

LUCKY HUNTER

BOOK NATION

TECHNICAL ENTHUSIAST

ELECTRONIC

MASTERING THE COMMAND LINE LIKE LIKE A A HACKER



Xiaodong Xu

Mastering the Command Line Like a Hacker

Xiaodong Xu

Contents

1	Getting Started	1
1.1	Console	1
1.2	Terminal	3
1.3	Terminal Emulator	4
1.3.1	Linux	5
1.3.2	macOS	6
1.3.3	Windows	6
1.4	Shell	6
1.4.1	sh	8
1.4.2	csh	8
1.4.3	ksh	8
1.4.4	bash	9
1.4.5	zsh	9
1.5	Command-Line Interface	9
1.5.1	Powerful Functionality	10
1.5.2	Flexible and Efficient	11
1.5.3	Can Be Automated	11
1.6	How to Enter the Command Line	12
1.6.1	Enter the Command Line Through the Console	12
1.6.2	Access the Command Line Through the Terminal Emulator	12
1.7	Hello, the Command Line	13

2	Magical Auto-Completion	15
2.1	What Is Auto-Completion	15
2.2	Trigger Auto-Completion with Keys	17
2.3	File Name and Path Name Auto-Completion	17
2.4	Command Name and Program Name Auto-Completion	19
2.4.1	Zsh Auto-Suggestion Plugin	25
2.5	Username, Hostname, and Variable Name Auto-Completion	26
2.6	Programmable Auto-Completion	30
2.6.1	Bash Example	30
2.6.2	Zsh Example	32
3	Reviewing History	35
3.1	Setting Historical Variables	35
3.2	Viewing Command History	37
3.3	Searching Command History	38
3.4	Navigating Command History	39
3.5	Quickly Edit and Execute Previous Command	39
3.5.1	Remove Redundant Content	40
3.5.2	Replace Content	40
3.5.3	Global Replacement	41
3.6	Quickly Executing Historical Commands	41
3.6.1	Repeat the Previous Command	41
3.6.2	Executing Commands Starting with Specific Characters	42
3.6.3	Execute the Nth Command in the Command History List	42
3.7	Quickly Referencing the Arguments of the Previous Command	44
3.7.1	Quoting the Last Argument	44
3.7.2	Quoting the First Argument	44
3.7.3	Quoting All Arguments	45
3.7.4	Quoting the Nth Argument	45
3.7.5	Quoting Arguments from M to N	46
3.7.6	Quoting Arguments from N to the Last	46
3.8	Partially Quoting an Argument	47

3.8.1	Quoting Path Head	47
3.8.2	Quoting Path Tail	47
3.8.3	Quoting File Name	48
3.8.4	Change Reference Portion to Uppercase	48
3.8.5	Change Referenced Portion to Lowercase	48
3.9	Historical Command Expansion Modes Summary	49
4	The Art of Editing	51
4.1	Set Editing Mode	51
4.2	Emacs Editing Mode in Practice	52
4.2.1	Moving and Deleting by Character	52
4.2.2	Moving and Deleting by Word	54
4.2.3	Moving and Deleting by Line	56
4.2.4	Emacs Editing Mode Summary	57
4.3	vi Editing Mode in Practice	57
4.3.1	Movement Commands	58
4.3.2	Repeating Commands	59
4.3.3	Add Text	59
4.3.4	Delete Text	60
4.3.5	Replace Text	61
4.3.6	Searching for Characters	62
4.3.7	vi Editing Mode Summary	63
5	Essential Toolkit	65
5.1	Quick Navigation	65
5.1.1	Back to User Home Directory	65
5.1.2	Return to the Last Working Directory	66
5.1.3	Accessing Common Directories	67
5.1.4	Auto-Correction of Errors	67
5.1.5	Automatic Navigation	68
5.1.6	Using the Stack of Directories	69
5.2	Using Aliases	71
5.2.1	Defining Aliases	71

5.2.2	Viewing Aliases	72
5.2.3	Removing Aliases	72
5.2.4	Drawbacks of Aliases	73
5.3	Using {} to Construct Arguments	74
5.3.1	Backup Files	74
5.3.2	Generating Sequences	75
5.3.3	Combination and Nesting	76
5.4	Additional Tips	77
5.4.1	Command Substitution	77
5.4.2	Using Variables	78
5.4.3	Repeated Execution of Commands	79
6	Shell Goodies	81
6.1	Configuring the Framework	81
6.1.1	Bash Configuration Framework	81
6.1.2	Zsh Configuration Framework	91
6.2	Enhanced Tools	99
6.2.1	Quick Path Switching: z.lua	99
6.2.2	Efficient Querying of Shell History: HSTR	103
7	Conclusion	109

List of Tables

2.1	Username, hostname, and variable name auto-completion prefix characters	30
3.1	Navigating command history	39
4.1	Emacs mode navigation and deletion operations by character	53
4.2	Emacs mode word movement and deletion operations	56
4.3	Emacs mode line movement and deletion operation methods	57
4.4	vi mode movement commands	58
4.5	vi mode text addition commands	60
4.6	vi mode delete text commands	60
4.7	vi mode copy and paste commands	61
4.8	vi mode commands for replacing text	62
4.9	vi mode commands for searching characters	62

List of Figures

1.1	Console of the IBM 1620 Model 1	2
1.2	Operator console panel of the IBM 1620 Model 1	2
1.3	Linux virtual console	3
1.4	DEC VT100 terminal	4
1.5	Xterm terminal emulator	5
1.6	The shell and the kernel	7
1.7	Right prompt in Zsh	10
1.8	Command line interface	13
2.1	Bash autocomplete configuration results	17
2.2	Auto-completion of file names in GIMP	20
2.3	Command auto-completion candidates	21
2.4	Alternate command completion list	23
2.5	Command option auto-completion in Zsh	24
2.6	Command auto-suggestions in Zsh	25
2.7	Username auto-completion candidates in Bash	26
2.8	Username auto-completion candidates in Zsh	27
2.9	Sources of auto-completion hostname	28
2.10	List of possible variable name completions in Bash	29
2.11	Variable name auto-completion suggestions in Zsh	29
2.12	Example of programmable Bash completion	32
2.13	Zsh programmable completion example	33
3.1	Reverse search of command history	39

3.2	history 5 execution results	43
3.3	Command and option numbering	45
3.4	History command expansion mode	50
4.1	Emacs editing mode illustration	58
4.2	vi editing mode illustration	63
6.1	Bash-it installation process	83
6.2	Viewing aliases in Bash-it	84
6.3	Viewing completion in Bash-it	85
6.4	Viewing plugins in Bash-it	85
6.5	Bash-it git aliases	87
6.6	Bash-it prompt theme	89
6.7	Oh My Zsh installation process	92
6.8	Oh My Zsh plugin directory	93
6.9	Output of the man zsh command	94
6.10	Output of the sc-status sshd command	94
6.11	The simple theme style of Oh My Zsh	96
6.12	zsh-syntax-highlighting is disabled	97
6.13	zsh-syntax-highlighting is enabled	98
6.14	Upgrade Oh My Zsh	99
6.15	HSTR interface	106
6.16	Result of running hh nvim	107

Copyright

Copyright protection applies to this publication. Except for personal use, no part of it may be copied, recorded, stored, or transmitted for commercial purposes in any format without the prior consent of the publisher.

This work is released under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, which allows for personal use and sharing as long as the work is attributed to its original creator, not used commercially, and not modified in any way. For more information about this license, visit <https://creativecommons.org/licenses/by-nc-nd/4.0/>.

The content of this publication is owned and copyrighted by Xiaodong Xu, 2019-2024.

