BGGN 213

Working with UNIX

Barry Grant

UC San Diego

http://thegrantlab.org/bggn213

Recap From Last Time:

- **Motivation**: Why we use UNIX for bioinformatics. Modularity, Programmability, Infrastructure, Reliability and Unix Philosophy
- **Shell advantages**: Makes your work less error-prone, more reproducible and less boring allowing you to automate repetitive tasks and concentrate on more exciting things.
- **Key commands**: Introduced the 21 key unix commands that you will use during ~95% of your future unix work
- **Jetstream**: Many bioinformatic tasks require large amounts of computing power and can't realistically be run on your own machine. These tasks are best performed using remote computers or cloud computing, which can only be accessed through a shell.

Basics	File Control	Viewing & Editing Files	Misc. useful	Power commands	Process related
ls	mv	less	chmod	grep	top
cd	cp	head	echo	find	ps
pwd	mkdir	tail	wc	sed	kill
man	rm	nano	curl	sudo	Crl-c
ssh	| (pipe)	touch	source	git	Crl-z
scp	> (write to file)		cat	R	bg
	< (read from file)		tmux	python	fg

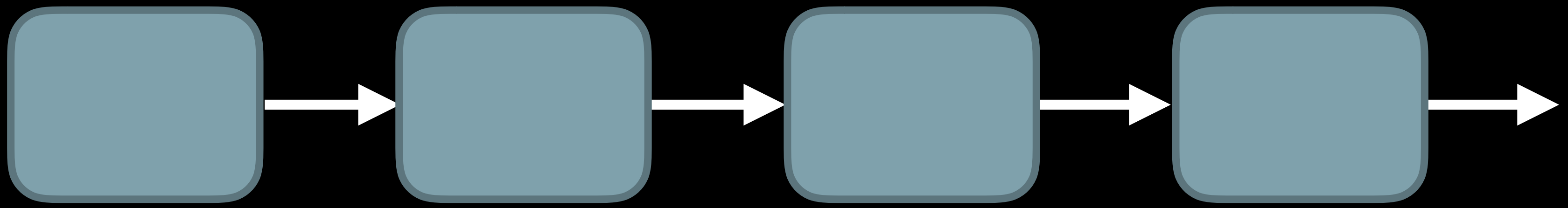
Basics	File Control	Viewing & Editing Files	Misc. useful	Power commands	Process related
ls	mv	less	chmod	grep	top
cd	cp	head	echo	find	ps
pwd	mkdir	tail	wc	sed	kill
man	rm	nano	curl	sudo	Crl-c
ssh	| (pipe)	touch	source	git	Crl-z
scp	> (write to file)		cat	R	bg
	< (read from file)		tmux	python	fg

Working with Unix

How do we actually use Unix?

Combining Utilities with **Redirection** (**>**, **<**) and **Pipes** (**|**)

- The power of the shell lies in the ability to combine simple utilities (*i.e.* commands) into more complex algorithms very quickly.



- A key element of this is the ability to send the output from one command into a file or to pass it directly to another program.

