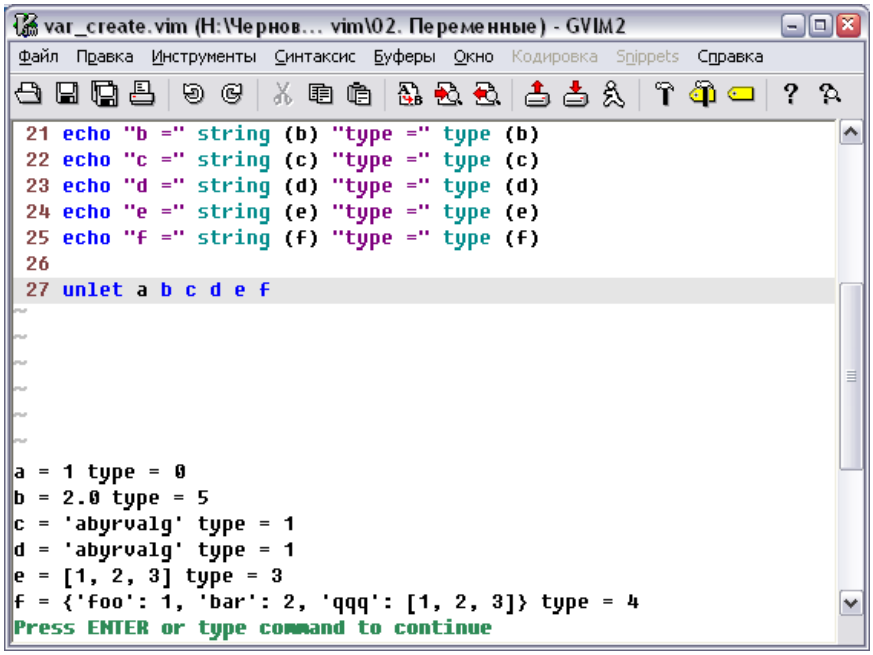
```
:so %
```

Итак, сам пример:

```
" Целое число
let a = 1

" Число с плавающей точкой
let b = a + 1.0

" Строка
let c = "abyrvalg"

" Тоже строка
let d = 'abyrvalg'

" Список
let e = [1, 2, 3]

" Словарь
let f = {"foo": 1, "bar": 2, "qqq": [1, 2, 3]}


echo "a =" string (a) "type =" type (a)
echo "b =" string (b) "type =" type (b)
echo "c =" string (c) "type =" type (c)
echo "d =" string (d) "type =" type (d)
echo "e =" string (e) "type =" type (e)
echo "f =" string (f) "type =" type (f)

unlet a b c d e f
```

Исходник

Результат работы этого скрипта выглядит примерно так:



В этом примере мы создаем 6 переменных различных типов (функции и указатели на них пока трогать не будем). Наверняка вы уже знакомы со всеми этими типами по другим языкам программирования. Обратите внимание на то, что для задания строк можно использовать как одинарные, так и двойные кавычки.

Говорить чем отличается целые числа от дробных и список от словаря, думаю, смысла нет, но для успокоения совести скажу про словари. Словари представляют собой хранилище данных, где каждый хранимый элемент может быть получен по так называемому ключу, другое название словаря - ассоциативный массив. В данном случае ключами являются строки "foo" и "bar", а хранимые значения: 1 и 2. В качестве ключей могут использоваться строки и целые числа (это возможно благодаря тому, что целые числа автоматически могут преобразовываться в строки). В качестве хранимых значений могут использоваться значения любого типа, в том числе один словарь может хранить значения разных типов. Но более подробно про строки, списки и словари давайте поговорим в следующий раз.

