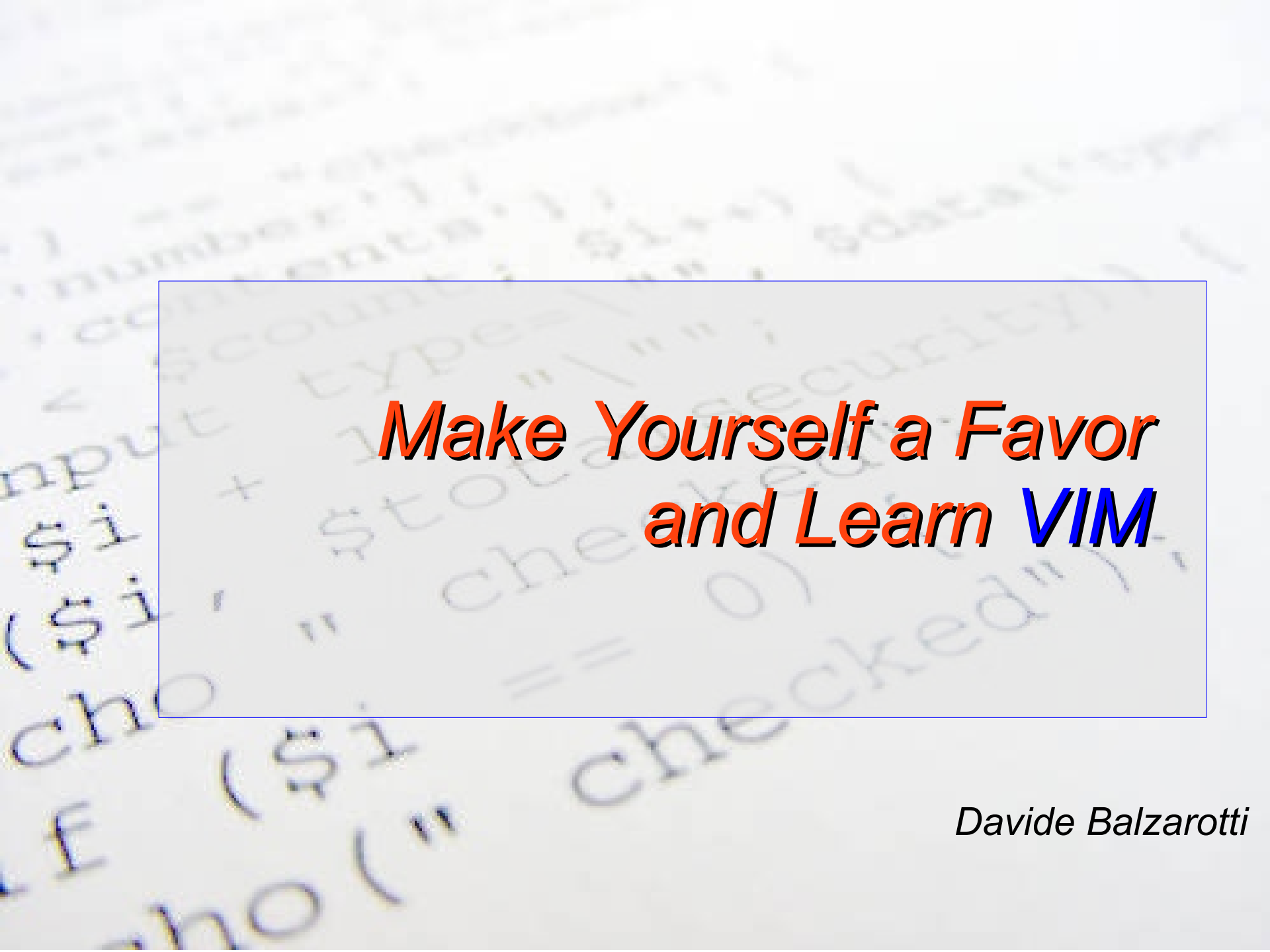

Make Yourself a Favor
and Learn VIM
Davide Balzarotti

Vim is a beautiful tool.
Unfortunately, it is about as user-friendly as a radioactive crocodile.

Why Mastering an Editor?

- In a world of text streams, the text editor is your home
- Computer scientists spend a lot of time editing files, writing documents, writing source code, answering emails...
  - Choose a good editor
    (no, they are not all the same)
  - Invest time to learn how to use it
    (really, the better you master its cryptic commands, the more productive you will be)
  - Use it all the time
    (using different editors for different tasks is not usually a good idea)

I Said an Editor.. not an IDE

- *Integrated Development Enviroments* are pre-packaged toolboxes that includes many tools needed for a specific job (usually writing code)
  - Good to handle very large code bases
  - Bad for typing text

- The Unix philosophy
  - Write tools that do one thing (in our case “edit text”) and do it well
  - The shell is your IDE and Unix is your toolbox
  - Integration can be achieved by configuration

I Said an Editor.. not a Word Processor

- Preparing a document involves two *separate* tasks:
  - Composition – preparing the text content
  - Typesetting – preparing the layout: fonts, colors, alignment, section headings, one or two columns, …

I Said an Editor.. not a Word Processor

- Preparing a document involves two *separate* tasks:
  - Composition – preparing the text content
  - Typesetting – preparing the layout: fonts, colors, alignment, section headings, one or two columns, …
- Word processors are WYSIWYG ("What You See Is What You Get") tools that combine *composition* and *typesetting*
- Should I mention again the Unix Philosophy ?
  - Use a text editor to write text
  - Use a specialized markup/language for typesetting (e.g., LaTeX)

Why should I use a text editor that is so difficult?
After one year I still don't know what I am doing

Why should I use a text editor that is so difficult? After one year I still don't know what I am doing

The same reason why you use a violin to play music instead of a kazoo

VS

*from a discussion on reddit/r/vim

Writing Text

- Text *Composition* involves two aspects:
  - Creative – in which the author *produces* words and sentences
  - Operational – in which the author *manipulates* the text and gives command to the editor
    (e.g., to save files, search and replace a word, …)

Writing Text

- Text *Composition* involves two aspects:
  - Creative – in which the author *produces* words and sentences
  - Operational – in which the author *manipulates* the text and gives command to the editor
    (e.g., to save files, search and replace a word, …)

- Most of the editors combine the two aspects together:
  - You write text by typing letters, numbers, and symbols
  - You give commands using:
    - Mouse + Menus
    - Special keys (e.g. F1-F12)
    - Shortcuts based on modifier keys (ALT, CTRL, Meta, ...)

VIM is a Modal Editor

- Modal editors keep the *creative* and *operational* tasks separated
  - One mode is dedicated to type text, another to manipulate it
  - The same key performs different actions depending on the currently active mode

type the word "`cat`" in insert mode and
Change Around a Tag in normal mode

The Lesson of VI

“ Vi is fundamentally built on command composability.
It favors small, general-purpose commands that can be combined with objects to compose larger commands ”

- Mike Kozlowski

History

- vi was originally developed by Bill Joy in 1976 as part of the BSD Unix distribution
  - Originally a visual extension of a line editor named ex
  - Developed on an ADM 3A terminal, to be usable over a 300 bits-per-second modem line
- vim was developed by Bram Moolenaar in 1988 for the Amiga operating system
  - Originally it stood for "Vi Imitation" but quickly became synonym of "Vi iMproved"

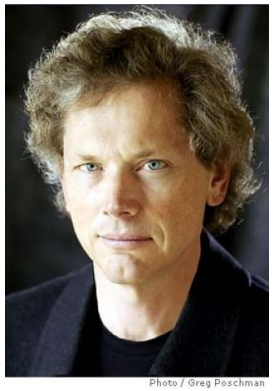
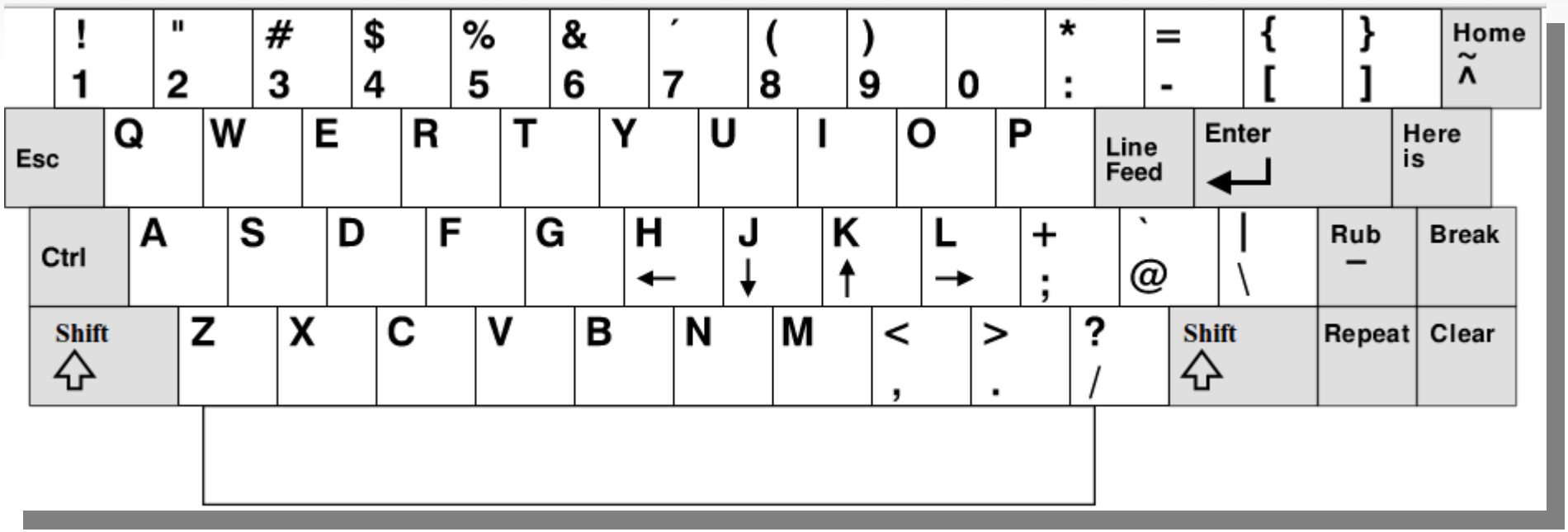
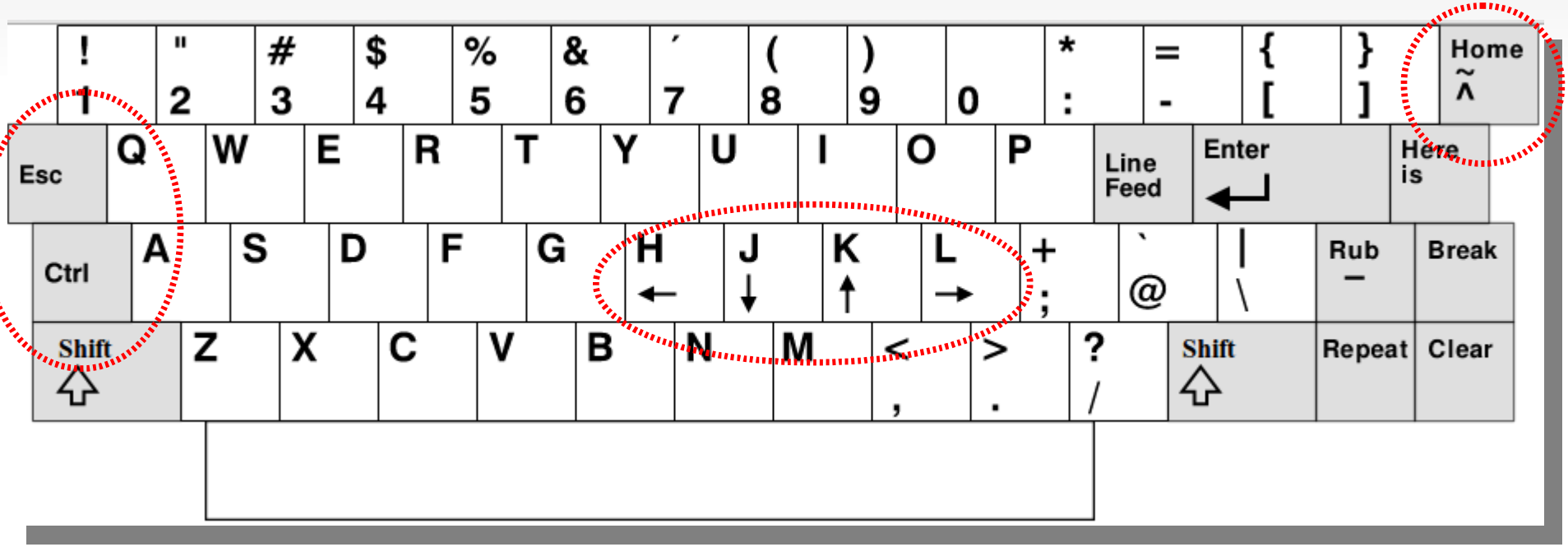