- This is the job of >, < and |

Side-Note: **Standard Input** and **Standard Output** streams

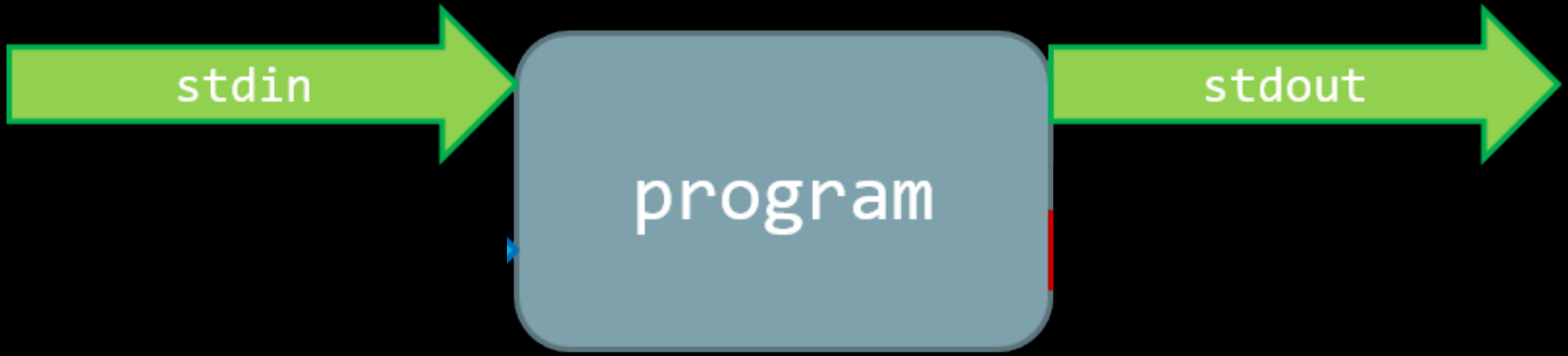
Two very important concepts that unpin Unix workflows:

- Standard Output (**stdout**) - default destination of a program's output. It is generally the terminal screen.

- Standard Input (**stdin**) - default source of a program's input. It is generally the command line.

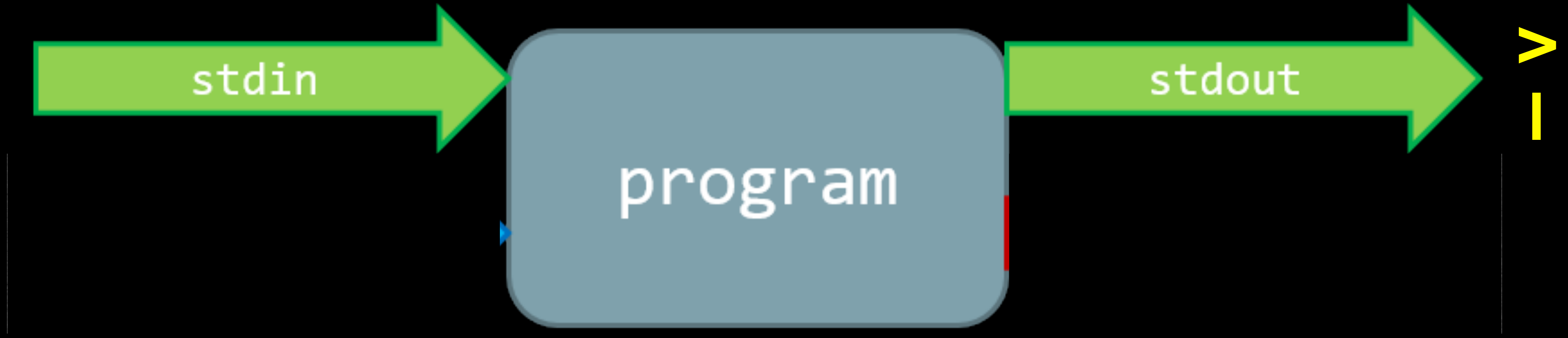
Output redirection and piping

Do it Yourself!