Обратите внимание на комментарии. В Vim они начинаются с двойных кавычек (не обязательно как в данном примере в начале строки) и заканчиваются в конце строки. Вы можете возмутиться: "А как же строки? Они же тоже начинаются с двойных кавычек!", но интерпретатор Vim умный и пытается угадать, что вы имели в виду по первой команде в строке программы. В большинстве случаев это ему удается, но бывают и неоднозначные ситуации, например при

использовании команды *echo*. Чтобы начать комментарий после этой команды, нужно использовать символ вертикальной черты (|), а после нее можно вновь открывать комментарий. Например:

```
echo "Tram-pam-pam" | " А это комментарий
```

Исходник

По сути, символ | обозначает обрыв строки, то есть все, что идет за ним Vim будет считать как следующую строку, поэтому комментарий в данном случае с точки зрения Vim будет начинаться на следующей после команды *echo* строки.

Еще, вы наверное обратили внимание, в первом примере мы использовали команду *echo* с несколькими аргументами, разделенных пробелами. Здесь, думаю, все понятно, все строковые значения (или значения, который могут быть автоматически преобразованы в строку) выводятся через пробел.

Здесь же мы воспользовались функцией *string()*, которая преобразует переменную в строку. На самом деле в нашем примере мы могли бы не использовать эту функцию для переменных целого типа и, разумеется, самих строк. Функция *string()* используется везде для наглядности, но преобразовывать, например, дробные числа с помощью этой функции обязательно, сами они, как целые числа, преобразовываться в строку не могут.

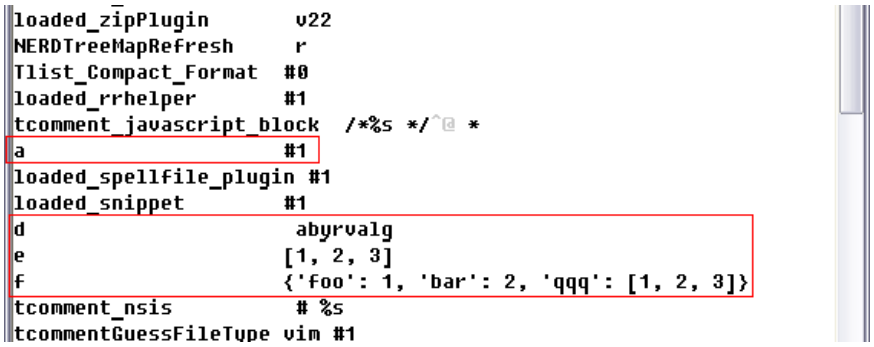
И последнее, что нам осталось рассмотреть на этом примере - оператор *unlet*, который удаляет указанные переменные из памяти. Чтобы удалить созданную переменную, достаточно указать через пробел оператору *unlet* какие именно переменные нам больше не нужны.

Теперь можем провести эксперимент. Закомментируйте последнюю строку с оператором *unlet* и запустите скрипт с помощью команды 'source %'. После этого в командной строке Vim выполните команду:

```
: let
```

Эта команда выведет список всех переменных, которые хранятся в Vim в данный момент времени. Среди огромного

количества переменных вы можете отыскать и наши только что созданные:

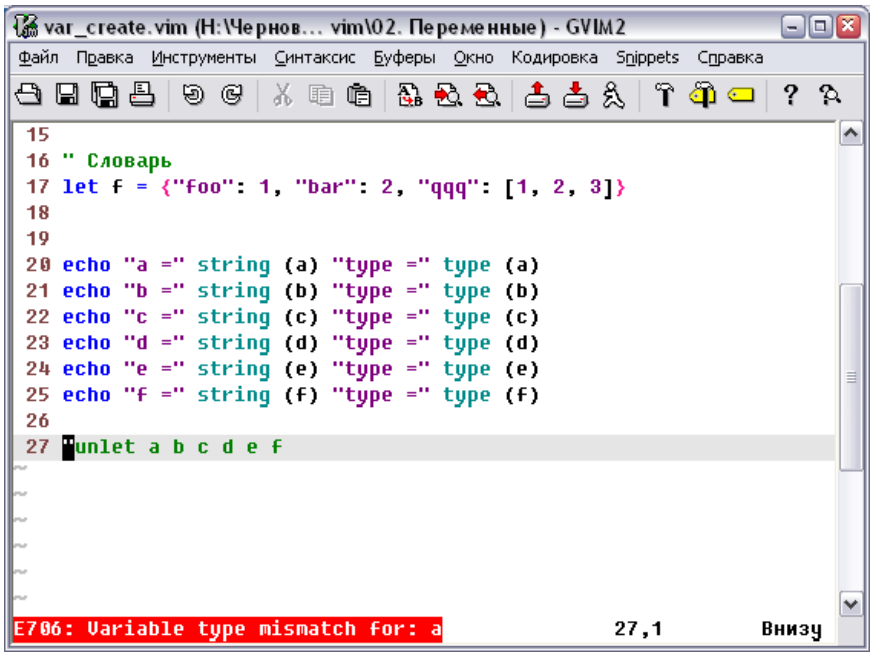


На скриншоте обведены некоторые наши переменные, остальные также можно отыскать в списке. А это значит, что скрипт выполнился, созданные переменные больше не нужны, но они остались храниться в памяти, и из-за этого могут возникнуть неожиданные на первый взгляд проблемы. Например, пусть мы отлаживаем скрипт и нам понадобилось поменять тип переменной. После выполнения предыдущего примера (с закомментаренным оператором *unlet*) попробуем в командной строке выполнить такую, казалось бы безобидную, операцию:

```
: let a = [1, 2, 3]
```

Исходник

И мы получим ошибку:



Она произошла из-за того, что забытая переменная *a* пытается изменить свой тип с целочисленного на тип

списка. Чтобы все-таки изменить ее тип нужно воспользоваться оператором *unlet*:

```
: unlet a
: let a = [1, 2, 3]
```

Исходник

Простейшие операторы

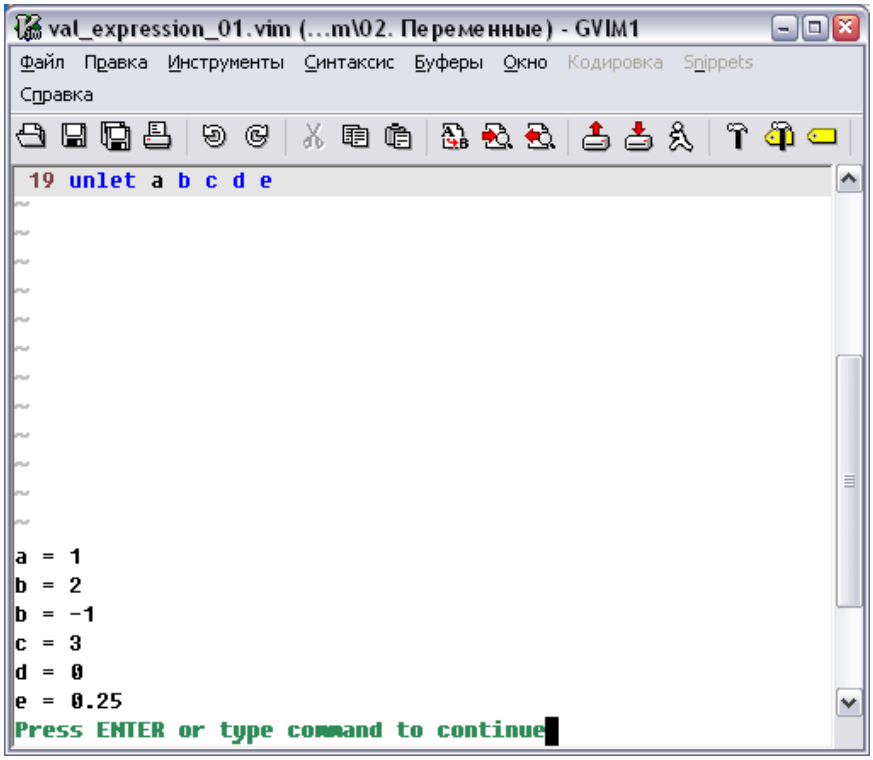
Теперь, когда мы научились создавать переменные, рассмотрим некоторые операции над простейшими типами: целочисленными, числами с плавающей точкой и немного со строками. Для начала простейшие математические операции:

```
let a = 1
echo "a =" a

let b = a + 1
echo "b =" b

let b = a - 2
echo "b =" b

let c = a * 3
echo "c =" c

let d = a / 4
echo "d =" d

let e = a / 4.0
echo "e =" string (e)

unlet a b c d e
```

