Andrew Scala

Five Minute Vimscript

May 1, 2012

This article is intended so that you can learn the basics of vimscript as quickly as possible. You might even be able to consider this to be a cheat sheet.

You should probably already know how to program before reading this.

Vim's built in documentation is excellent. Try `:h <searchterm>` to read more inside Vim. You can start a REPL inside Vim by issuing the command `gQ` in normal mode if you want to play around with Vimscript.

Note: Examples may contain tokens resembling `<token>`*. These are meant to be replaced completely, including the* `<` *and the* `>`*. Vimscript does use* `<` *and* `>` *as comparison operators.*

Variables

`let` is used to set a variable.

`unlet` is used to unset a variable.

`unlet!` unsets a variable and surpresses the error if it doesn't exist.

By default, a variable is scoped globally if it is initially defined outside a function or it is local to the function it was initialized in in. You can explicitly scope variables by prepending a specific prefix to their name:

```
g:var - global.
a:var - function argument.
l:var - local to function.
b:var - local to buffer.
w:var - local to window.
t:var - local to tab.
v:var - Predefined by Vim.
```

You can set and get environment variables by referencing the variable as `$variable`. Built-in vim options are also available by referencing the variable as `&option`.

Data Types

Number : 32-bit signed number.

```
-123
0x10
0177
```

Float : Floating point number. *Requires* ***+float*** *on vim compile*

```
123.456
1.15e-6
-1.1e3
```

String : NUL terminated string of 8-bit unsigned characters.

```
"ab\txx\"--"
'x-z''a,c'
```

Funcref : A reference to a function. *Variables used for funcref objects must start with a capital letter.*

```
:let Myfunc = function("strlen")
:echo Myfunc('foobar') " Call strlen on 'foobar'.
6
```

List : An ordered sequence of items.

```
:let mylist = [1, 2, ['a', 'b']]
:echo mylist[0]
1
:echo mylist[2][0]
a
:echo mylist[-2]
2
:echo mylist[999]
E684: list index out of range: 999
:echo get(mylist, 999, "THERE IS NO 1000th ELEMENT")
THERE IS NO 1000th ELEMENT
```

Dictionary : An associative, unordered array. Each entry has a key and a value.

```
:let mydict = {'blue': "#0000ff", 'foo': {999: "baz"}}
:echo mydict["blue"]
#0000ff
:echo mydict.foo
{999: "baz"}
:echo mydict.foo.999
baz
:let mydict.blue = "BLUE"
:echo mydict.blue
BLUE
```

There is no **`Boolean`** type. Numeric value 0 is treated as *falsy*, while anything else is *truthy*.

Strings are converted to integers before checking truthiness. Most strings will covert to 0, unless the string starts with a number.

Vimscript is **dynamically** and **weakly** typed.

```
:echo 1 . "foo"
1foo
:echo 1 + "1"
2

:function! TrueFalse(arg)
:   return a:arg? "true" : "false"
:endfunction

:echo TrueFalse("foobar")
false
:echo TrueFalse("1000")
true
:echo TrueFalse("x1000")
false
:echo TrueFalse("1000x")
true
:echo TrueFalse("0")
false
```

String Conditionals and Operators

`<string> == <string>`: String equals.

`<string> != <string>`: String does not equal.

`<string> =~ <pattern>`: String matches pattern.

`<string> !~ <pattern>`: String doesn't match pattern.

`<operator>#`: Additionally match case.

`<operator>?`: Additionally don't match case.

*Note: Vim option **`ignorecase`** sets default case sensitivity for `==` and `!=` operators. Add `?` or `#` to the end of the operator to match based on a case or not.*

`<string> . <string>`: Concatenate two strings.

```
:function! TrueFalse(arg)
:   return a:arg? "true" : "false"
:endfunction

:echo TrueFalse("X start" =~ 'X$')
false
:echo TrueFalse("end X" =~ 'X$')
true
:echo TrueFalse("end x" =~# 'X$')
```

```
false
```

If, For, While, and Try/Catch

```
if <expression>
        ...
elseif <expression>
        ...
else
        ...
endif
```

```
for <var> in <list>
        continue
        break
endfor

for [var1, var2] in [[1, 2], [3, 4]]
        " on 1st loop, var1 = 1 and var2 = 2
        " on 2nd loop, var1 = 3 and var2 = 4
endfor
```

```
while <expression>
endwhile
```

```
try
        ...
catch <pattern (optional)>
        " HIGHLY recommended to catch specific error.
finally
        ...
endtry
```

Functions

Define a function with the `function` keyword. If you want to overwrite a function, use `function!` instead. Functions can be defined in a specific scope, just like variables.

Note: *Functions names must start with a capital letter.*

```
function! <Name>(arg1, arg2, etc)
        <function body>
endfunction
```

`delfunction <function>` deletes a function.

`call <function>` executes a function and is required *unless the call is part of an expression.*

Example: Create global function `Foobar` forcefully (with !). including `...` as the last arg creates variable length arg list. To fetch the first

value from `...`, use `a:1`. To get the second value, use `a:2` and so on. `a:0` is special and contains the number of arguments held in `...`.

```
function! g:Foobar(arg1, arg2, ...)
        let first_argument = a:arg1
        let index = 1
        let variable_arg_1 = a:{index} " same as a:1
        return variable_arg_1
endfunction
```

There's a special way to call functions, and that is on a range of lines from a buffer. Calling a function this way looks like `1,3call Foobar()`. A function called with a range is executed once for every line in the range. In this case, `Foobar` is called three times total.

If you add the keyword `range` after the argument list, the function will only be called once. Two special variables will be available within the scope of the function: `a:firstline` and `a:lastline`. These variables contain the start and end line numbers for the range on the function call.

Example: Create buffer function `RangeSize` forcefully which will print out the size of the range it is called with.

```
function! b:RangeSize() range
    echo a:lastline - a:firstline
endfunction
```

Classes

Vim doesn't have classes built in, but you can get rudimentary class-like functionality by leveraging the `Dictionary` object. In order to define a method on a class, use the `dict` keyword in the function definition to expose the internal dictionary as `self`.

```
let MyClass = {foo: "Foo"}
function MyClass.printFoo() dict
        echo self.foo
endfunction
```

Classes are like singletons by default. In order to make instances of classes in vimscript, call `copy()` on the dictionary.

```
:let myinstance = copy(MyClass)
:call myinstance.printFoo()
Foo
:let myinstance.foo = "Bar"
:call myinstance.printFoo()
```

```
Bar
```

Keep Going

Now that you know what you're getting yourself into, there are some good resources for learning more:

Tutorials:

Vim's internal documentation

IBM's "Scripting the Vim Editor"

Other people's code:

Github: tpope

Github: scrooloose

Thanks

Hope you found this useful. Thanks for reading.

If you have any suggestions or corrections, be sure to give me a shout!

email twitter post archive