Photo / Greg Poschman

An Important Historical Aspect



An Important Historical Aspect



> *CAPS LOCK is the second biggest key on the home row.. and it is practically useless!! Switching it with ESC will make your life much easier (and not only in VIM)*

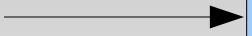
“

CAPS LOCK is the second biggest key on the home row.. and it is practically useless!! Switching it with ESC will make your life much easier (and not only in VIM)

”

```
> cat ~/.xmodmap
remove Lock   = Caps_Lock
keycode 0x09 = Caps_Lock
keycode 0x42 = Escape
add    Lock   = Caps_Lock
> xmodmap ~/.xmodmap
```

Changing Mode



Normal Mode

where you move around, delete, cut&paste,
and do most of the *operational* editing

Insert Mode

where you type your text
and do most of the *creative* editing

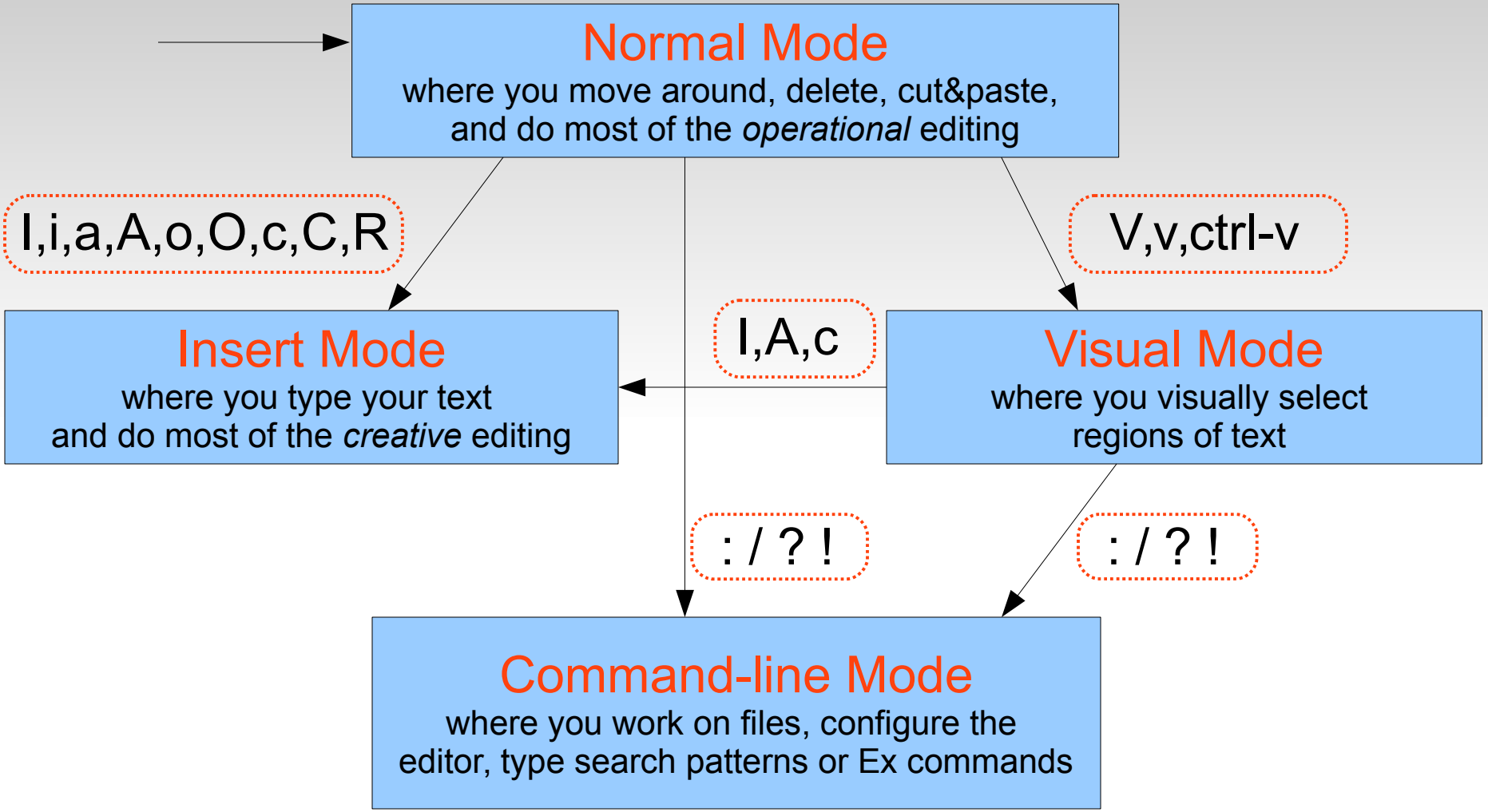
Visual Mode

where you visually select
regions of text

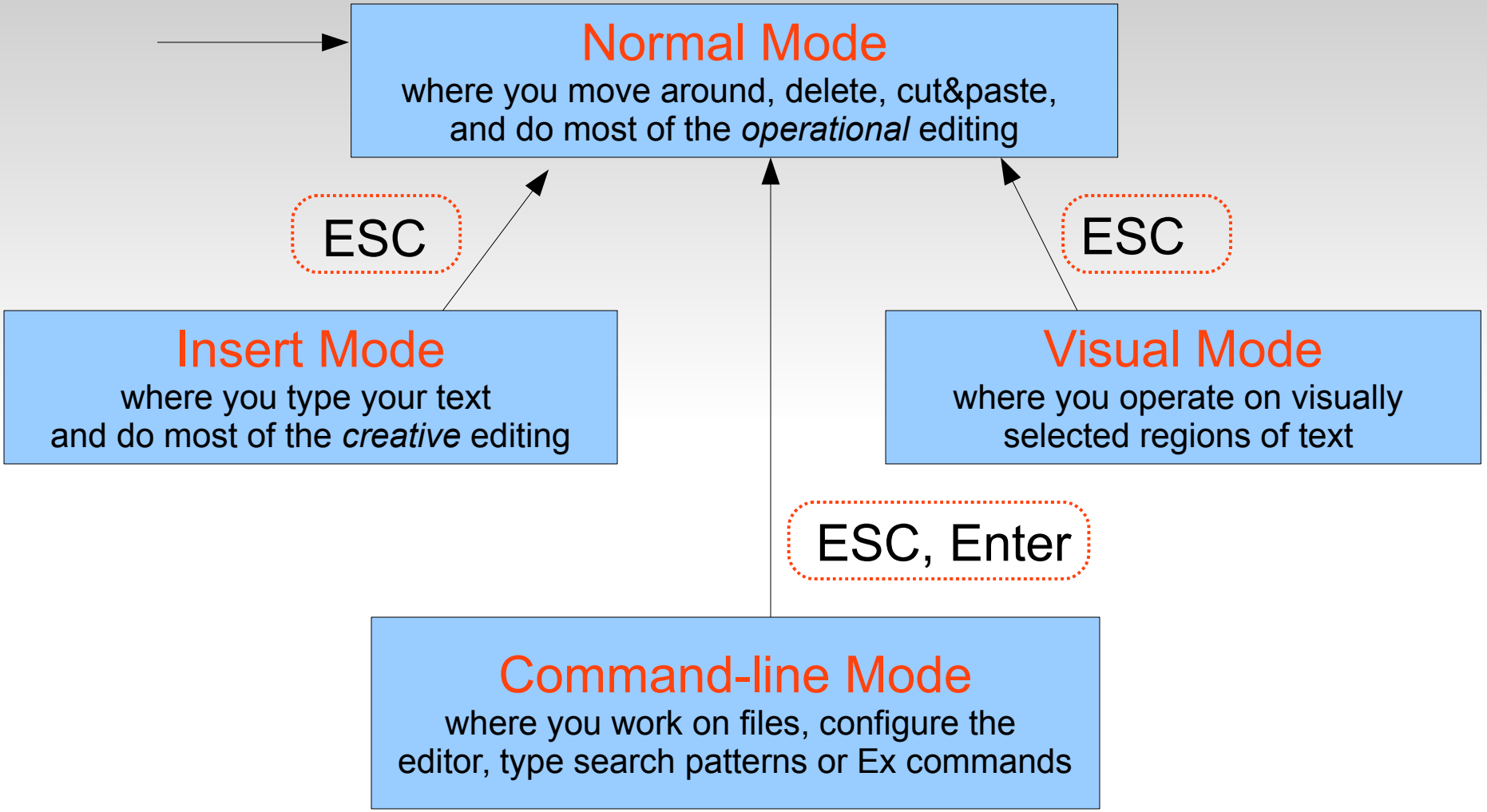
Command-line Mode

where you work on files, configure the
editor, type search patterns or Ex commands