Исходник

Здесь нет ничего сложного, но хотелось бы обратить внимание на то, что даже если переменная существует, то для присвоения ей другого значения все-равно нужно не забывать использовать оператор *let*. Именно для демонстрации этой особенности переменной *b* значение присваивается два раза - один раз при создании переменной, а другой раз - чуть ниже, через строку.

А это скриншот выполнения скрипта:



Для строк в качестве операции объединения используется точка:

```
let foo = "foo "
let bar = "bar"
let spam = foo . bar
```

Исходник

Также Vim поддерживает операции +=, -= и .=. Первые два оператора действуют так же, как в C/C++ и подобных им языках, то есть соответственно увеличивают или уменьшают переменную на значение, указанное после этих операторов. Оператор .= используется для добавления к концу строки другой строки. При использовании этих операторов также нужно не забывать использовать команду *let*. Операторов *= и /= у Vim нет.

```
let foo = 5
let bar = "Abyrvalg"

let foo += 15
" foo равен 20
echo foo

let bar .= " rulezzz"
" bar равен 'Abyrvalg rulezzz'
echo bar

unlet foo bar spam
```

Исходник

А теперь вспомним, что целые числа могут автоматически преобразовываться в строку, поэтому возможна следующая

ситуация:

```
let foo = "Abyrvalg "
let bar = 123

let spam = foo . bar
echo spam

unlet foo bar spam
```