Acknowledgments

I'd like to express my sincere appreciation to the outstanding community of developers and contributors who work with Bash and Zsh open-source projects for all their remarkable efforts and contributions.

Update

You can find a more updated version of this book at <https://selfhostedserver.com/usingcli-book>. Additionally, access to a video guide for the book is available at <https://selfhostedserver.com/usingcli>.

- Version 2019.03.17: First Edition
- Version 2019.08.24: Revision Edition
- Version 2024.11.15: English Edition

About the Author

Xiaodong Xu, also known by his alias “~toy”, is an avid GNU/Linux enthusiast and DevOps practitioner. He has a strong passion for technology and sharing knowledge with others. Through his personal website, [LinuxToy.org](https://linuxtoy.org)¹, he has written and translated over 3,000 articles. Xiaodong Xu is also the author of several books including “Mastering the Command Line Like a Hacker²”, “The Containerization Trio: Podman, Buildah, and Skopeo³”, and “Terraform: Automating Cloud Infrastructure Management⁴”. Furthermore, he has translated notable books such as “Git for Dummies” and “Things Every Perl Programmer Should Know”. To stay updated with his latest work, you can follow him on Twitter at [@linuxtoy](https://twitter.com/linuxtoy)⁵ or reach out to him via email at xuxiaodong@pm.me⁶.

¹<https://linuxtoy.org>

²<https://selfhostedserver.com/usingcli-book>

³<https://selfhostedserver.com/nextcontainer>

⁴<https://selfhostedserver.com/terraform>

⁵<https://twitter.com/linuxtoy>

⁶<mailto:xuxiaodong@pm.me>

Chapter 1

Getting Started

Although graphical user interfaces are omnipresent in today's technology, command-line interfaces were the predominant method of interaction during the early stages of computing. In graphical user interfaces, it is customary to utilize a mouse to interact with icons or windows in order to accomplish various tasks. In contrast, command-line interfaces necessitate more keyboard-based interaction. To provide a comprehensive definition of a command-line interface, it is first necessary to explore the fundamental concepts of consoles, terminals, terminal emulators, and shells.

1.1 Console

The console, also referred to as a system console, computer console, root console, or operator's console, has a long and storied history that dates back to the early days of computing. Historically, a console was a hardware component that allowed users to interact with a computer, as depicted in Figure 1.1.¹

This image shows that the control unit of the IBM 1620 computer is made up of two main components. On the left is the front panel where the operator works, as seen in Figure 1.2. On the right, there is a typewriter. The control unit allows operators to input text or instructions, which the computer then reads or executes as per their input.

The introduction of computers has led to the evolution of the console from a hardware concept to a software concept. Consequently, the console has taken on a new name: the virtual console. This virtual console is distinct and separate from the physical console hardware. By observing the Linux system's boot process, one can easily observe that after the computer's hardware self-test, once the boot loader takes control, the system soon enters the system console. During this process, the Linux system boot information is typically displayed, as shown in Figure 1.3.²

¹https://en.wikipedia.org/wiki/System_console#/media/File:IBM_1620_Model_1.jpg

²https://en.wikipedia.org/wiki/Linux_console#/media/File:Knoppix-3.8-boot.png



Figure 1.1: Console of the IBM 1620 Model 1

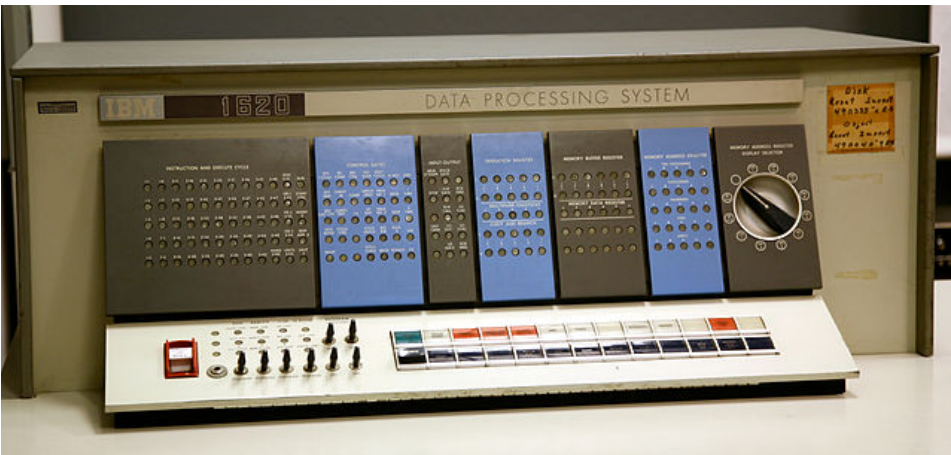
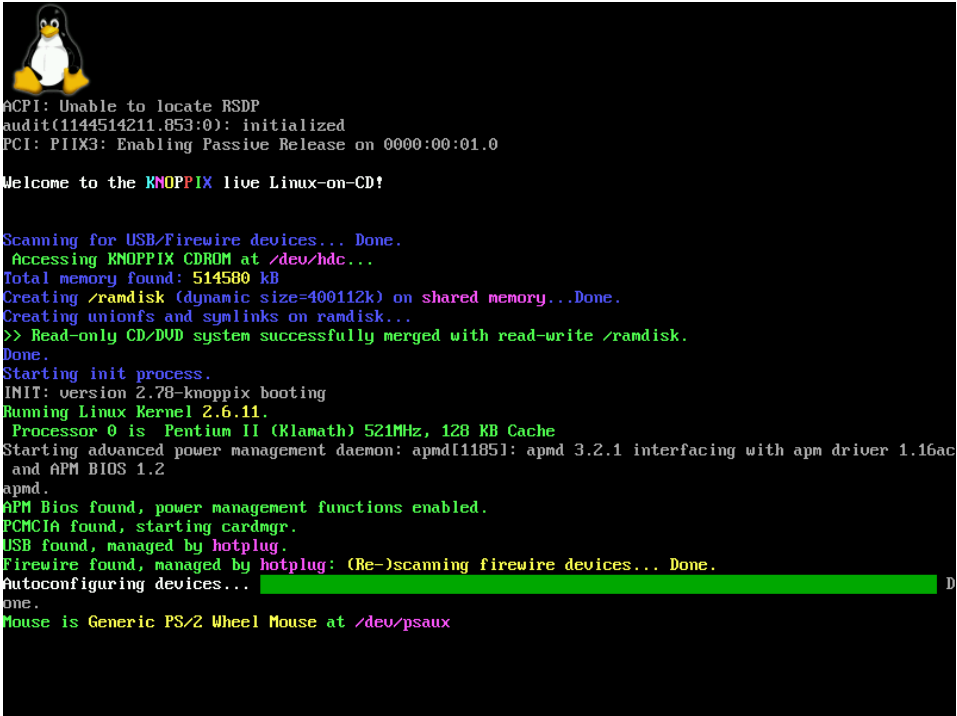


Figure 1.2: Operator console panel of the IBM 1620 Model 1



```

ACPI: Unable to locate RSDP
audit(1144514211.853:0): initialized
PCI: PIIX3: Enabling Passive Release on 0000:00:01.0

Welcome to the KNOPPIX live Linux-on-CD!

Scanning for USB/Firewire devices... Done.
  Accessing KNOPPIX CDROM at /dev/hdc...
Total memory found: 514580 kB
Creating /ramdisk (dynamic size=400112k) on shared memory...Done.
Creating unionfs and symlinks on ramdisk...
>> Read-only CD/DVD system successfully merged with read-write /ramdisk.
Done.
Starting init process.
INIT: version 2.78-knoppix booting
Running Linux Kernel 2.6.11.
  Processor 0 is Pentium II (Klamath) 521MHz, 128 KB Cache
Starting advanced power management daemon: apmd[1185]: apmd 3.2.1 interfacing with apm driver 1.16ac
and APM BIOS 1.2
apmd.
APM Bios found, power management functions enabled.
PCMCIA found, starting cardmgr.
USB found, managed by hotplug.
Firewire found, managed by hotplug: (Re-)scanning firewire devices... Done.
Autoconfiguring devices... Done.
Mouse is Generic PS/2 Wheel Mouse at /dev/psaux