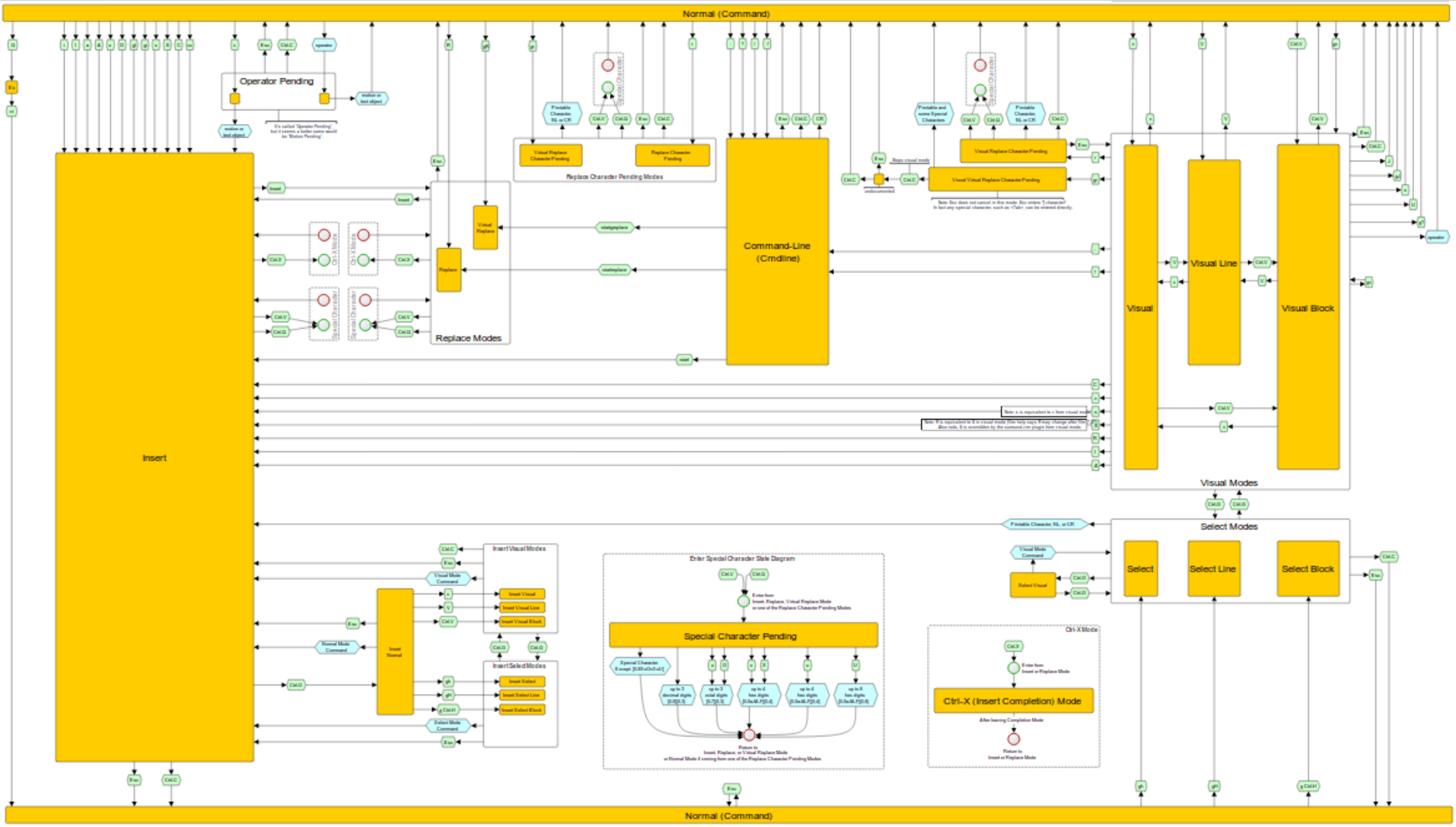
Changing Mode



Changing Mode



The real picture



> *Stay in insert mode only while typing.*
> *When you pause to think,*
> *move back to normal mode*

Vim Help and Documentation

- You can access an inline help page by typing

  `:help <subject>`

- The Vim documentation consists of two parts:
  - The User Manual: task oriented explanations, from simple to complex. It reads from start to end like a book.

    `:help usr_N.txt` N=01,02,03...99

    `:help usr_toc.txt`
  - The Reference Manual: precise description of how everything in Vim works.

    `:help motion.txt`

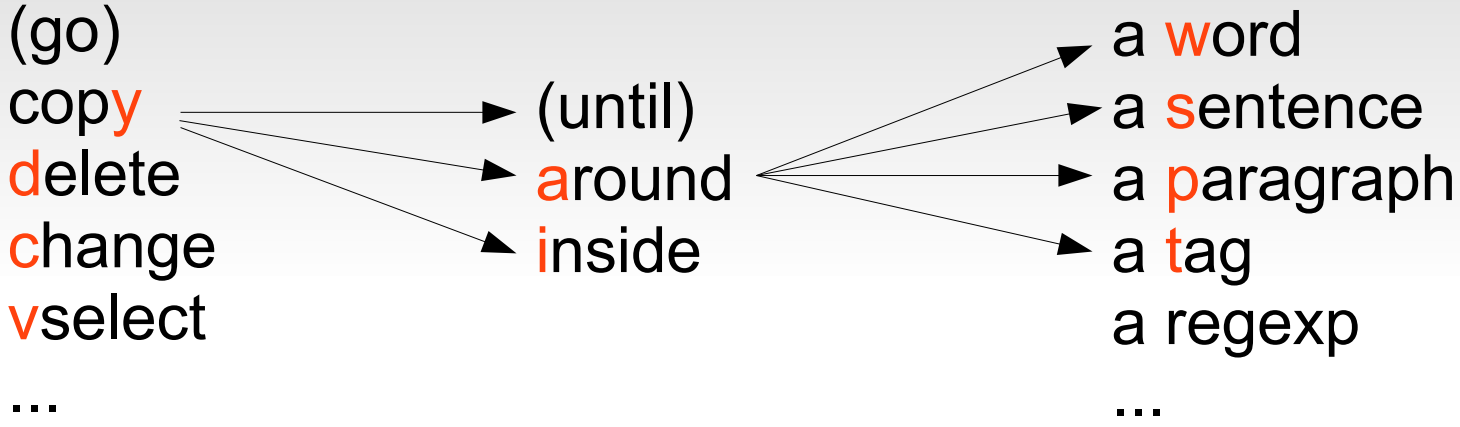
    `:help reference_toc`

- Vim help/documentation is a sort of hypertext with links you can follow between each part

Command Mode

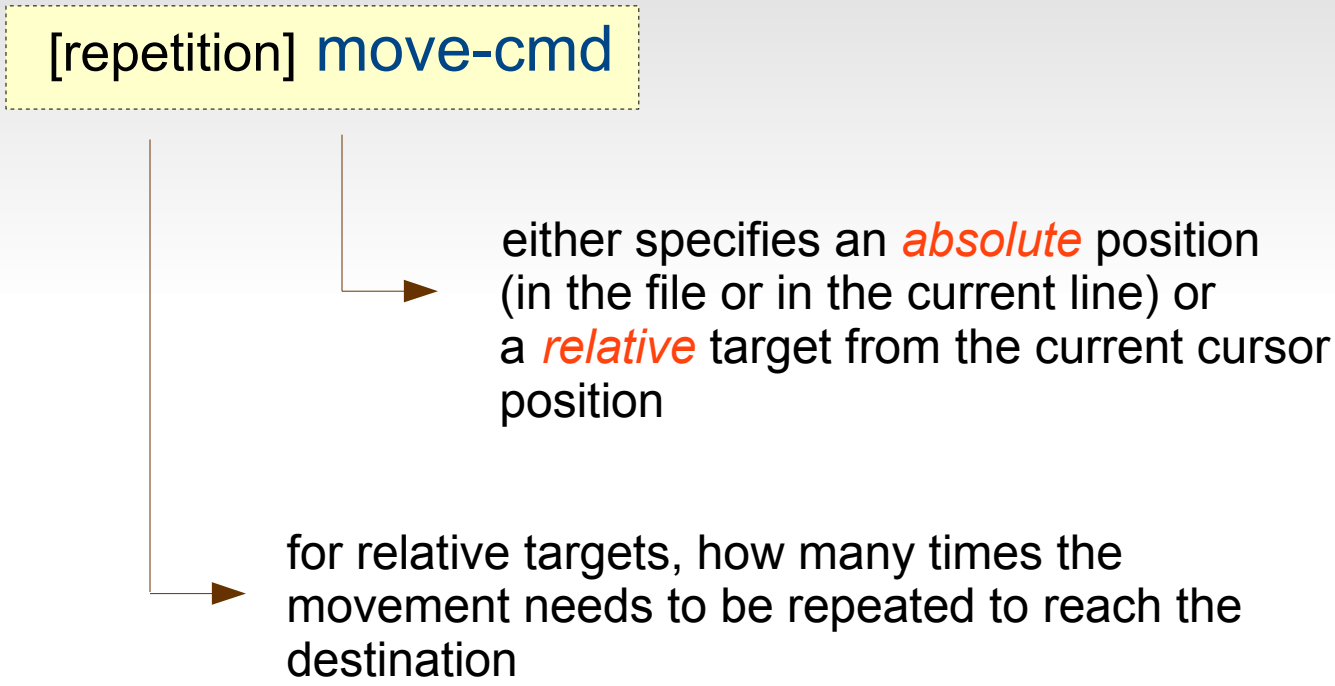
Command Mode

Vim commands are based on a composable grammar



Command Mode

Movement commands:



`:help motion.txt`

Absolute, File-Based

gg	first line
G	last line
nG	n line
n|	n column

Absolute, Line-Based

^	first non-blank character
g_	last non-blank character
0	first character
$	last character

Relative, Line-Based

fc Fc	next (previous) occurrence of character c
tc Tc	before the next (previous) occurrence of character c
; ,	repeat the last tf or the last TF command

Relative, Text-Based

← → ↑ ↓	move one character
h l j k	same as above
w b	beginning of the next (previous) word
e ge	end of the next (previous) word
W N E gE	like above but use space-separated words
( )	previous (next) sentence
{ }	previous (next) paragraph
%	match of next brace, bracket, or comment
* #	next (previous) occurrence of the word under the cursor
[( ])	previous (next) unmatched parenthesis
[{]}	previous (next) unmatched curly bracket

Absolute, File-Based

gg	first line
G	last line
nG	n line
n|	n column

Absolute, Line-Based

^	first non-blank character
g_	last non-blank character
0	haracter
$	

Use `:set iskeyword` to configure what is a word for you.

Relative, Text-Based

← → ↑ ↓	move one character
h l j k	same as above
w b	beginning of the next (previous) word
e ge	end of the next (previous) word
W N E gE	like above but use space-separated words
( )	previous (next) sentence
	ragraph
	ce, ent[1]
	word
	under the cursor

Rel

fc Fc	occurrence of character c
tc Tc	before the next (previous) occurrence of character c
; ,	repeat the last tf or the last TF command
[( ])	previous (next) unmatched parenthesis
[{]}	previous (next) unmatched curly bracket

[1] cannot be preceded by a counter

> You should not try to learn every command an editor offers. That would be a complete waste of time. Most people only need to learn 10 to 20% of the commands for their work.
> But it's a different set of commands for everybody

- *Bram Moolenar*

Marks

- Allow the user to record the current cursor position
  - There is no visible indication of where marks are set
- Four types of marks:
  - Local marks (letters a-z) are unique to each file
  - Global marks (letters A-Z) identify a location in a particular file (so jumping to the mark also open that file)
  - Special marks, managed automatically by VIM
    - `.` : position of the last change
    - `` ` `` : position before the last *jump* (`'G/(){}n)
    - `^` : position where the cursor last exited insert mode
    - `< >` : start and end positions of the last selected text
  - Change marks, automatically generated every time a piece of text is modified. These marks have no name

Marks

Marks-related Commands	
mx	record the current position in mark x
`x	jump to mark x
'x	jump to the first char of the line containing mark x
:marks	list the values of all the marks
:changes	list the values of all the changes
g, g;	jump to the next (previous) change mark