Исходник

В результате будет выведена строка

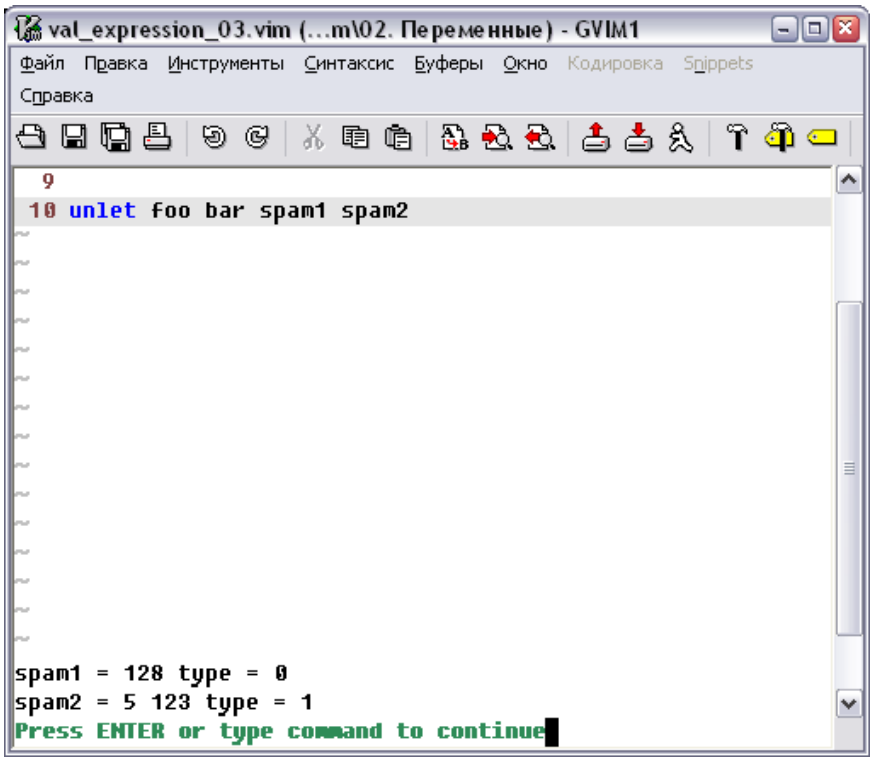
```
Abyrvalg 123
```

Но кроме этого, также и строки могут автоматически преобразовываться в целое число. Рассмотрим следующий пример:

```
let foo = "5 "
let bar = 123

let spam1 = foo + bar
echo "spam1 =" spam1 "type =" type (spam1)

let spam2 = foo . bar
echo "spam2 =" spam2 "type =" type (spam2)

unlet foo bar spam1 spam2
```

Исходник

Результат работы скрипта виден на следующем скриншоте.



Обратите внимание, что значение переменной *foo* оканчивается на пробел, но это не помешало интерпретатору преобразовать строку в целое число. Напомню, что если *type = 0*, значит переменная целочисленного типа, а если *type = 1*, то строкового.

Области видимости переменных

А теперь я вам скажу одну страшную вещь. Готовы? Все переменные, которые мы до этого создавали были глобальными и, соответственно, были доступны из разных скриптов, буферов и закладок. Чтобы в этом убедиться проведем следующий эксперимент. Создадим скрипт, состоящий из одной единственной строки.

```
let foo = 101
```

Исходник

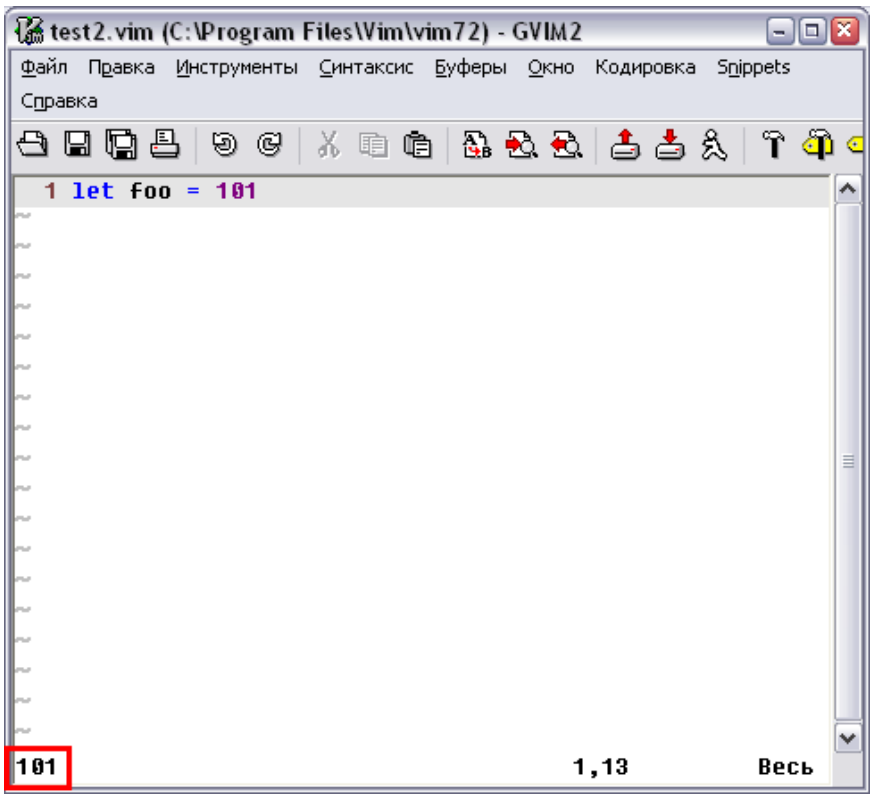
Выполним этот скрипт с помощью команды

```
:source %
```

А теперь, когда скрипт выполнен, в той же командной строке Vim выведем значение этой переменной:

```
:echo foo
```

Vim найдет нашу переменную и выведет ее значение:



То есть наша переменная не только не удалилась по завершению скрипта (мы ведь не использовали оператор *unlet*), но и после выполнения переменная осталась в памяти, и она доступна откуда угодно.