```

Figure 1.3: Linux virtual console

1.2 Terminal

Similar to a console, a terminal was initially a computer hardware device. In terms of appearance, the terminal resembled a combination of the display screen and keyboard that we use today. The terminal allowed users to input commands and data into the computer system. It also displayed the outcome of the computer's processing to the user. Figure 1.4 depicts the widely popular DEC VT100 terminal.³

It may seem puzzling why terminal devices originally emerged. However, when considering their origins, it becomes clear that this innovation was largely driven by necessity. When computers first emerged, they were incredibly costly and inaccessible to the general public. Computers were often only found in large commercial establishments or university research institutions. As a result, terminals were developed to enable the sharing of computing resources. With the passage of time and advancements in technology, terminals eventually became obsolete. However, they were later revived in a modern form, now referred to as terminal emulators or virtual terminals.

³https://en.wikipedia.org/wiki/Computer_terminal#/media/File:DEC_VT100_terminal.jpg



A terminal emulator is a software application that replicates the functionality of a physical terminal device. These programs can also display complex terminal features, such as escape sequences used for managing text color and cursor placement, similar to those found in traditional terminals. One popular example of a terminal emulator in Linux is shown in Figure 1.5, which depicts Xterm, a widely used terminal application.

Regardless of whether you operate on Linux, macOS, or Windows, numerous terminal emulator options are currently available. A selection of well-known terminal emulators for each of these three operating systems is listed below.



Figure 1.5: Xterm terminal emulator

1.3.1 Linux

- **Xterm**⁴: Xterm is the standard terminal emulator for the X Window System, providing features similar to those found on DEC VT102 and Tektronix 4014 terminals, with added support for ISO/ANSI color modes.
- **GNOME Terminal**⁵: GNOME Terminal serves as the primary terminal application within the GNOME desktop environment. It offers functionalities comparable to those of Xterm. In addition to these capabilities, the application also provides support for multiple configurations, tab management, mouse events recognition, and several other notable features.
- **Konsole**⁶: Konsole is the standard terminal application in the KDE desktop environment. It offers several useful features such as tabbed terminals, multiple user profiles, a bookmarking function, and a search option.
- **rxvt-unicode**⁷: rxvt-unicode, which was originally derived from rxvt, includes support for Unicode characters and offers a wide range of customization options. Furthermore, rxvt-unicode provides daemon mode and embedded Perl programming capabilities, making it a versatile terminal emulator and the preferred choice of the author.
- **kitty**⁸: Kitty is a fast and feature-rich terminal emulator that utilizes a graphics processing unit (GPU). It provides support for graphics, emojis, and hy-

⁴<https://invisible-island.net/xterm/>

⁵<https://gitlab.gnome.org/GNOME/gnome-terminal/>

⁶<https://kde.org/applications/system/konsole/>

⁷<http://software.schmorp.de/pkg/rxvt-unicode.html>

⁸<https://sw.kovidgoyal.net/kitty/>

perlinks. Furthermore, kitty offers a comprehensive window management system, allowing users to organize their workspace into tabs, splits, and multiple layouts. Additionally, kitty is scriptable and can be extended using the Python programming language, providing users with a high degree of customization.

1.3.2 macOS

- **Terminal:** The Terminal application is the default terminal emulator that comes pre-installed with macOS. Although it has limited features, two of its most notable functions are the ability to set the TERM environment variable and a built-in search function that can be used to locate user manuals, also known as Man pages.
- **iTerm2⁹:** iTerm2 is a highly sought-after, open-source alternative to the default terminal on macOS, boasting an array of impressive features such as window splitting, auto-completion, mouseless copying and pasting, and a rich history of pasted commands, among others. As a macOS user, you may find iTerm2 to be an indispensable terminal emulator that is sure to exceed your expectations.

1.3.3 Windows

- **Mintty¹⁰:** Mintty is a terminal emulator that supports multiple environments, including Cygwin, MSYS, and Windows Subsystem for Linux (WSL). It is compatible with Xterm, offering features such as 256-color and true-color support, as well as Unicode and Emoji compatibility.
- **ConEmu¹¹:** ConEmu is a highly popular, open-source terminal emulator designed for Windows, which features a tabbed interface, multiple graphic window modes, and user-friendly text block selection.

1.4 Shell

The shell is a command interpreter that reads, parses, and executes user input commands. In addition to interacting directly with users, modern shells also possess programming capabilities, supporting basic programming elements such as variables, arrays, functions, loops, and conditional statements.

The term “shell” originates from its position as the outermost layer of the operating system, relative to the Unix and Linux core, or kernel. This also means that the shell serves as a user interface to access system services, playing a crucial role in interacting with the kernel, as illustrated in the accompanying Figure 1.6.

⁹<https://www.iterm2.com/>

¹⁰<https://mintty.github.io/>

¹¹<https://conemu.github.io/>

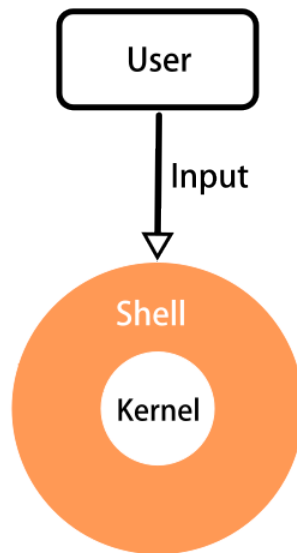


Figure 1.6: The shell and the kernel

During the development of Unix and Linux, numerous shells have emerged, with some notable examples including `sh`, `csh`, `ksh`, `bash`, and `zsh`.

1.4.1 `sh`

The `sh`, also known as the Bourne Shell, was the default shell for the seventh edition of Unix. The `sh` was developed by Stephen Bourne at Bell Labs and was released in 1979. Its popularity significantly increased following the publication of “The Unix Programming Environment,” which was written by Brian Kernighan and Rob Pike.

The Bourne Shell has been largely replaced by later versions of shells, and on modern Linux systems, `sh` is usually a symbolic link to a compatible shell. For instance, on the Debian 12 system used by the author of this book, `sh` is equivalent to `dash`.

```
root@toydroid:~# ls -l /bin/sh
lrwxrwxrwx 1 root root 4 Jan 24  2017 /bin/sh -> dash
```

On the author’s other system, which runs Arch Linux, the command `sh` actually refers to the Bash shell.

```
root@codeland:~# ls -l /bin/sh
lrwxrwxrwx 1 root root 4 Feb  7 15:15 /bin/sh -> bash
```

1.4.2 `csh`

The C-shell (`csh`), which was developed by Bill Joy, was widely distributed via the Berkeley Software Distribution (BSD). Modeled after the C programming language, the C-shell was designed to provide C programmers with a familiar interface, incorporating several user-friendly interactive elements. A number of its innovative features, including command history, aliases, directory stacks, filename completion, and job control – which were ultimately adopted by other shells – originated from the C-shell.

The `tcsh`, an enhanced version of `csh`, is currently the default shell used in FreeBSD.