Command Mode

- Move commands

  [repetition] move-cmd

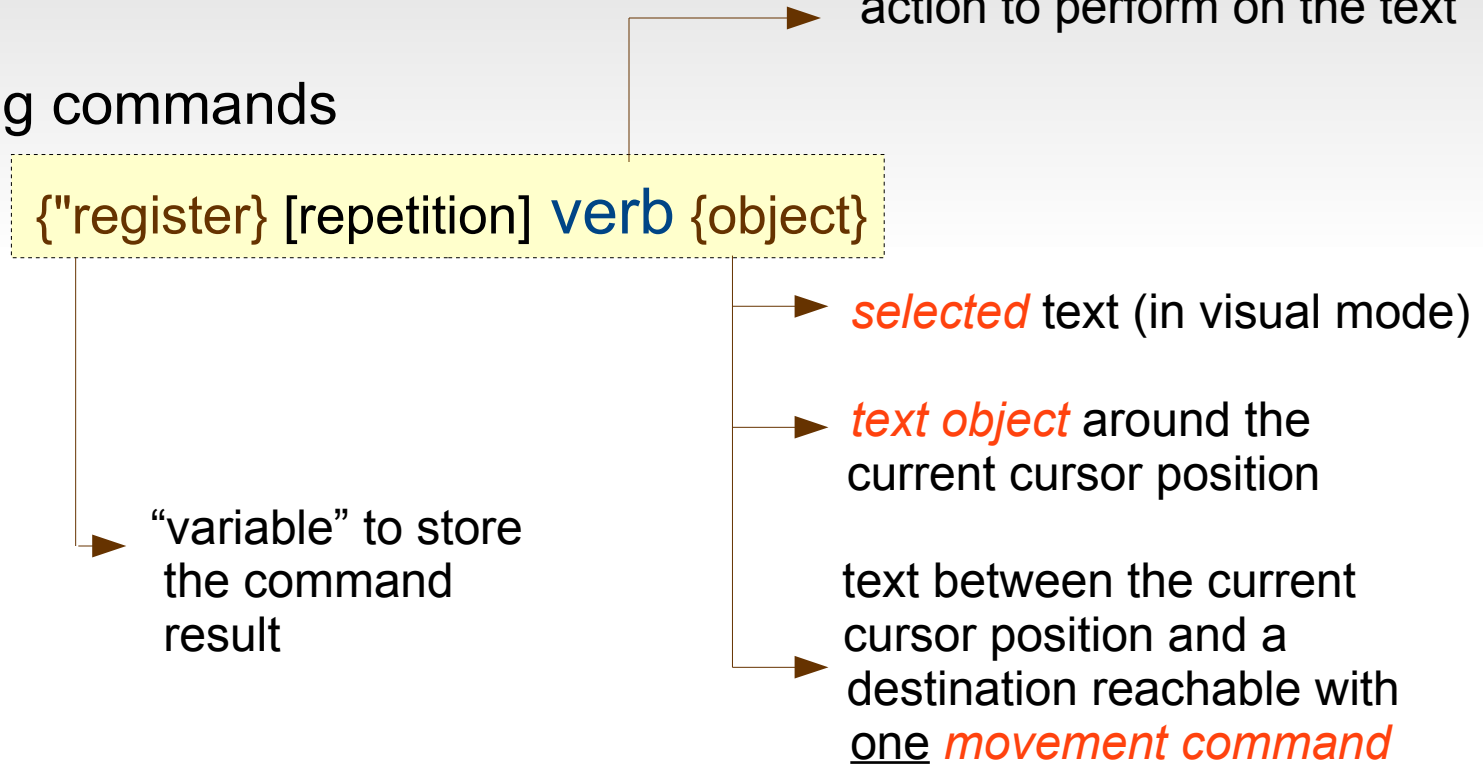
Command Mode

- Move commands

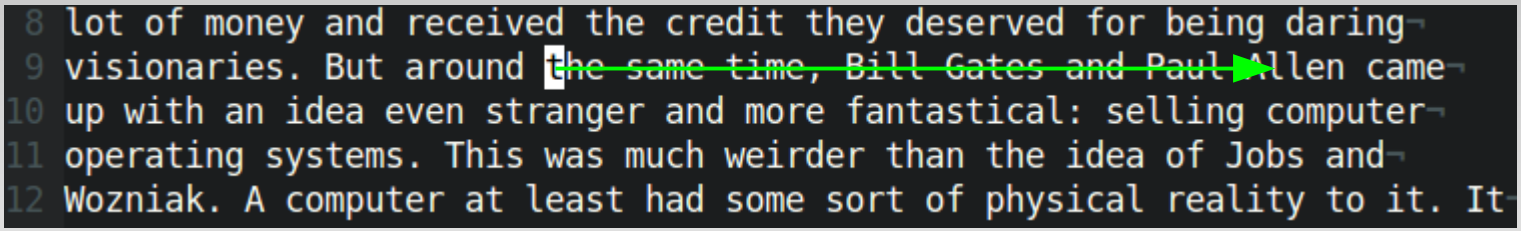
[repetition] move-cmd

- Editing commands

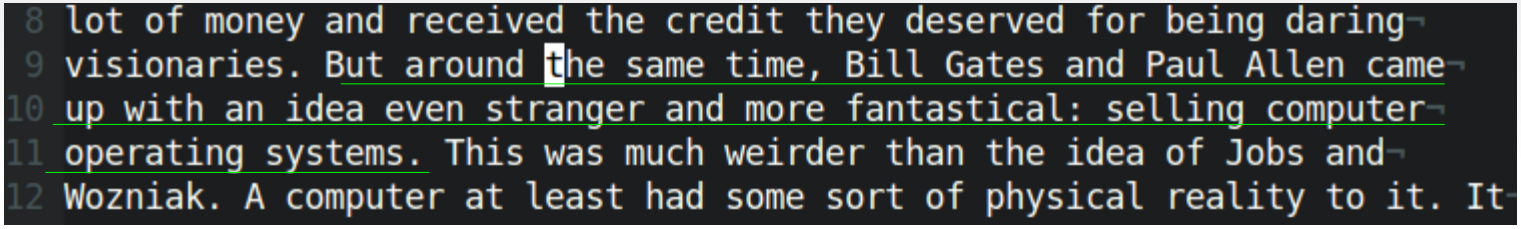
{"register} [repetition] verb {object}



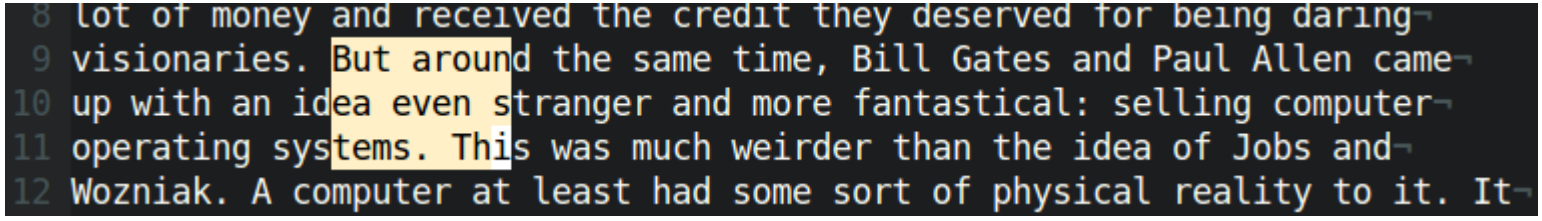
From the current position to a destination reachable through a movement command:



On the text around the current position (text objects):

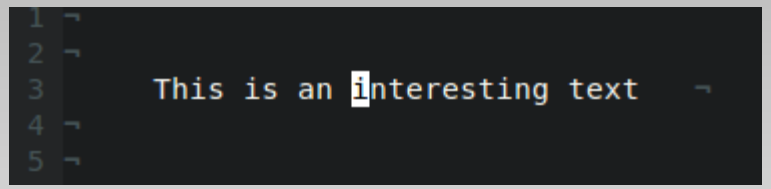


On the visually selected text (more about it later):

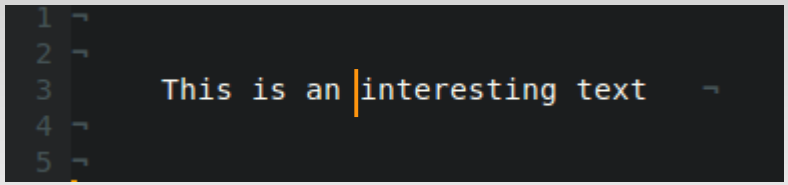


Learning some Verbs

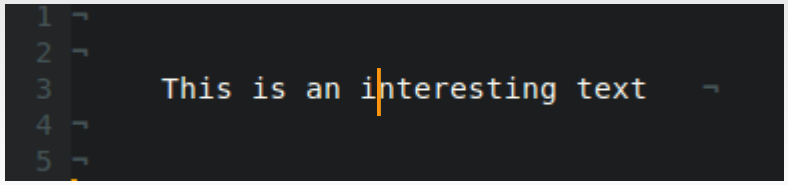
Verbs that enter insert mode	
i a	insert before (after) the cursor
I	insert before the first non-blank character at the beginning of the line
A	appending at the end of the line
gI	insert at the beginning of the line
gi	Insert where you left insert mode last time
o O	Insert text in a new line below (above) the current one
R	insert text in overwrite mode
co	change text of object o
cc	change the current line
C	change till the end of the line


This is an interesting text

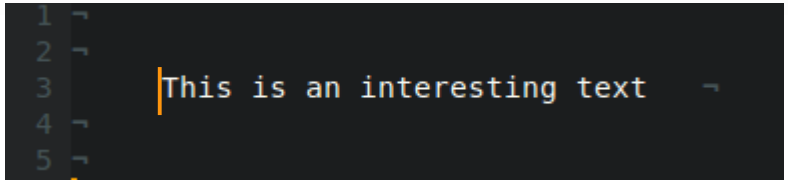

i


This is an interesting text

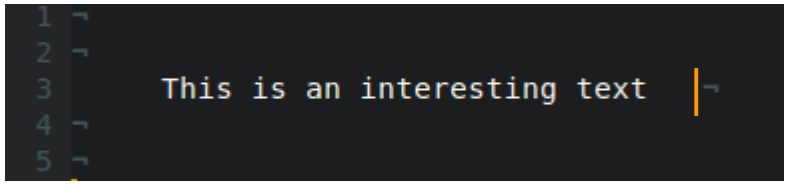

a


This is an interesting text


I


This is an interesting text


A


This is an interesting text

Cut & Paste

General commands	
do	Cut the object o
yo	Yank (copy) the object o
p P	Paste after (or before) the cursor
Linewise versions	
dd	cut current line
yy	Copy current line
D	Delete till the end of the line

For compatibility issues, Y is a synonym of yy
If you want it to yank till the end of the line (more intuitive) you can redefine it with:
:map Y y$

Few More Verbs

guo gUo	Make lowercase or uppercase
gqo	Format text
<o >o	Indent left and right
J	Join the current line with the next one
~	Swap case of a single letter

Text Objects

modifier object

- Modifiers
  - i – inside
  - a – around
- Object
  - w W – word or Word
  - s – sentences
  - p – paragraphs
  - t – Tagged blocks
  - [] () {} <> – Blocks delimited by these characters
  - “ ' ` – Strings delimited by these characters

```
1 What would the engineer say, after you ¬
2 had explained your problem,¬
3 and enumerated all of the dissatisfactions in your life? He would¬
4 probably tell you that life is a very hard and complicated thing;¬
5 that ¬
6 no interface can change that; that anyone ¬
7 who believes otherwise¬
8 is a sucker; and that if you don't like having choices made for you,¬
9 you should start making your own. ¬
```


```
1 What would the engineer say, after you had explained your problem, and¬
2 enumerated all of the dissatisfactions in your life? He would probably tell¬
3 you that life is a very hard and complicated thing; that no interface can¬
4 change that; that anyone who believes otherwise is a sucker; and that if¬
5 you don't like having choices made for you, you should start making your¬
6 own.¬
```