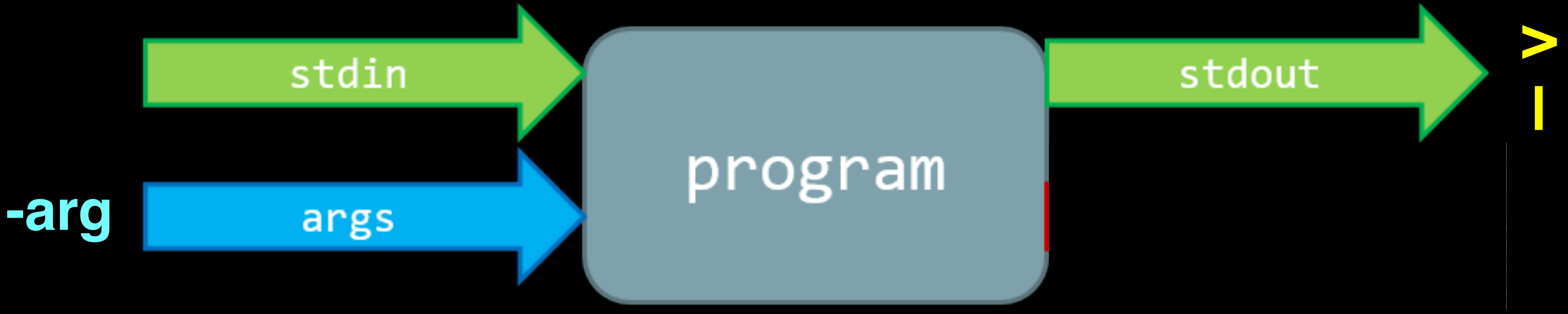
```
> ls ~/Desktop  # stdin is “~/Desktop”; stdout to screen
```

Output redirection and piping



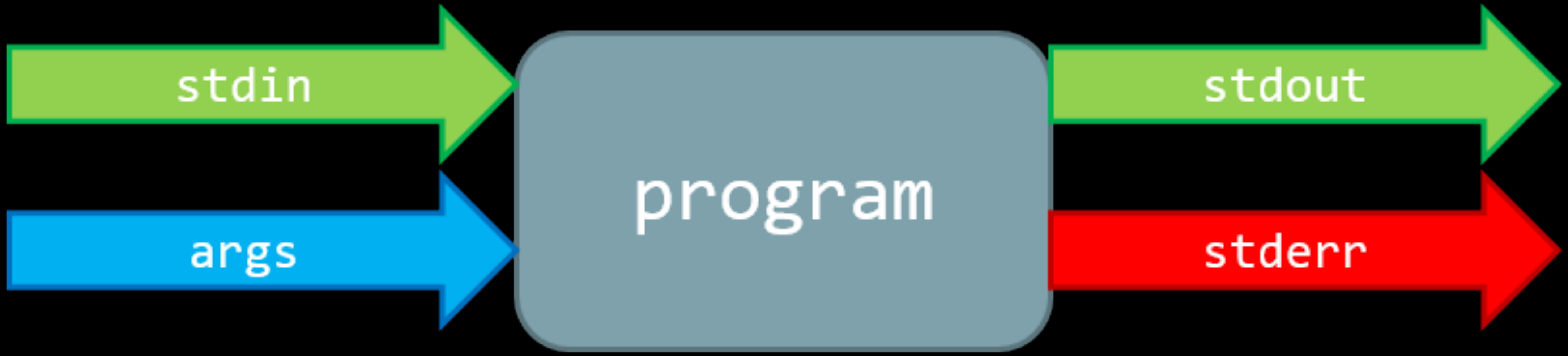
```
> ls ~/Desktop  # stdin is "~/Desktop"; stdout to screen

> ls ~/Desktop > mylist.txt  # stdout redirected to file

> ls ~/Desktop | less  # sdout piped to less (no file created)
```

Output redirection and piping



```
> ls ~/Desktop  # stdin is "~/Desktop"; stdout to screen

> ls ~/Desktop > mylist.txt  # stdout redirected to file

> ls ~/Desktop | less  # sdout piped to less (no file created)

> ls -l ~/Desktop  # extra optional input argument "-l"
```