1.4.3 `ksh`

The KornShell, commonly referred to as `ksh`, was developed by David Korn and initially released in 1983. It is compliant with the POSIX standard and offers backward compatibility with the Bourne Shell, incorporating several features from

the C-shell. A significant feature of the KornShell is the option to use either vi or Emacs command-line editing modes, enabling users to operate in their preferred way. Furthermore, the Korn Shell includes the feature of associative arrays.

KornShell was initially distributed as a proprietary software, so its widespread adoption was limited. As a result, alternative versions emerged, including Public Domain KornShell (pdksh) and MirBSD KornShell (mksh), the latter of which eventually became the default shell on Android.

1.4.4 bash

Bash originated as part of the GNU Project, with the primary objective of replacing the Bourne Shell. Brian Fox developed the initial version, which was first released in 1989. Over time, Bash has gained immense popularity and has become the default shell for the majority of Linux distributions. In addition, with the introduction of the Windows Subsystem for Linux (WSL), Bash can now also be installed and utilized on Windows 10.

The name “bash” stands for Bourne-again shell, an acronym derived from the words “bourne” and “again” and the abbreviation “sh” for shell. It follows POSIX standards and incorporates features from various shells, such as sh, csh, and ksh.

1.4.5 zsh

The Z-Shell, commonly referred to as Zsh, was first developed by Paul Falstad and made available in 1990. This powerful shell has significantly expanded upon the functionality of the Bourne Shell by incorporating features from various other shells, including tcsh, ksh, and bash.

In terms of interactive user experience, Zsh truly excels. For example, it supports option completion for commands and allows for customizable right-side prompts, as shown in Figure 1.7.

This book primarily concentrates on Bash and Zsh, which are currently the two most widely used shells.

1.5 Command-Line Interface

Users input instructions through a Command-Line Interface (CLI). After submitting a command, the shell assumes responsibility for interpreting and executing the instruction.

In contrast to CLI, GUI stands for Graphical User Interface, which uses graphical means to enable user-computer interaction. Given its ease of use, modern operating systems, including Linux, macOS, and Windows, all provide a graphical user interface.

```

→ 1                                     ~/src/usingcli ± master · 1
-
├── [ 416]  _book/
├── [  96]  css/
├── [ 416]  images/
├── [ 192]  latex/
├── [7.5K]  01-getting-started.Rmd
├── [ 116]  _bookdown.yml
├── [ 687]  _output.yml
├── [ 848]  index.Rmd
├── [ 225]  usingcli.Rproj
└── [  60]  usingcli.code-workspace

4 directories, 6 files
→ █                                     ~/src/usingcli ± master · 1

```

Figure 1.7: Right prompt in Zsh

Just because a graphical user interface is generally more user-friendly than a command-line interface, it does not mean we should completely abandon the command-line interface. In fact, experienced users often excel in using the command-line interface due to several advantages.

1.5.1 Powerful Functionality

Let's take a look at an example:

```

xiaodong@codeland:~$ history |
awk '{CMD[$2]++;count++;}END {
  for (a in CMD)print CMD[a] " " " \
  CMD[a]/count*100 "% " a;}' |
grep -v "./" |
column -c3 -s " " -t |
sort -nr |
nl |
head -n10

```

After running this command on the author's macOS system, the output is as follows:

```

1 1348 14.3771% cd
2 1034 11.0282% l
3 838 8.93771% git

```

4	569	6.06869%	ssh
5	513	5.47142%	cat
6	405	4.31954%	vim
7	372	3.96758%	brew
8	360	3.83959%	scp
9	265	2.82637%	rm
10	264	2.8157%	grep

This command, despite appearing intimidating due to the use of seven commands, including `history`, `awk`, `grep`, `column`, `sort`, `nl`, and `head`, connected by the pipe symbol (`|`), yields a fascinating result. It statistically analyzes all the commands you have executed in the command line and displays the top 10 most frequently used commands, along with their usage counts and corresponding percentages.

The pipe symbol enables the output of the previous command to be used as input for the next command, allowing seemingly unrelated commands to collaborate seamlessly, much like building with blocks. This is where the true power of the command line lies.

1.5.2 Flexible and Efficient

Let's consider another example. Suppose we want to find photos taken in March from the `photos` directory and save their file names to a text file called `mar_photos.txt`. In a graphical user interface, we might open a file manager (such as GNOME Files on Linux or Finder on macOS). We would navigate to the `photos` directory and switch to detailed view mode. Then, we would carefully scan through the files and identify the photos we need. Now, how would we write the file names to the text file? We could type them out manually or use copy and paste to save some effort. However, if we have many files to find, this would be a laborious task.

However, if we're working from the command line, we can accomplish this with a simple command:

```
xiaodong@codeland:~$ cd photos; \
ls -l | grep 'Mar' | awk '{ print $9 }' > mar_photos.txt
```

1.5.3 Can Be Automated

Using the command line also has the benefit of allowing us to automate various operations. The shell enables us to compile our commands into functions or scripts, which not only can be executed repeatedly but also work more efficiently than manual inputs.

```
xiaodong@codeland:~$ ./script.sh
```

1.6 How to Enter the Command Line

From the previous descriptions, you should now understand that the interface where we input commands is provided by the shell. How do we execute the shell? There are two ways to access the command line, as shown below.

1.6.1 Enter the Command Line Through the Console

To conserve system resources, Linux servers typically do not come with a graphical user interface. After booting up, you will enter the command-line interface after logging in by inputting your username and password at the console prompt.

```
login:  
Password:
```

As an ordinary user, you typically use a Linux desktop system with a graphical user interface. Once it's launched, you're taken directly to the desktop. To access the console, follow these steps:

1. Press Ctrl + Alt + F1 to access console number 1.
2. Use the keyboard shortcut Ctrl + Alt + F2 to switch to console number 2.
3. Following suit, you can enter the 3rd, 4th, 5th, and 6th consoles in sequence. By default, Linux typically provides six consoles.
4. To switch back to the desktop from the console, simply press Ctrl + Alt + F7.



If we lose our bearings in the console jungle, we can use the `tty` command to determine which console we are currently using.

1.6.2 Access the Command Line Through the Terminal Emulator

Another way to access the command-line interface is by using a terminal emulator. The choice of terminal emulator program varies depending on the operating system

being used. As the author of this book, I personally prefer to use `rxvt-unicode` on Linux and `iTerm2` on macOS.

In general, terminal emulator programs are tied to the system's login shell (or default shell). Some terminal emulators provide the ability to change the shell, allowing users to conveniently choose their preferred shell. If the shell cannot be changed directly from the terminal program, it can also be altered using the `chsh` command. For instance, to change the default shell to `zsh`, the following command can be executed:

```
xiaodong@codeland:~$ chsh -s /bin/zsh
```



To determine the current shell you're using, simply run the command `echo $SHELL`.

1.7 Hello, the Command Line

In “The C Programming Language”, the first program introduced by the authors Brian W. Kernighan and Dennis M. Ritchie is to display the message “Hello world” on the screen. To illustrate the use of the command line, we will also display a similar message - “hello, the command line”.

When we enter the console or open a terminal emulator, we typically see a command-line interface similar to the one shown in Figure 1.8.

```
xiaodong@codeland:~$ echo -e "\tHello,command line"
```

1
2
3 4
5

Figure 1.8: Command line interface

From Figure 1.8, we can see that a typical command-line interface usually consists of the following components:

1. The name of the currently logged-in user, which in this case is `xiaodong`.
2. `codeland` is the hostname, consistent with the output of `hostname -s`.
3. The current working directory, with `~` representing the user's home directory, which corresponds to `/home/<username>` on Linux systems and `/Users/<username>` on macOS.