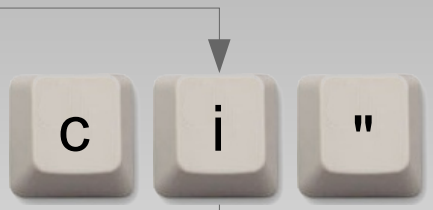
```
What would the engineer say, after you ¬
had explained your problem,¬
and enumerated all of the dissatisfactions in your life? He would¬
probably tell you that life is a very hard and complicated thing;¬
that ¬
no interface can change that; that anyone ¬
who believes otherwise¬
is a sucker; and that if you don't like having choices made for you,¬
d start making your own. ¬
```

This operation is so common when you write text that I believe it deserves its own key:

:map Q gqip

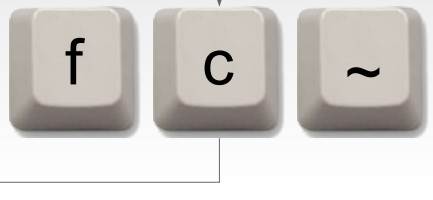
```
W
e                                                              ll¬
y                                                              ¬
change that; that anyone who believes otherwise is a sucker; and that if¬
you don't like having choices made for you, you should start making your¬
own.¬
```

```
1  ¬
2  weird_function("this is the first parameter", 2, 55);¬
3 ¬
4 ¬
```




```
1  ¬
2  weird_function("", 2, 55);¬
3 ¬
4 ¬
```

```
2 ¬
3 In the Beginning was the command Line¬
4 ¬
5 ¬
```




```
2 ¬
3 In the Beginning was the Command Line¬
4 ¬
5 ¬
```

```
1 ¬
2 Some random text (now a comment¬
3 about f(x) and g(x)) and more text.¬
4 ¬
```


```
1 ¬
2 Some random text and more text.¬
3 ¬
```

Registers

- Sort of global variables that can contain text
- VIM has in total 48 registers (some read/write, some read-only)
  - Unnamed register (") is the default target for most of the commands, including cut (d), copy (y), and paste (p) operations
  - Last inserted text (.)
  - Register zero (0) contains the last yanked text
  - 26 named registers (a-z)
    - Using the uppercase version (A-Z) append to a register instead of replacing its content
  - Last search register (/)
  - Blackhole register (_) is like `/dev/null`
  - System Clipboard (+) and mouse selection (*)

Playing with Registers

"r<cmd>	Use register r as a destination for <cmd>
<ctrl-r>r	in *insert mode*, insert the content of register r
:let @r=”...”	manually set a register value in command-line mode
:reg [r]	list all (or one im particular) registers values

Unlimited Undo/Redo

- One undo command normally undoes a typed command, no matter how many changes that command makes
  - This sequence of undo-able changes forms an undo block
  - When you enter insert mode you start a new block that ends when you move back to normal mode (or when you press an arrow key!)

.	Repeats, at your current location, the last edit command you executed in normal mode
u	Undo the last change (use multiple times to undo many changes)
<ctrl-r>	Re-do changes that were previously undone
:earlier {N}s :earlier {N}m :earlier {N}h :earlier {N}f	Move back to the state {N} seconds, minutes, hours, or file writes ago
:later {N}s ...	Move forward to the state {N} seconds, minutes, hours, or file writes in the future

```
1 <a href="fooo.html">First link</a>¬
2 <a href="foo2.html">Second link</a>¬
3 <a href="foo3.html">third link</a>¬
```

```
1 <a href="fooo.html">FIRST LINK</a>¬
2 <a href="foo2.html">Second link</a>¬
3 <a href="foo3.html">third link</a>¬
```

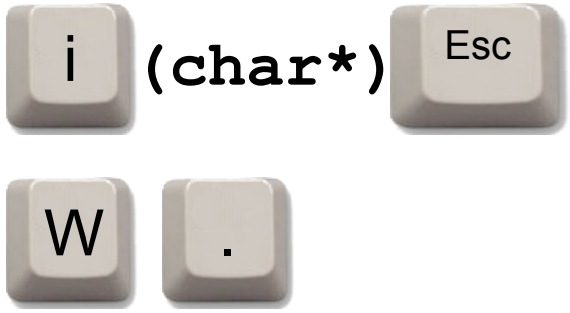
```
1 <a href="fooo.html">FIRST LINK</a>¬
2 <a href="foo2.html">SECOND LINK</a>¬
3 <a href="foo3.html">third link</a>¬
```

```
1 <a href="fooo.html">FIRST LINK</a>¬
2 <a href="foo2.html">SECOND LINK</a>¬
3 <a href="foo3.html">THIRD LINK</a>¬
```

```
2 ¬
3     res = some_function( a, b);¬
4 ¬
```

```
2 ¬
3     res = some_function( (char*)a, b);¬
4 ¬
```

```
2 ¬
3     res = some_function( (char*)a, (char*)b);¬
4 ¬
```



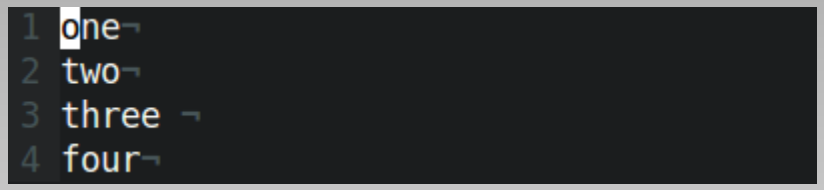
Macros

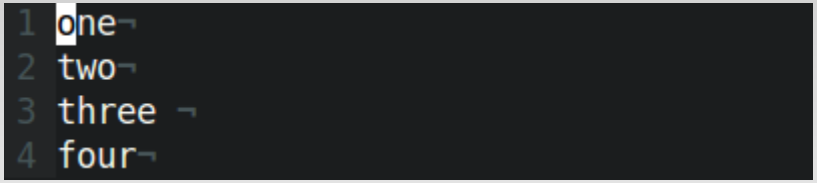
- The . command is great to repeat a single command
- For more, you need to record a macro (a sequence of commands) in a register

  Record: q<register>cmd_1 cmd_2 …. cmd_n q

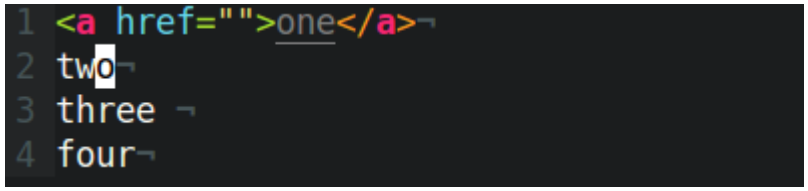
  Play: [counter]@<register>

- Excellent to:
  - Repeat repetitive operations that involve multiple commands
  - Repeat one or more commands in many different places
    (record the command + the motion to move to the next place)

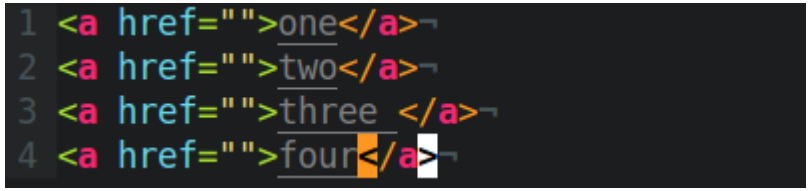




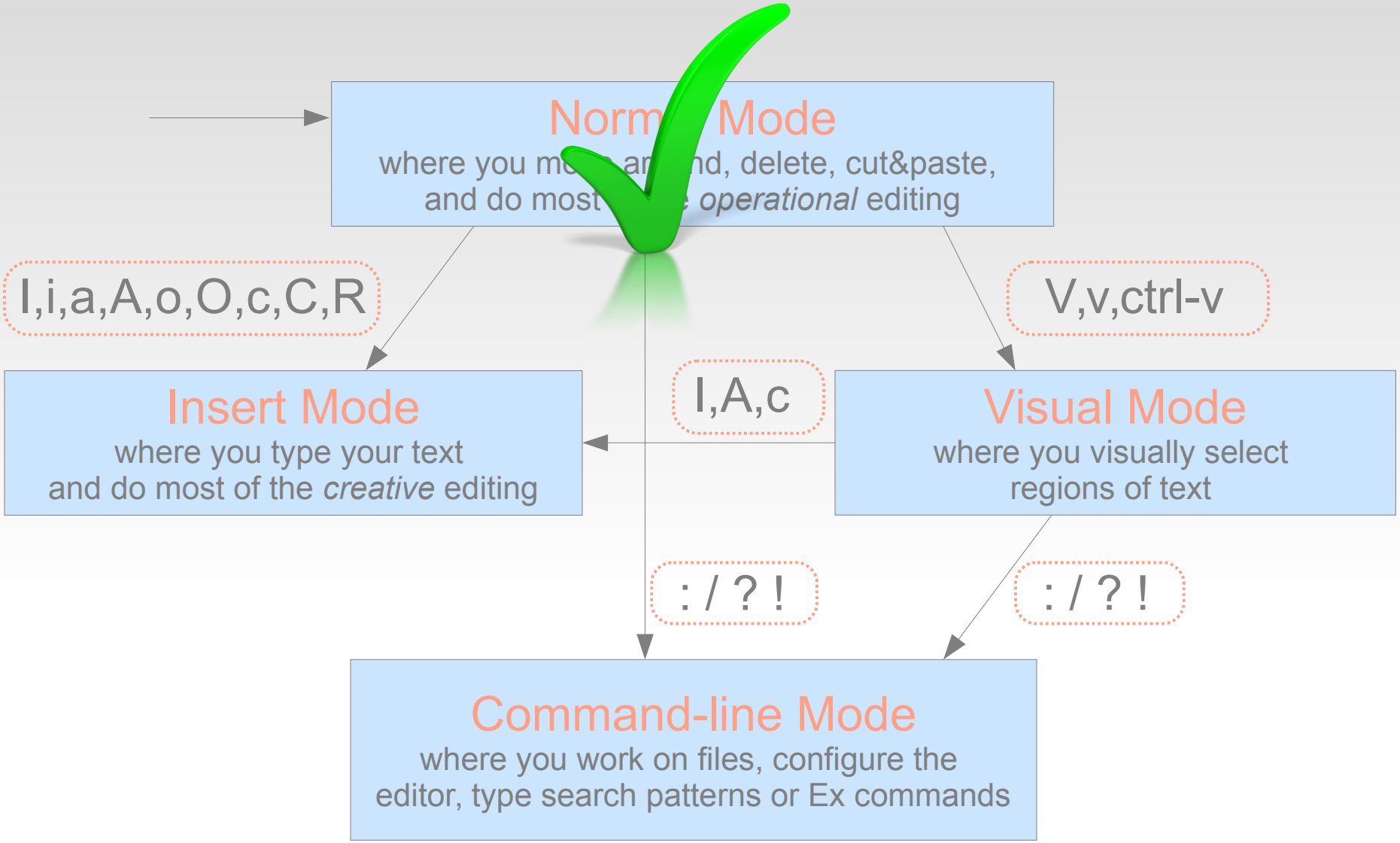
Start recording a macro in the register w



Edit the first line, then press j (to move to the next one) and q (to terminate the macro recording)



Execute the macro 3 times


Normal Mode
where you move around, delete, cut&paste,
and do most operational editing
I,i,a,A,o,O,c,C,R
V,v,ctrl-v
Insert Mode
where you type your text
and do most of the creative editing
I,A,c
Visual Mode
where you visually select
regions of text
: / ? !
: / ? !
Command-line Mode
where you work on files, configure the
editor, type search patterns or Ex commands