Как вы понимаете, глобальные переменные надо использовать как можно реже, разработчики Vim это тоже понимают, поэтому переменные могут располагаться в различных областях видимости. Их сейчас и рассмотрим.

Пусть мы хотим сделать так, чтобы переменная была видна только внутри скрипта. Для этого достаточно обозначить переменную не просто как *foo*, а как *s:foo*. Изменим предыдущий пример:

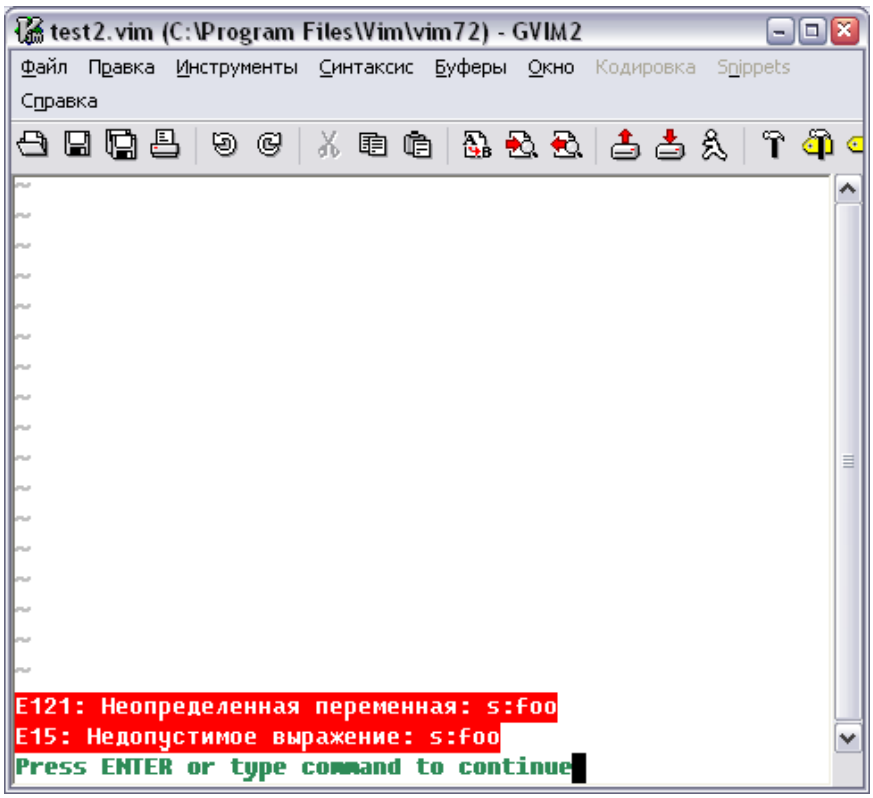
```
let s:foo = 101
```

Исходник

Теперь попытаемся выполнить команду

```
echo s
     foo
```

И мы получим ошибку:



Таким образом, нам удалось создать локальную переменную. Видима она будет только внутри скрипта, где и была объявлена. Можем провести интересный эксперимент. После выполнения предыдущего скрипта, не закрывая Vim, закомментируем первую строку и добавим вторую:

```
"let s:foo = 101
echo s:foo
```

Исходник

Снова выполним этот скрипт с помощью команды *:source %* и увидим, что переменная *s:foo* осталась в памяти и опять доступна из того же скрипта, пусть и измененного. Чтобы убедиться, что переменная *s:foo* доступна только из скрипта, можем открыть другой скрипт в соседнем буфере и из него выполнить команду:

```
echo s:foo
```

Исходник

Мы опять получим ошибку. А вот если теперь в двух соседних буферах откроем один и тот же скрипт, в котором и создавалась переменная, то предыдущий пример будет выполняться в обоих буферах, не зависимо от того, в каком именно переменная была создана, лишь бы это был один и тот же скрипт.