4. The `$` symbol represents the command prompt. Typically, the command-line prompt for regular users differs from that of the superuser (root), for example, in Bash, the default prompt for the root user is `#`.
5. The command to be executed, in this case, `echo -e "\tHello, command line"`, includes not only the `echo` command itself but also its options (`-e`) and parameters (`\tHello, command line`). Options and parameters of a command are typically enclosed in quotes (`"`) to prevent ambiguity due to special characters such as spaces. Both single quotes (`'`) and double quotes (`"`) can be used, but they have different semantic implications.

In addition to these five parts, we can also see characters like `@`, `:`, and spaces in this command prompt. The `@` symbol is used to separate the username from the hostname, similar to how it is used in email addresses. The colon `:` serves as a prompt. Spaces are commonly used to separate command options and arguments. It's worth noting that the command prompt can be customized, so your interface may differ from the one presented here.

Now, please follow along and enter `echo -e "\tHello, command line"` after the command prompt (`$` or `#`). If you make a mistake while typing, don't panic; simply press the **Backspace** or **Delete** key to delete the error and re-enter the correct text. When you've finished entering all the characters, press the **Enter** key.

What did you discover? The command line echoed a message 'hello, command line' back to us. Furthermore, the `\t` in the `echo` command parameters produced a tab character in the output, resulting in an indentation effect.

```
xiaodong@codeland:~$ echo -e "\tHello, command line"
    Hello, command line
```

Congratulations. You've just successfully executed a command in the terminal. Doesn't it feel less daunting than you imagined? In the following chapters, we'll show you how to use the command line more efficiently, thereby boosting your productivity.

Chapter 2

Magical Auto-Completion

If you’ve written code before, you’ve probably heard of “code completion.” This feature is a staple in popular code editors and IDEs (Integrated Development Environments) and is widely appreciated by developers. The shell completion I’m about to discuss is similar in concept. I’m confident that once you’ve learned about the topics covered in this chapter, you’ll grow to love it. First, we’ll explore what auto-completion entails, then examine how to trigger it, and finally delve into the various types of auto-completion, including file and path names, program and command names, usernames, hostnames, and variable names, before introducing programmable completion.

2.1 What Is Auto-Completion

Looking back, the feature I wanted to learn first when studying the command line is definitely auto-completion. Why do I say that? Because auto-completion allows us to input just the beginning of a command or filename and have the shell automatically complete the rest. For those who despise typing lengthy commands or file names, auto-completion is a veritable godsend. By significantly streamlining input and saving time, auto-completion greatly boosts our operational efficiency.

To illustrate what is meant by auto-completion, let’s consider an example. Suppose I wanted to type a complete command line directly in Bash:

```
xiaodong@codeland:~$ ls -l reallylongname.txt
```

to view the information about the text file `reallylongname.txt`.

Then I entered the input:


```
xiaodong@codeland:~$ ls -l r
```

I press the **Tab** key, and Bash auto-completes the rest of the file name for me.

```
xiaodong@codeland:~$ ls -l reallylongname.txt
```

Now let's try the same task with Bash. By leveraging the power of Bash, I saved 17 keystrokes. Isn't that a refreshing experience?

Let's take another example:

```
xiaodong@codeland:~$ ls -l f
```

Next, press **Tab** key, and Bash auto-completes the **file**.

```
xiaodong@codeland:~$ ls -l file
```

Next, I press **Tab** key twice in a row. At this point, Bash displays a list of 5 files that can be completed automatically.

```
xiaodong@codeland:~$ ls -l file  
file1 file2 file3 file4 file5
```

I enter 1 to complete the Bash auto-completion process.

Comparing these two examples, we can observe that if the initial characters we enter are unique, Bash automatically completes the remaining content. On the other hand, if the characters are not unique, a list of candidate completions is provided. However, this requires us to press the **Tab** key twice in a row, which can be a bit tedious.

Below we optimize the Bash automatic completion configuration to make it more user-friendly. Open the `~/.inputrc` file using a text editor (create one if it doesn't exist) and add the following content:

```
# completion  
set show-all-if-ambiguous on  
set visible-stats on  
set colored-completion-prefix on
```

With `show-all-if-ambiguous` enabled, you can view the list of completion candidates by pressing **Tab** key only once. The `visible-stats` option explains the type of each candidate by adding indicator symbols to the end of each list item, such as `@` for symbolic links, `/` for directories, and so on. Finally, the `colored-completion-prefix` option highlights the prefix characters in color. The effect is shown in Figure 2.1.

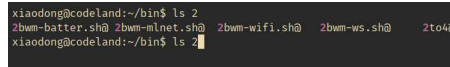


Figure 2.1: Bash autocomplete configuration results

2.2 Trigger Auto-Completion with Keys

Through these examples, we can also see that to trigger auto-completion, it is usually just a matter of pressing the **Tab** key. Both Bash and Zsh use this as their default setting.

2.3 File Name and Path Name Auto-Completion

The above example shows file name auto-completion in Bash. Below, we'll take a look at another example of file name auto-completion in Zsh. Let's say I type in

```
xiaodong@codeland:~$ ls -l f
```

After pressing **Tab** key, Zsh automatically completed the filename for me.

```
xiaodong@codeland:~$ ls -l file
```

I continue by pressing the **Tab** key, at which point Zsh provides an auto-completion menu with possible options.

```
xiaodong@codeland:~$ ls -l file
file1 file2 file3 file4 file5
```

Pressing **Tab** key again will allow you to select a specific menu item.

```
xiaodong@codeland:~$ ls -l file2
file1  **file2**  file3  file4  file5
```

Then press the **Enter** key to complete the auto-completion process.

```
xiaodong@codeland:~$ ls -l file2
```

Finally, press **Enter** key again to execute the command.

Have you noticed the difference in auto-completion between Bash and Zsh? While Bash only provides a list of auto-completion options without allowing us to select a specific item, Zsh offers more flexibility in this regard. This is another example of how Zsh outshines Bash in terms of user interaction. Personally, I prefer using Zsh for this very reason. If you haven't tried Zsh yet, I highly recommend giving it a shot.

Talking about alternative completion lists, in Bash, I have a fondness for the short-cut **Alt + ?**. After Bash completes `file`, instead of pressing **Tab** key, I press **Alt + ?**, and Bash instantly displays a list of alternative completions.

```
xiaodong@codeland:~$ ls -l file
file1  file2  file3  file4  file5
```

There is another scenario where we want the shell to disable completion for certain file types. To achieve this, we can use the `FIGNORE` variable. In the following example, I want to view the contents of `Welcome.java`, so I only want the shell to complete `.java` files and exclude `.class` files.

```
xiaodong@codeland:~$ cat W
Welcome.class  Welcome.java
xiaodong@codeland:~$ cat Welcome.
```

After assigning the `.class` extension to the `FIGNORE` variable, the shell indeed excluded the `.class` file type for me.

```
xiaodong@codeland:~$ FIGNORE='.class'
xiaodong@codeland:~$ cat W<Tab>
xiaodong@codeland:~$ cat Welcome.java
```

If you want to exclude multiple file types, simply separate them with a colon (:). For example:

```
xiaodong@codeland:~$ FIGIGNORE='.o:.class'
```

This tells the shell to exclude `.o` and `.class` files from auto-completion. Both Bash and Zsh currently support `FIGIGNORE`.