Visual Mode

Visual Mode

- Visually select regions of text
  - Pro: sometimes easier than remembering obscure ranges or motion commands
  - Cons: does not work well with the dot . command
- Enter visual mode

  V – linewise selection

  v – characterwise selection

  CTRL-V – rectangular (or block) selection

  gv – select the last visually selected area
- You can use any vim movement commands to move the selection corner
  - o and O let you cycle through the different corners

In line- and character-visual mode	
rc	Replace each character of the selected area with c
s	Delete the selected text and enter insert mode
p	Replace the selected area with the clipboard
J	Join all the selected lines
I	Insert before the selection
A	Insert after the selection
g<CTRL-g>	Show some statistics about byte/word/line counts
Special behaviors in block mode	
I	Insert some text at the start of the block in *each selected line*
A	Insert some text at the end of the block in *each selected line*
s	As in other visual mode, but the text is copied in each line of the selection
Special configurations	
:set virtualedit	Control if (and when) the cursor can be positioned where there is no text

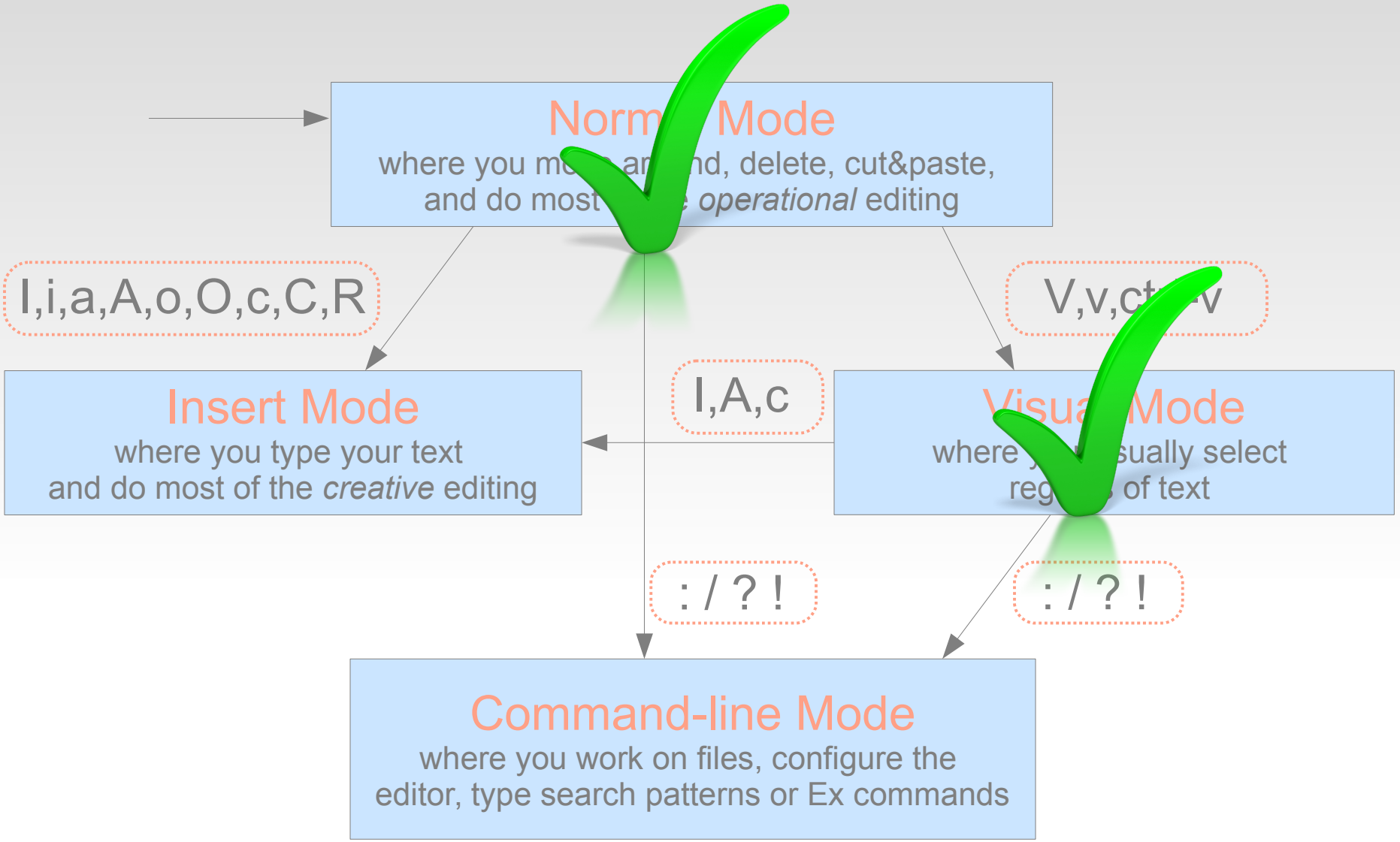
```
1 In the Beginning was the Command Line¬
2 ¬
3 ¬
```

```
1 In the Beginning was the Command Line¬
2 In the Beginning was the Command Line¬
3 ¬
```

```
1 In the Beginning was the Command Line¬
2 ==================================¬
3 ¬
```

```
1 point1¬
2 point2¬
3 point3¬
4 point4¬
5 point5¬
6 point6¬
7 point7¬
8 point8¬
```

```
1 - point1¬
2 - point2¬
3 - point3¬
4 - point4¬
5 - point5¬
6 - point6¬
7 - point7¬
8 - point8¬
```


Normal Mode
where you move around, delete, cut&paste,
and do most operational editing
I,i,a,A,o,O,c,C,R
V,v,ctrl-v
Insert Mode
where you type your text
and do most of the creative editing
I,A,c
Visual Mode
where you visually select
regions of text
: / ? !
: / ? !
Command-line Mode
where you work on files, configure the
editor, type search patterns or Ex commands

Insert Mode

Some useful shortcuts..

<ctrl-r>r	Insert the content of register r
<ctrl-y> <ctrl-e>	Insert the character just above or below
<ctrl-v>c <ctrl-v>ddd <ctrl-v>xhh <ctrl-v>uxxxx	Literally enter the character c (e.g. a tab or an escape) or the character with ascii code ddd (decimal) or hh (hexadecimal), or the unicode character xxxx (hexadecimal)
<ctrl-k>digraph	Insert a digraph (characters that normally cannot be entered by an ordinary keyboard. E.g: e' = é o: = ö

You can find out the code of a character by using the `ga` command in normal mode

:help insert.txt

Spell Checking

- Vim has an integrated, on-the-fly spell checker

  `:set spell`

  `:spelllang=en_us,it` (you can use multiple languages at the same time)

- Four types of words are highlighted:
  - *Not recognized*, *not capitalized*, *rare* words, and *wrongly spelled* for the selected region (e.g., grey in US English)

]s [s	Move to the previous (next) misspelled word
zg	Mark as good (add the current word to your local dictionary)
zw	Mark as wrong (comment the word out of the dictionary)
z=	Suggest a correction for the word under the cursor

`:help spell.txt`

Abbreviations

- Abbreviations are a way to automatically substitute a typed word with something else
  - Useful to auto-correct words you often misspell

    `teh` → `the` `dont` → `don't` `eurecom` → `Eurecom`
  - Useful to abbreviate text you need to type often

    `me@` → `davide.balzarotti@eurecom.fr`

    `main(` → `main(int argc, char* argv[]){`
- Definition:

  `:iab word whathever_you_want`
- To avoid the expansion of a world, type `<ctrl-v>` after it

Auto-Completion (`ctrl-x` mode)

- A sub-mode of insert-mode used to auto-complete text

ctrl-x ctrl-n	words in the current document
ctrl-x ctrl-k	words from the dictionary
ctrl-x ctrl-t	words from the thesaurus
ctrl-x ctrl-f	file names
ctrl-x ctrl-i	included file (depends of the file type)
ctrl-x ctrl-o	user-provided context-aware completion (works out of the box for html, css, php, python,...)
ctrl-x ctrl-]	c-tags
ctrl-x ctrl-l	entire lines
ctrl-n	words from a number of sources, by default including all buffers

```
1 <HTML> ¬
2   <a h
3 </HTML>¬
```

```
1 <HTML> ¬
2   <a href="
3 </HT  href="      *URI
       hreflang="  LangCode
       charset="   LangCode
       shape="     Shape
```

```
1 ¬
2 A fast solution¬
3 ¬
```

```
1 ¬
2 A fast solution¬
3 ¬ fast    /home/balzarot/.vim/thesaurus
4 ¬ speedy  /home/balzarot/.vim/thesaurus
    quick   /home/balzarot/.vim/thesaurus
    swift   /home/balzarot/.vim/thesaurus
    rapid   /home/balzarot/.vim/thesaurus
```

```
1 <HTML> ¬
2   <a h
3 </HTML>¬
```

Ctrl x Ctrl o

```
1 <HTML> ¬
2   <a href="
3 </HT  href="      *URI
       hreflang="  LangCode
       charset="   LangCode
       sh          hape
```

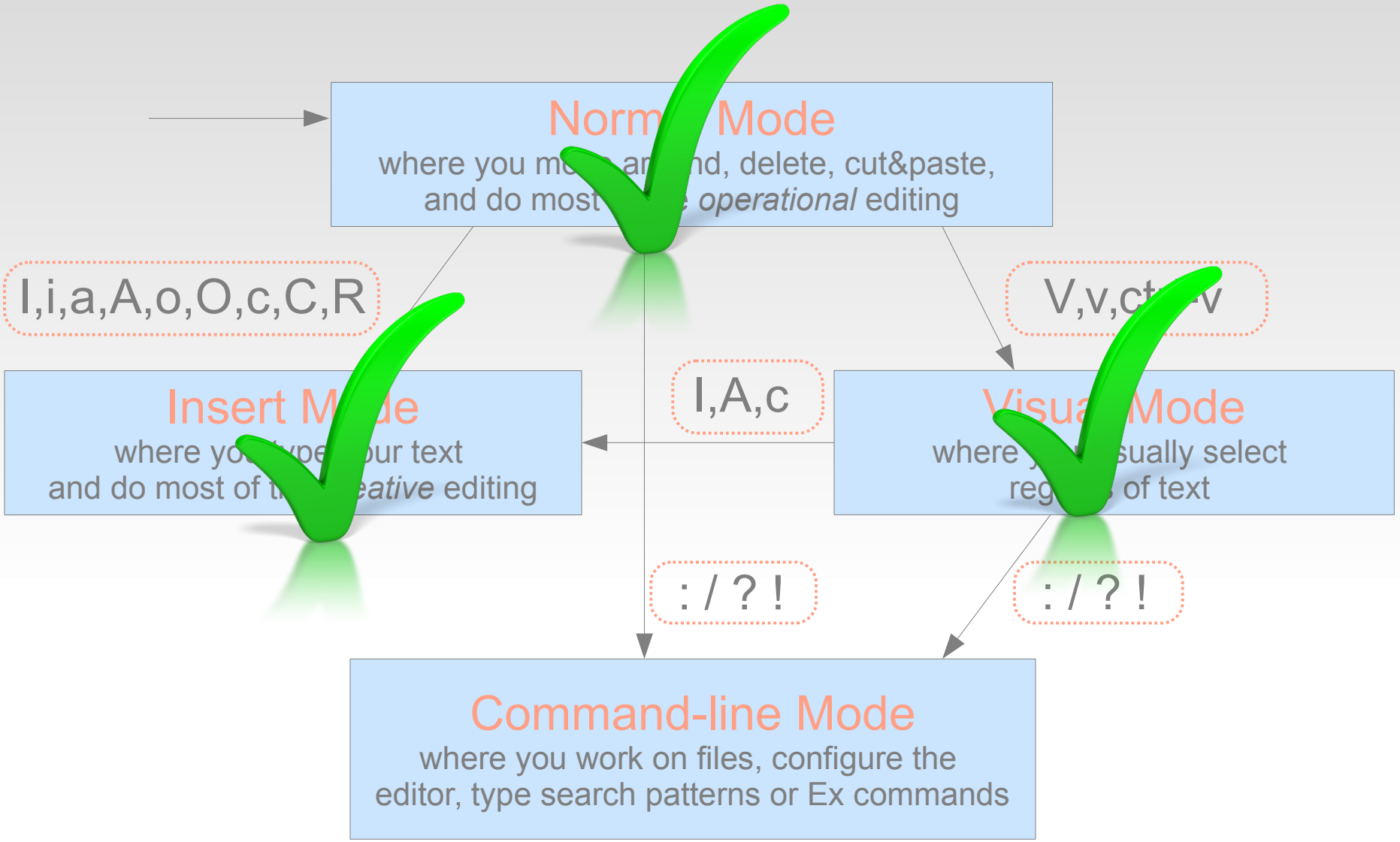
Dictionary and thesaurus auto-completion need to be configured to point to the right dictionary files