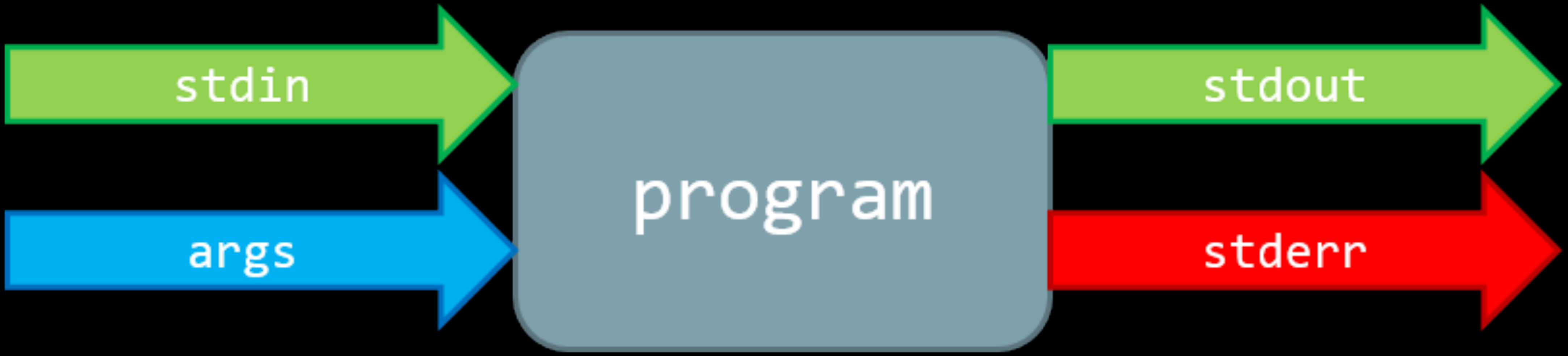
Output redirection and piping



```
> ls ~/Desktop  # stdin is "~/Desktop"; stdout to screen

> ls ~/Desktop > mylist.txt  # stdout redirected to file

 > ls ~/Desktop l less  # sdout piped to less (no file created)

> ls /nodirexists/ 2> binlist.txt  # stderr to file
```

Do it Yourself!

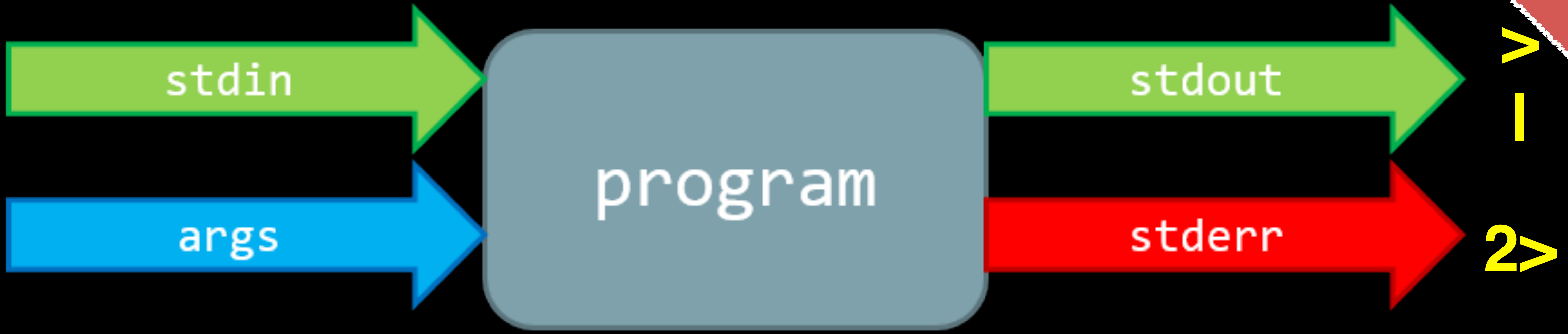
Output redirection and piping



```
> ls ~/Desktop  # stdin is "~/Desktop"; stdout to screen

> ls ~/Desktop > mylist.txt  # stdout redirected to file

> ls ~/Desktop | less  # sdout piped to less (no file created)

> ls /nodirexists/ 2> binlist.txt  # stderr to file
```