Path name completion is similar to file name completion, but after completion, it automatically appends a `/` (slash) for easier navigation to the next level of the path. In the following example, while inputting

```
xiaodong@codeland:~$ cd g
```

Pressing **Tab** key again, the shell completes the full name, appending a `/` at the end.

```
xiaodong@codeland:~$ cd guessing_game/
```

Next, I continue to type `s` and press **Tab** key, this time the shell completes the subdirectory `src`.

```
xiaodong@codeland:~$ cd guessing_game/src/
```

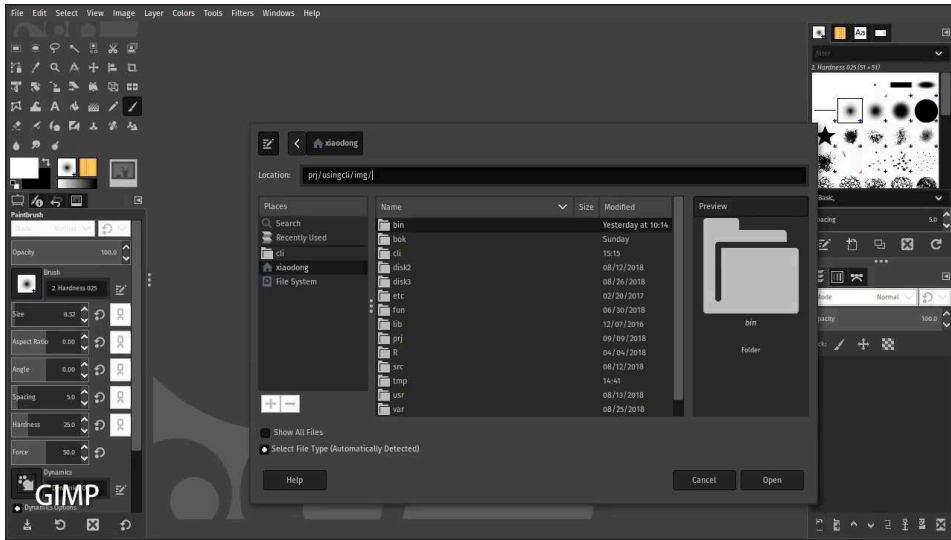
When the initial character is not unique, Bash has the same behavior as file name completion, where pressing **Tab** key displays a list of possible completions.

```
xiaodong@codeland:~$ cd h
hello/          hello_world/
xiaodong@codeland:~$ cd hello
```

Incidentally, auto-completion is not only useful in command-line interfaces but also in some graphical applications. For instance, I can use it to open files in GIMP, as shown in the following Figure 2.2.

2.4 Command Name and Program Name Auto-Completion

Command completion without options is almost the same as filename completion. Let's look at an example. If you've seen the movie "The Matrix", then the following scene should be very familiar. When I type



```
xiaodong@codeland:~$ cmat
```

What if I enter only `c` (the first character of a command)

```
xiaodong@codeland:~$ c
```

Next, I press the **Tab** key, and Bash asks me: “Display all 474 possibilities? (y or n)”. I press **y** to display them. If I press **n**, they will not be displayed.

```
xiaodong@codeland:~$ c
Display all 474 possibilities? (y or n)
```

Because the list of items available for auto-completion is too long to fit on one screen, Bash uses the **more** paginated viewer to display it. Now you can press **Space** to flip to the next page, or press **q** to exit, as shown in Figure 2.3.

Figure 2.3: Command auto-completion candidates

Besides automatically completing command names, the shell can also automatically complete a program’s subcommands, such as **status** in **git status**, as well as command options. However, Bash requires the installation of a separate **bash-completion** package, while Zsh does not need an additional package due to its built-in support for this feature.

The source code for bash-completion can be found on GitHub¹, where you can learn how to install and configure it.

In Debian or Ubuntu, this can be achieved by executing

```
xiaodong@codeland:~# apt install bash-completion
```

to install it.

In addition to installing `bash-completion` on Fedora, I recommend installing `bash-completion-extras` as well.

```
xiaodong@codeland:~# yum install bash-completion bash-completion-extras
```

On Arch Linux, you can input

```
xiaodong@codeland:~# pacman -S bash-completion
```

to perform the installation.

To configure `bash-completion`, simply add the following command to your `~/.bashrc` (personal) or `/etc/bash.bashrc` (system-wide) file.

```
[ -r /usr/share/bash-completion/bash_completion ] \
&& . /usr/share/bash-completion/bash_completion
```

Before using Zsh's command completion feature normally, we also need to add the following content to the `~/.zshrc` configuration file:

```
# completion
autoload -U compinit
compinit -i
```

Let's take a look at an example of a command option for auto-completion. I'm typing:

¹<https://github.com/scop/bash-completion>

```
xiaodong@codeland:~$ find -
```

Immediately press **Tab** key, and Bash will list the available auto-completion options. See Figure 2.4 for illustration.

```
xiaodong@codeland:~$ find -
-amin      -fls      -iwholename  -nowarn    -true
-anewer    -follow   -links       -ok         -type
-atime     -fprint   -lname       -okdir     -uid
-cmin      -fprint0  -ls          -path      -used
-cnewer    -fprintf  -maxdepth    -perm      -user
-context   -fstype   -mindepth    -print     -version
-ctime     -gid      -mmin        -print0    -warn
-daystart  -group    -mount       -printf    -wholename
-delete    -help     -mtime       -prune     -writable
-depth     -ignore_readdir_race -name        -quit      -xdev
-empty     -ilname   -newer       -readable  -xtype
-exec      -iname    -nogroup     -regex
-execdir   -inum     -noignore_readdir_race -regextype
-executable -ipath    -noleaf      -samefile
-false     -iregex   -nouser      -size
xiaodong@codeland:~$ find -
```

Figure 2.4: Alternate command completion list

I then type `ina`

```
xiaodong@codeland:~$ find -ina
```

And again, press the **Tab** key, the Bash will auto-complete the `-iname` option.

```
xiaodong@codeland:~$ find -iname
```

The following example demonstrates Bash autocompletion in action. Upon typing

```
xiaodong@codeland:~$ git in
```

After that, press **Tab** key, Bash provides a list of subcommands that can be automatically completed.


```
xiaodong@codeland:~$ git in
info      init      instaweb
```

Then enter i

```
xiaodong@codeland:~$ git ini
```

And when you press **Tab** key again, Bash will auto-complete the subcommand `init` for you. The process is the same for running the `git status` subcommand.

```
xiaodong@codeland:~$ git init
```

In contrast to Bash, Zsh provides a better user experience for command option completion. In the following example, you will see that Zsh not only lists the options available for completion but also provides a brief description of each option. As shown in Figure 2.5, you can also select these list items.

```
--ignore-backups      -B -- don't list entries ending with ~
--inode               -i -- print file inode numbers
--kilobytes           -k -- use block size of 1k
-l                   -- long listing
--literal             -N -- print entry names without quoting
-m                   -- comma separated
--no-group            -G -- inhibit display of group information
--numeric-uid-gid     -n -- numeric uid, gid
-o                   -- no group, long
--quote-name          -Q -- quote names
--quoting-style        -- specify quoting style
--recursive           -R -- list subdirectories recursively
--reverse             -r -- reverse sort order
-S                   -- sort by size
--si                  -- sizes in human readable form; powers of 1000
--size                -s -- display size of each file in blocks
--sort                -- specify sort key
-t                   -- sort by modification time
--tabsize             -T -- specify tab size
--time                -- specify time to show
--time-style           -- show times using specified style
-U                   -- access time
-U                   -- unsorted
-V                   -- sort by version (filename treated numerically)
--version             -- display version information
--width               -w -- specify screen width
-X                   -- sort horizontally
-X                   -- sort by extension
--dereference-command-line-symlink-to-dir  --show-control-chars
--group-directories-first

ls -l
```

Figure 2.5: Command option auto-completion in Zsh

For subcommands, Zsh provides the same level of completion as it does for command options.