:set dictionary=/etc/dictionaries-common/words

```
1 ¬
2 A fast sc
3 ¬
```

Ctrl x Ctrl t

```
1 ¬
2 A fast solution¬
3 ¬ fast    /home/balzarot/.vim/thesaurus
4 ¬ speedy  /home/balzarot/.vim/thesaurus
    quick   /home/balzarot/.vim/thesaurus
    swift   /home/balzarot/.vim/thesaurus
    rapid   /home/balzarot/.vim/thesaurus
```


Normal Mode
where you move around, delete, cut&paste,
and do most operational editing
I,i,a,A,o,O,c,C,R
V,v,ctrl-v
Insert Mode
where you type your text
and do most of the creative editing
I,A,c
Visual Mode
where you visually select
regions of text
: / ? !
: / ? !
Command-line Mode
where you work on files, configure the
editor, type search patterns or Ex commands

Command-Line Mode

Command-Line Mode

- Command-line mode is used to enter
  - Ex commands ("`:`" or "`q:`")
  - Forward ("`/`" and "`q/`") and backward ("`?`" and "`q?`") search patterns
  - Filter commands ("`!`")
- Normally the command is typed in a single line that appears at the bottom of the screen
  - However, if entered through the `q` command, the entire history is shown in a separate vim windows

Searching

- The basics:
  - Use `/regex` or `?regex` to search for a regular expression forward or backward
  - Use `n` and `N` to move to the next and previous match
- Tricks
  - Search commands can be used as any other movement command. E.g. `d/foo` deletes until the next appearance of `foo`
  - Use `<` and `>` to delimit words in a regex. E.g. `<foo>`
  - Start a regex by `\v` to use a more intuitive syntax
  - You can tell vim to highlight the matches, and then move from one to the other. If you want to select the current match you can use `v//e`

Ex commands: the core of VI

[range] command [parameters]

`N`	– line N
`.`	– the current line
`%`	– the entire file
`$`	– the last line of the file
`'m`	– line of mark m
`N,M`	– between line N and line M
`'<,'>`	– the lines containing the visually-selected area
`+X`	– X lines ahead
`/regexp/`	– next line matching the regular expression
`?regexp?`	– previous line matching the regular expression

`:help cmdline-ranges`

Ex commands: the core of VI

:[range] command [parameters]

print	just print the line
write [>>] [filename]	write range (default the file) to disk (default to the open file). '>>' can be used to append to a file
read filename	insert the content of a file after the line defined by range (default the current one)
read !shell_cmd	same as before but with the output of a shell command
del [reg], yank [reg]	equivalent to y and d in normal mode
copy [dest_pos] move [dest_pos]	copy the range lines after dest_pos move the range lines after dest_pos
normal cmd	execute the normal-mode command on each line of the range

```
:%normal A;
```

- Add a semicolon at the end of each line

```
:'<,'>normal .
```

- Repeat the last command on the selected lines

s & g

`:[range]s/{regex}/{replacement}/[flags]`

- Replace the first match of the regular expression in the range (default the current line) with the replacement.
- Use the `g` flag to replace all matches
- By default the replacement is done line-by-line. If you want to restrict a regex to a visually selected area start the regex with a **\%V**
- If replacement starts with **\=** it is evaluated as an expression

`:[range] g[!] /{regex}/ [ex_cmd]`

- Execute the Ex command on every line in range (default the entire file) that match the regular expression
- ! invert the match, i.e. runs the command on the lines that do not match the regex

s & g

`:[range]s/{regex}/{replacement}/[flags]`

place the first match of the regular expression in the range

`:set gdefault`
to enable the g flag by default

`:set ignorecase`
`:set smartcase`
to have case-insensitive search *unless* the pattern contains uppercase letters

that match the regular expression

- ! invert the match, i.e. runs the command on the lines that do not match the regex

`:g/re/p`

- Does the name sound familiar?
- For instance: `g/FIXME/p`

`:g/FIXME/.w >> fixme.txt`

- Same as before, but save them to a separate file

`qaq :g/FIXME/y A`

- Same as before, but copy them to register a

`:g/^\s*$/d`

- Removes empty lines

`:g/DEBUG/normal I//`

- Comment out every line that contains “DEBUG”

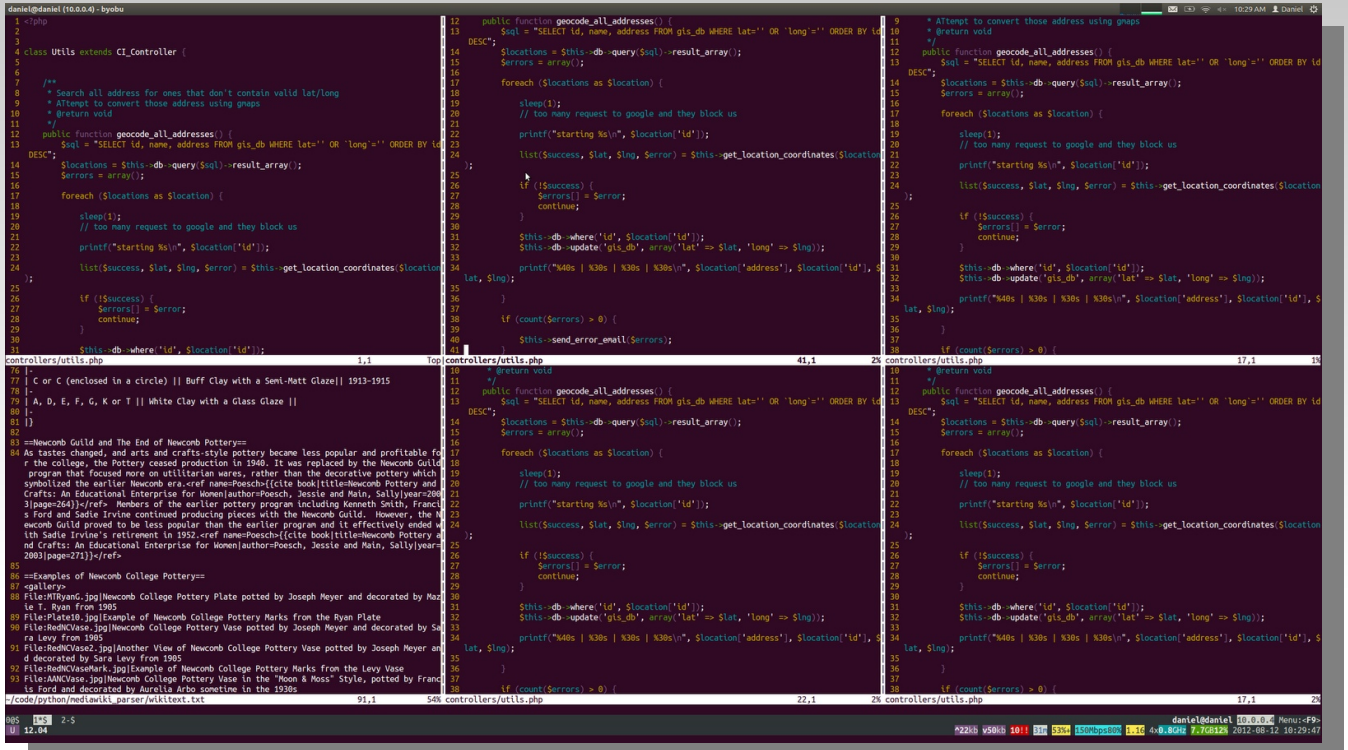
Filters

:[range]! unix_cmd

- Executes a system command, pipes the range lines to its standard input, and insert the output in the buffer
- If no range is specified, the command is still executed but its output is only displayed and NOT inserted back !!
- It can also be triggered by ! in visual mode

- Examples:

```
:!ls   vs   :.!ls
:%!uniq
:'<,'>!column -t
```


daniel@daniel (10.0.0.4) - byobu
controllers/utils.php
~/code/python/mediawiki_parser/wikitext.txt
Working with Multiple Files

Buffers

- A buffer is the in-memory text of a file
  - You can open a buffer by using `:edit filename`
  - You can list buffers with `:ls` (buffers have names and numbers)
  - Buffers are not necessarily visible on the screen
    - By default a buffer can be in background only if it does not have pending modifications. You can change it by setting `:set hidden`
- Working with buffers

:b n	bring buffer n to the current window
:bnext	switch to the next buffer
:bd[!] n	close a buffer
:bufdo cmd	execute a command in all buffers

`:help buffers`

Windows

- A window is a view on a single buffer
  - You can have the same buffer open in multiple windows
  - Useful when you want to see multiple buffers at the same time
  - You can open new windows with `:split` or `:vsplit`
  - Use `ctrl-w w` to move the cursor between windows
  - `:q` close a window (and vim if that was the last window)
    `:q!` force it to close and discharge changes, but...

If you allow modified hidden buffers things may get weird...