В Vim существуют также и другие области видимости переменных кроме *s:*, они перечислены в следующей таблице:

Обозначение	Область видимости
g:	Глобальные переменные. Использование *g:foo* равносильно записи *foo*
v:	Переменная является глобальной, но предопределенной самим Vim
s:	Переменная является локальной для скрипта
b:	Переменная является локальной для буфера
w:	Переменная является локальной для окна
t:	Переменная является локальной для вкладки (tab)
l:	Переменная является локальной для функции
a:	Переменная является аргументом функции

В разных областях видимости разные переменные могут иметь одинаковые имена.

```
let foo = 101
let s:foo = 202

echo foo
echo s:foo
```

Исходник

Этот скрипт выведет

```
101
202
```

В заключении раздела еще раз хочется напомнить о том, что нужно не забывать удалять переменные, даже локальные, которые больше не понадобятся с помощью команды *unlet*, чтобы они не занимали память.

Практика

Давайте теперь рассмотрим какие-нибудь более жизненные примеры, а за одно будем осваивать встроенные функции Vim.

Для начала напишем скрипт, который будет добавлять к концу открытого файла количество содержащихся в нем строк. Назовем его *addlinescount.vim*.

```
let s:count = line ("$")

let s:line1 = "----------"
let s:line2 = printf ("Lines count: %d", s:count)

call append (s:count, [s:line1, s:line2])

unlet s:count s:line1 s:line2
```

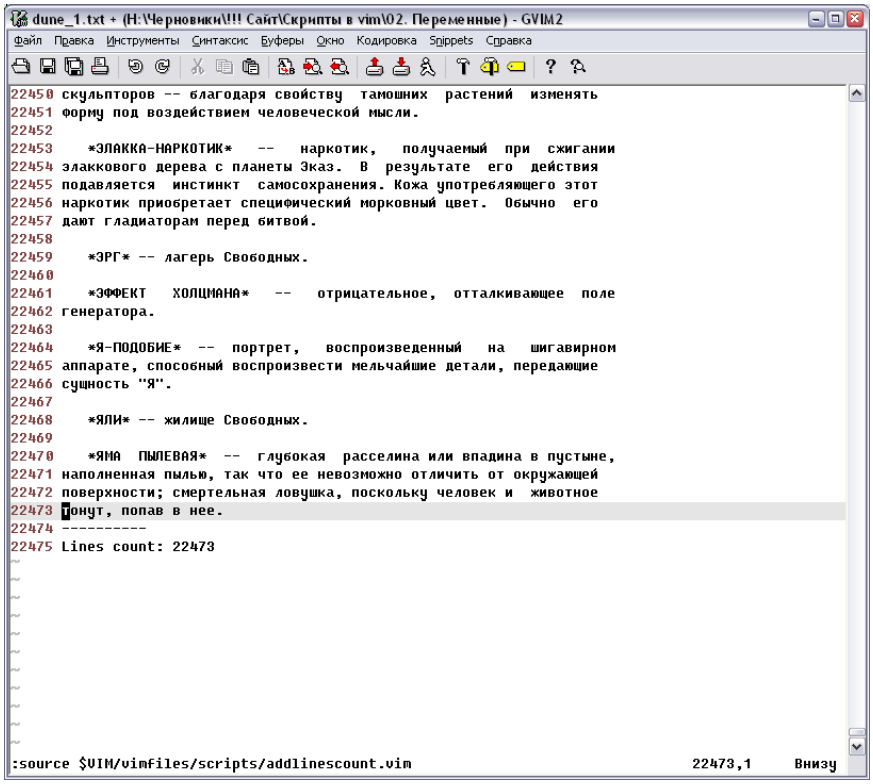
Исходник

Сохраним скрипт куда-нибудь недалеко от самого Vim, чтобы до него можно было бы легко добраться с помощью переменной *$VIM* или *$VIMRUNTIME*. Я для скриптов буду использовать путь *C:\Program Files\Vim\vimfiles\scripts*, куда я смогу добраться с помощью строки *$VIM/vimfiles/scripts*.