Furthermore, subcommand and option completion can also be used together. In the following example, I first completed the subcommand `git status`, and then completed the option `--verbose`.

```
xiaodong@codeland:~$ git sta
stash -- stash away changes to dirty working directory
status -- show working-tree status
xiaodong@codeland:~$ git status --v
xiaodong@codeland:~$ git status --verbose
```

2.4.1 Zsh Auto-Suggestion Plugin

For Zsh users, I highly recommend a useful command auto-suggestion plugin called `zsh-autosuggestions`². This plugin borrows the auto-suggestion feature from the `fish` shell, allowing loyal Zsh enthusiasts to enjoy this functionality as well.

Installing `zsh-autosuggestions` is straightforward – simply clone it from GitHub to your local machine and refer to `zsh-autosuggestions.zsh` in your `.zshrc`, then reopen your terminal.

```
xiaodong@codeland:~$ git clone \
https://github.com/zsh-users/zsh-autosuggestions.git \
~/.zsh-autosuggestions
xiaodong@codeland:~$ echo source \
~/.zsh-autosuggestions/zsh-autosuggestions.zsh \
>> ~/.zshrc
xiaodong@codeland:~$ source ~/.zshrc
```

Below, let's take a look at how `zsh-autosuggestions` works. First, I will demonstrate the behavior without enabling the `zsh-autosuggestions` plugin. When typing commands like `ls -l` and `cd hello_world`, the only assistance provided is command completion, and there are no automatic suggestions. However, once the `zsh-autosuggestions` plugin is enabled, upon typing `ls`, a grayed-out suggestion `-la` appears automatically. This is because Zsh recalls that I previously entered the command `ls -la`, prompting the autosuggestion. This is illustrated in Figure 2.6.

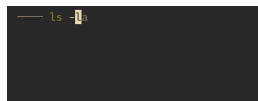


Figure 2.6: Command auto-suggestions in Zsh

At this point, we have two options: one is to press \rightarrow (**right arrow key**) to accept the suggestion, and the other is to continue entering new content, thereby abandoning the suggestion. After entering `cd h`, Zsh also provides automatic suggestions `ello_world`.

²<https://github.com/zsh-users/zsh-autosuggestions>

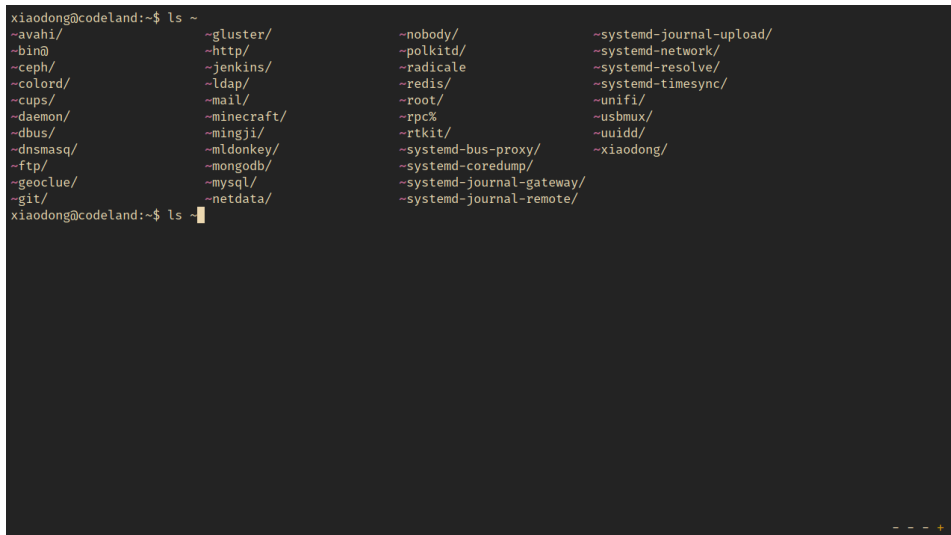
2.5 Username, Hostname, and Variable Name Auto-Completion

In addition to the common filename and command completion, shell auto-completion also supports other types of completion, fully demonstrating the versatility of shell auto-completion. Let's take a look at how shell auto-completes usernames.

When I input

```
xiaodong@codeland:~$ ls ~
```

Following that, I pressed **Tab** key, and Bash presented me with a list of existing usernames on the system. As shown in Figure 2.7.



```
xiaodong@codeland:~$ ls ~
~avahi/      ~gluster/      ~nobody/      ~systemd-journal-upload/
~bin/        ~http/         ~polkitd/     ~systemd-network/
~ceph/       ~jenkins/      ~radicale/    ~systemd-resolve/
~colord/     ~ldap/         ~redis/       ~systemd-timesync/
~cups/       ~mail/         ~root/        ~unifi/
~daemon/     ~minecraft/    ~rpc%         ~usbmux/
~dbus/       ~mingji/       ~rtkit/       ~uidd/
~dnsmasq/    ~mldonkey/     ~systemd-bus-proxy/ ~xiaodong/
~ftp/        ~mongodb/      ~systemd-coredump/
~geoclue/    ~mysql/        ~systemd-journal-gateway/
~git/        ~netdata/      ~systemd-journal-remote/
xiaodong@codeland:~$ ls ~
```

Figure 2.7: Username auto-completion candidates in Bash

I typed **x** and hit **Tab** again, and Bash automatically completed the username **xiaodong**.

```
xiaodong@codeland:~$ ls ~x<Tab>
xiaodong@codeland:~$ ls ~xiaodong/
```

In Zsh, we can see that, unlike Bash, the username completion list has a slightly different appearance. Bash includes the **~** prefix and a trailing **/** (forward slash).

2.5. USERNAME, HOSTNAME, AND VARIABLE NAME AUTO-COMPLETION²⁷

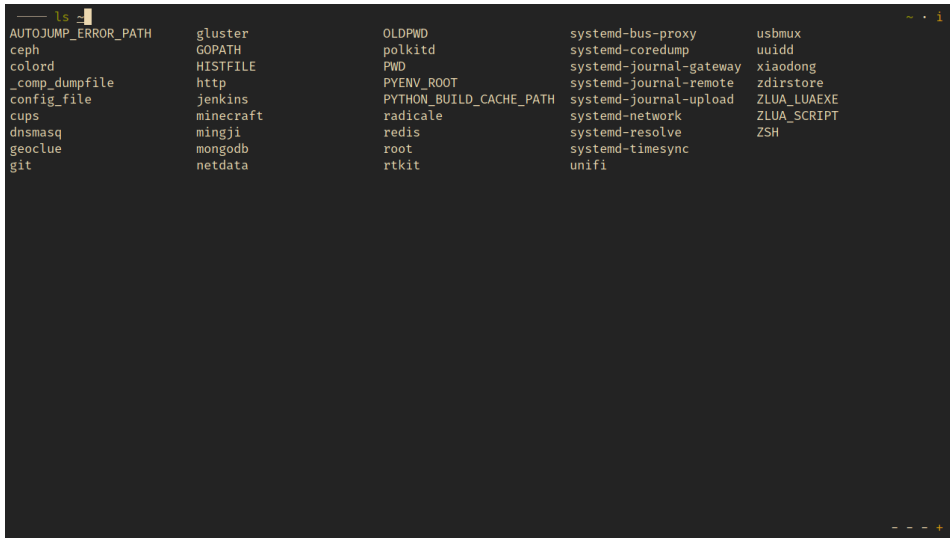


Figure 2.8: Username auto-completion candidates in Zsh

Zsh, on the other hand, only shows the username itself. This is illustrated in the following Figure 2.8.