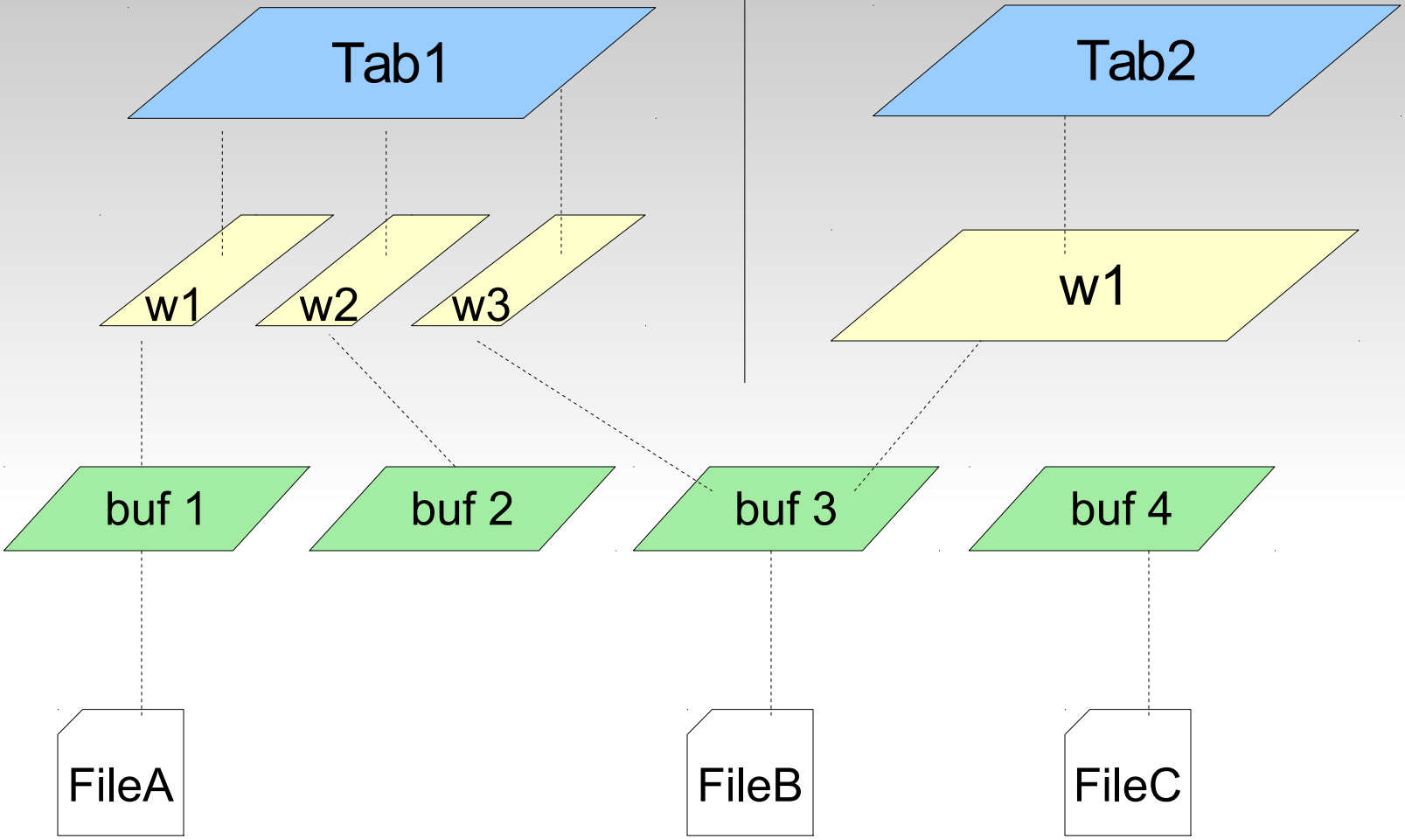
- either you save the changes...
- ..or you force close the buffers (:bd!)..
- .. or you use :qall!

Tabs

- A tab is a collection of windows

:tabnew [filename]	open a new tab
:tabn :tabp	move to the next (or previous) tab
:tab ball	assign each buffer to a different tab
vim -q file1 file2 ..	to open multiple file on different tabs

`:help tabpage`


Tab1
Tab2
w1
w2
w3
w1
buf 1
buf 2
buf 3
buf 4
FileA
FileB
FileC

Don't believe what people say:
There is no “real” or “proper” way to combine tabs, windows, and buffers.
Just find a way that works for you.

Configuration

Vim Configuration

- Even though vanilla Vim is not very fancy, Vim is designed to be extremely configurable to match your taste and needs
  - All the setup and initialization commands are stored in a vimrc file (default `~/.vimrc`)
  - Every line is an Ex command (without the : in front)
- Vim has a large number of internal variables and switches which can be set to tweak its behavior
- Key mappings can be used to change the behavior of typed keys (typically associating a sequence of commands to a single key)

Setting Options

:set {option}?	Show the value of one option
:set	Show the value of the modified options
:set all	Show the value of all options
:set option :set no{option} :set {option}!	Set a boolean option .. unset it.. .. switch its value..
:set option=value	Set a non-boolean option

`:options`

Setting Options

```
" General options¬
set nocompatible      "Use vim instead of vi settings"¬
set mouse=a           "Have the mouse enabled all the time:¬
set number            "Enable line numbering¬
set autoindent        "Autoindent a line as the previous¬
set showmode          "Display the current mode in the status line¬
set showmatch         "Show matching bracets¬
set nobackup          "Do not create annoying ~file¬
set nowrap            "Dont wrap long lines¬
set backspace=eol,start,indent¬
syntax on             "Enable Syntax highlightening¬
¬
" Search Options¬
set ignorecase        "Case-insensitive search¬
set smartcase         " ... unless the term contains upper-case letters ¬
set hlsearch          "Highlight search¬
set wrapscan          "When searching, restart from top when you hit the bottom
set incsearch         "Incremental search¬
¬
" Tab options¬
set expandtab          "Automatically replace TAB with spaces¬
set tabstop=4          "Use 4 characters per TAB¬
set shiftwidth=4       "Autoindent 4 characters¬
¬
:filetype on¬
autocmd FileType human,txt,tex,mail set spell¬
```

`:options`

Key Mapping

[mode][nore]map <key> commands..

`mode` specifies in which mode the mapping is defined
(i=insert, n=normal, v=visual, ...)

`nore` avoid recursive mappings

- Examples:

```
:nnoremap <C-l> :nohlsearch<CR><C-l>
:nmap <F1> <nop>
:vnoremap < <gv
:inoremap <CR> <CR><C-G>u
```

- Unfortunately, most of they key are already associated to useful operations :(

Key Mapping

- To extend the set of available keys for user mappings, Vim uses a leader key
  - By default it is set to \ but it can be easily changed:

```
:let mapleader= ","
```

- You can then redefine any sequence of keys that starts with the leader

```
:nnoremap <leader>d" F"xf"x
:nnoremap <leader>-- yypVr-o
:vnoremap <leader>" <esc>`>a"<esc>`<i"<esc>
:nnoremap <leader>t :call LatexBox_TOC()<CR>
```

> *You can find plenty of ideas of "useful" mappings. But keep in mind that the most important thing is to look at your current behavior and try to simplify repetitive tasks*

Events

- Vim provides hooks for 78 distinct editing events
  - `:help autocmd-events`

```
> vim
    BufWinEnter     (create a default window)
    BufEnter        (create a default buffer)
    VimEnter        (start the Vim session):edit example.txt
    BufNew          (create a new buffer to contain demo.txt)
    BufAdd          (add that new buffer to the session's buffer list)
    BufLeave        (exit the default buffer)
    BufWinLeave     (exit the default window)
    BufUnload       (remove the default buffer from the buffer list)
    BufDelete       (deallocate the default buffer)
    BufReadCmd      (read the contexts of demo.txt into the new buffer)
    BufEnter        (activate the new buffer)
    BufWinEnter     (activate the new buffer's window)i
    InsertEnter     (swap into Insert mode)
Hello
    CursorMovedI    (insert a character)
    CursorMovedI    (insert a character)
    CursorMovedI    (insert a character)
    CursorMovedI    (insert a character)
    CursorMovedI    (insert a character)<ESC>
    InsertLeave     (swap back to Normal mode):wq
    BufWriteCmd     (save the buffer contents back to disk)
    BufWinLeave     (exit the buffer's window)
    BufUnload       (remove the buffer from the buffer list)
    VimLeavePre     (get ready to quit Vim)
    VimLeave        (quit Vim)
```

Events

- It is possible to add automatic behaviors for each of them

```
autocmd  EventName  filename_pattern  :command
```

- Examples:
  - autocmd FileType help nmap <buffer> <Return> <C-]>
  - autocmd FileType html,css setlocal tabstop=2
  - autocmd FileType make setlocal noexpandtab shiftwidth=8
  - autocmd FileType human,txt,tex,mail,asciidoc set spell

Plugins

Plugins

- Plugins are scripts designed to extend VIM functionality
  - They can be global (saved under `~/.vim/plugin/`) or specific for a certain file type (under `~/.vim/ftplugin/`)
- You can install a new plugin by simply copying its file in the right directories
- Or you can use a plugin manager
  - Vim-addon-manager (command-line utility)
  - Pathogen (vim plugin)
  - Vundle (vim plugin)

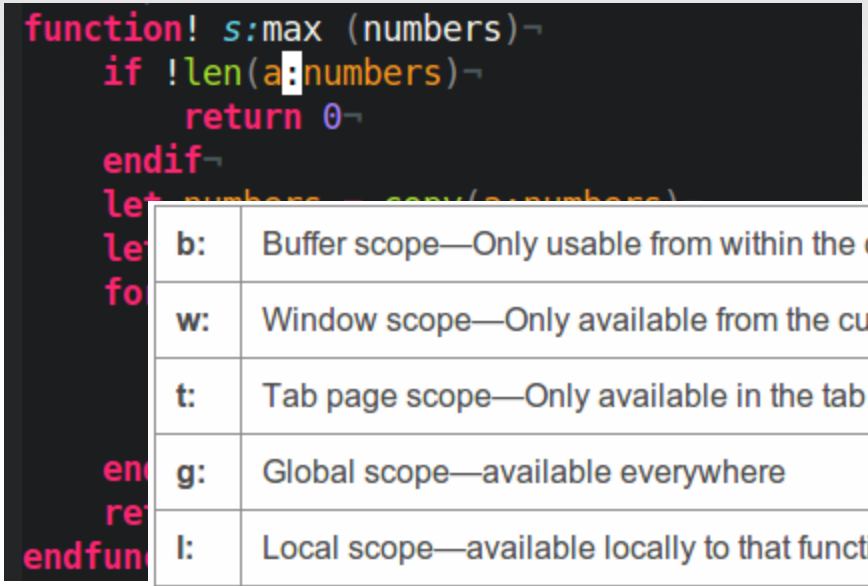
Plugins

By default, plugins are written using the vimscript scripting language:

```
function! s:max (numbers)¬
    if !len(a:numbers)¬
        return 0¬
    endif¬
    let numbers = copy(a:numbers)¬
    let maxnum = numbers[0]¬
    for nextnum in numbers[1:]¬
        if nextnum > maxnum¬
            let maxnum = nextnum¬
        endif¬
    endfor¬
    return maxnum¬
endfunction¬
```

Plugins

By default, plugins are written using the vimscript scripting language:



b:	Buffer scope—Only usable from within the current buffer
w:	Window scope—Only available from the current window
t:	Tab page scope—Only available in the tab page
g:	Global scope—available everywhere
l:	Local scope—available locally to that function defined
s:	Source scope—available only within the sourced file
a:	Argument scope—only available within the function
v:	Global scope—used to refer to a variable defined and used by **Vim**

Scripting in Python

```
function FixPlural()¬
python << EOF¬
import vim¬
cur_word = vim.eval('expand("<cWORD>")')¬
if cur_word.endswith('s'):¬
    cur_word = cur_word[:-1]¬
else:¬
    cur_word = cur_word+"s"¬
vim.command("normal ciW%s"%cur_word)¬
EOF¬
endfunction¬
¬
function ExpandNumber()¬
python << EOF¬
import vim¬
numbers = {'1':'one', '2':'two', '3':'three', '4':'four', ¬
           '5':'five', '6':'six', '7':'seven', '8':'eight', ¬
           '9':'nine', '10':'ten'}¬
cur_word = vim.eval('expand("<cWORD>")')¬
if cur_word in numbers:¬
    cur_word = numbers[cur_word]¬
vim.command("normal ciW%s"%cur_word)¬
EOF¬
endfunction¬
¬
nnoremap S :call FixPlural()<Esc>¬
nnoremap <leader>N :call ExpandNumber()<Esc>¬
¬
```

> *Mastering is more than just knowing. It is knowing in a way that lightens your load.*

-from “Apprenticeship Patterns”

Happy Vimming