Теперь откроем какой-нибудь файл и выполним команду (путь до скрипта у вас может быть свой):

```
:source $VIM/vimfiles/scripts/addlinescount.vim
```

В результате к открытому в буфере к файлу будет добавлено две строки, как, например, на следующем скриншоте:



А теперь рассмотрим построчно скрипт примера.

В начале мы получаем количество строк в текущем буфере с помощью встроенной функции *line()*. В зависимости от строки, переданной в качестве аргумента этой функции, *line()* может выполнять разные задачи. Если ей передать строку "$", как в данном примере, то мы получим количество строк в буфере. Если передать параметр ".", то функция вернет номер строки, на которой находится курсор. Так же с помощью этой функции можно получать номера строк для установленных меток. Подробнее про эту функцию и ее аргументы советую почитать в справке:

```
:help line
```

Многие другие функции используют те же строки-параметры, что и функция *line()*.

После того, как мы узнали количество строк, создаются две строковые переменные, первая из них *s:line1* содержит просто несколько знаков '-', чтобы отделить текст буфера от будущего количества строк, а вторая переменная *s:line2* содержит текст с количеством строк. Как вы поняли, переменные эти видимы только внутри скрипта. Многие из вас, наверное, обрадовались, увидев знакомую ~~с детства~~ по языкам C/C++ функцию *printf()*. Работает она почти точно так же (только функция *printf* здесь возвращает полученную строку, а не выводит ее на экран), поэтому про нее тоже не стоит долго рассказывать, тем более, что про одну эту функцию можно написать целую статью. Тех, кто не знаком с этой замечательной функцией, отправлю в справку, где про нее все подробно написано:

```
:help printf
```

Затем мы используем другую новую для нас функцию Vim - *append()*, которая добавляет строку или строки в текущий буфер. Первый ее параметр определяет номер строки, после которой нам нужно вставить новые строки, а второй аргумент должен быть строкой или списком строк, которые будут добавлены. В нашем примере мы используем список, составленный из только что созданных строк *s:line1* и *s:line2*.

Обратите внимание на команду *call*, которая обозначает, что мы хотим вызвать функцию, которая следует за ним. Как я уже говорил, наличие этого оператора необходимо из-за того, что в начале строки обязательно должен быть какой-нибудь оператор. Так как функция *append()* значение 1 в случае невозможности добавления строк, и 0 при их успешном добавлении, то мы могли бы переписать эту строку следующим образом:

```
let s:failed = append (s:count, [s:line1, s:line2])
```

Исходник

Здесь используется оператор *let*, и необходимость в операторе *call* отпала. В дальнейшем мы могли бы проверить успешность выполнения функции *append()* и вывести при необходимости ошибку, но мы пока этого делать не будем.

В конце скрипта удаляем все наши переменные с помощью оператора unlet.

На этом мы пока прервемся. В следующей части мы будем разбираться со списками.

Часть 3. Работа со списками

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

★★★★★

Рейтинг 4.9/5. Всего 45 голос(а, ов)

○Плохо ○Так себе ○Неплохо ○Хорошо ○Отлично

Голосовать

Подписаться на комментарии

Автор:

Тема:

Ваш комментарий

Введите код
886

Послать

08.04.2009 - 18:41
ДОБРЫЙ ВЕЧЕР, С НЕТЕРПЕНИЕ ЖДУ ПРОДОЛЖЕНИЯ СТАТЬИ "ПРОГРАММИРОВАНИЕ СКРИПТОВ ДЛЯ VIM"

Jenyay 08.04.2009 - 18:43
Я думаю, что продолжение будет к концу следующей недели. Постараюсь пораньше, но не обещаю.