If you frequently use `ssh` to log in to remote machines, hostname auto-completion will be a helpful feature. Let's take a look at an example. When I type

```
xiaodong@codeland:~$ ssh xiaodong@l
```

After pressing **Tab** key, Bash displays a list of hostnames that can be auto-completed.

```
xiaodong@codeland:~$ ssh xiaodong@l
xiaodong@lab.github.com          xiaodong@localhost
xiaodong@linuxtoy.org           xiaodong@localhost.localdomain
xiaodong@codeland:~$ ssh xiaodong@l
```

I then type `i` and press **Tab** key, and this time Bash automatically completes the full hostname `linuxtoy.org`. Alternatively, you can directly press **Tab** key after `@` to display a list of all hostnames.

```
xiaodong@codeland:~$ ssh xiaodong@li<Tab>
xiaodong@codeland:~$ ssh xiaodong@linuxtoy.org
```

Not only hostnames, but IP addresses can also be automatically completed. Alternatively, typing `ssh 1` and pressing **Tab** key allows Bash to auto-complete the hostname.

```
xiaodong@codeland:~$ ssh 1
lab.github.com          linuxtoy.org            localhost
```

By now, you may be wondering where Bash finds these hostnames to auto-complete. Well, one source is the contents of the `/etc/hosts` file, and the other is the `ssh` configuration file, such as `~/.ssh/config`. As shown in Figure 2.9, if you plan to let Bash auto-complete frequently used hostnames, you might consider adding them to these two files. Additionally, the `~/.ssh/known_hosts` file is also included. Any hosts that you have logged into via `ssh` will be included in this file.

```
0 #
1 # /etc/hosts: static lookup table for host names
2 #
3 #
4 #ip-address: <hostname domain.org> <hostname>
5 127.0.0.1 localhost.localdomain localhost
6 ::1 localhost.localdomain localhost
7
8 ##92.168.1.101 port.toyland.org toyland
9
10
```

`/etc/hosts`

```
0 # Host
1 #
2 ControlMaster no
3 ControlPath /tmp/ssh_mux_ah.sp.sp
4 ServerAliveInterval 90
5 ServerAliveCountMax 5
6 StrictHostKeyChecking no
7
8 #Host linuxtoy.org
9 #
10 #Hostname
11 #UseRout
12 #Port 22
13
```

`~/.ssh/config`

Figure 2.9: Sources of auto-completion hostname

The automatic completion feature for hostnames in Zsh is similar to Bash and will not be discussed here.

Finally, let's take a look at automatic completion of variable names. When I input

```
xiaodong@codeland:~$ echo $
```

Press **Tab** key again and then press **y** when prompted, and Bash will display all available variable names that can be completed. As shown in Figure 2.10.

I then continue typing and press the **Tab** key to auto-complete the `BASH_VERSION` variable.

2.5. USERNAME, HOSTNAME, AND VARIABLE NAME AUTO-COMPLETION²⁹

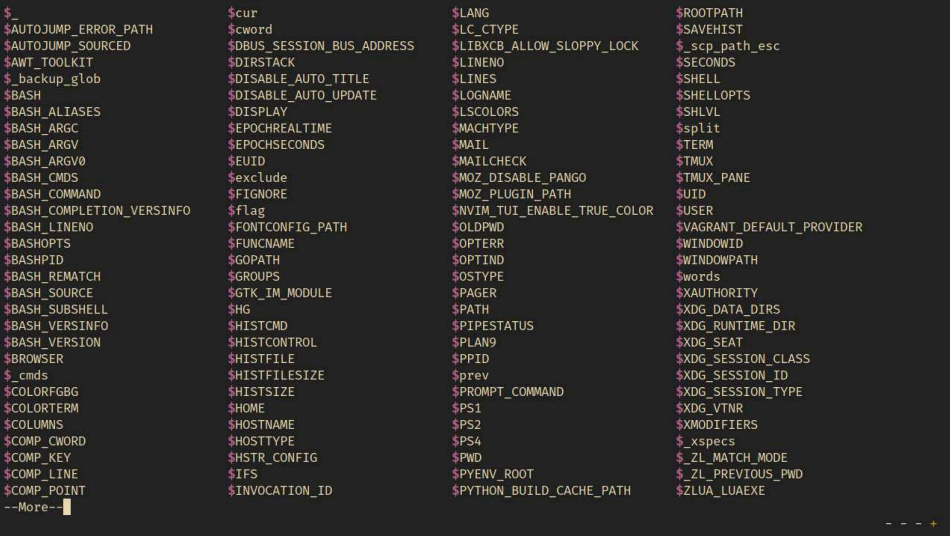


Figure 2.10: List of possible variable name completions in Bash

```
xiaodong@codeland:~$ echo $BASH_VERSION
```

The auto-completion of variables in Zsh is similar to that in Bash. However, on my system, it provides more completion options than Bash, as shown in Figure 2.11.

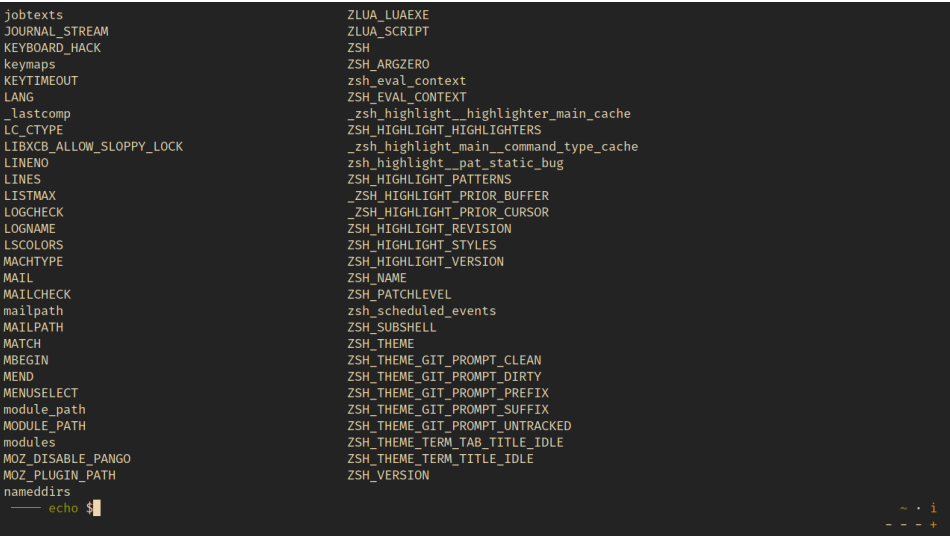


Figure 2.11: Variable name auto-completion suggestions in Zsh

Overall, these completion types differ slightly from the file name and command

name autocompletion we discussed earlier, as they are accompanied by a special prefix character, as shown in Table 2.1.

Table 2.1: Username, hostname, and variable name auto-completion prefix characters

Prefix character	Auto-completion type
~	Username
@	Hostname
\$	Variable name

2.6 Programmable Auto-Completion

Now that you are familiar with the usage of command line auto-completion, you may wonder as a developer, “How can I add command completion for my own programs or scripts?” Fortunately, leveraging the programmable completion features provided by Bash and Zsh, we can easily customize command line completion. Let’s dive into the details with an example.

2.6.1 Bash Example

Suppose I have written a program named `mycmd` with two command line options, `--help` and `--version`. Let’s see how command completion works for it. When I type

```
xiaodong@codeland:~$ mycmd -
```

And after pressing the **Tab** key, Bash presents me with a complete list of options for the command and auto-completes it to `mycmd --`.

```
xiaodong@codeland:~$ mycmd -
--help      --version
xiaodong@codeland:~$ mycmd --
```

I then typed `h` followed by pressing **Tab** key, and Bash automatically completed the option `--help`. Ah-ha, that was exactly the kind of command completion I was looking for. So, how can we achieve programmable completion?