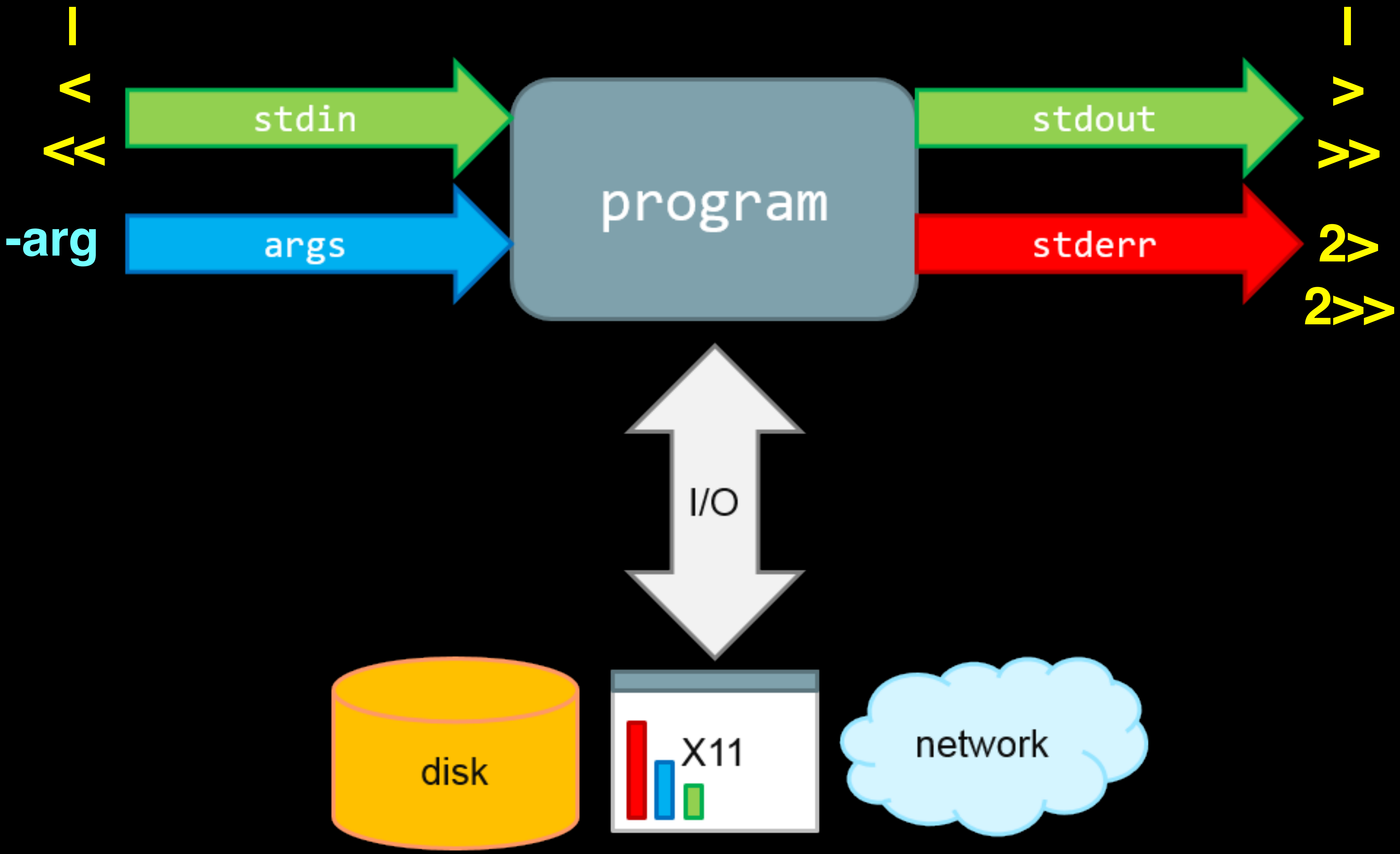
Output redirection and piping

Do it Yourself!

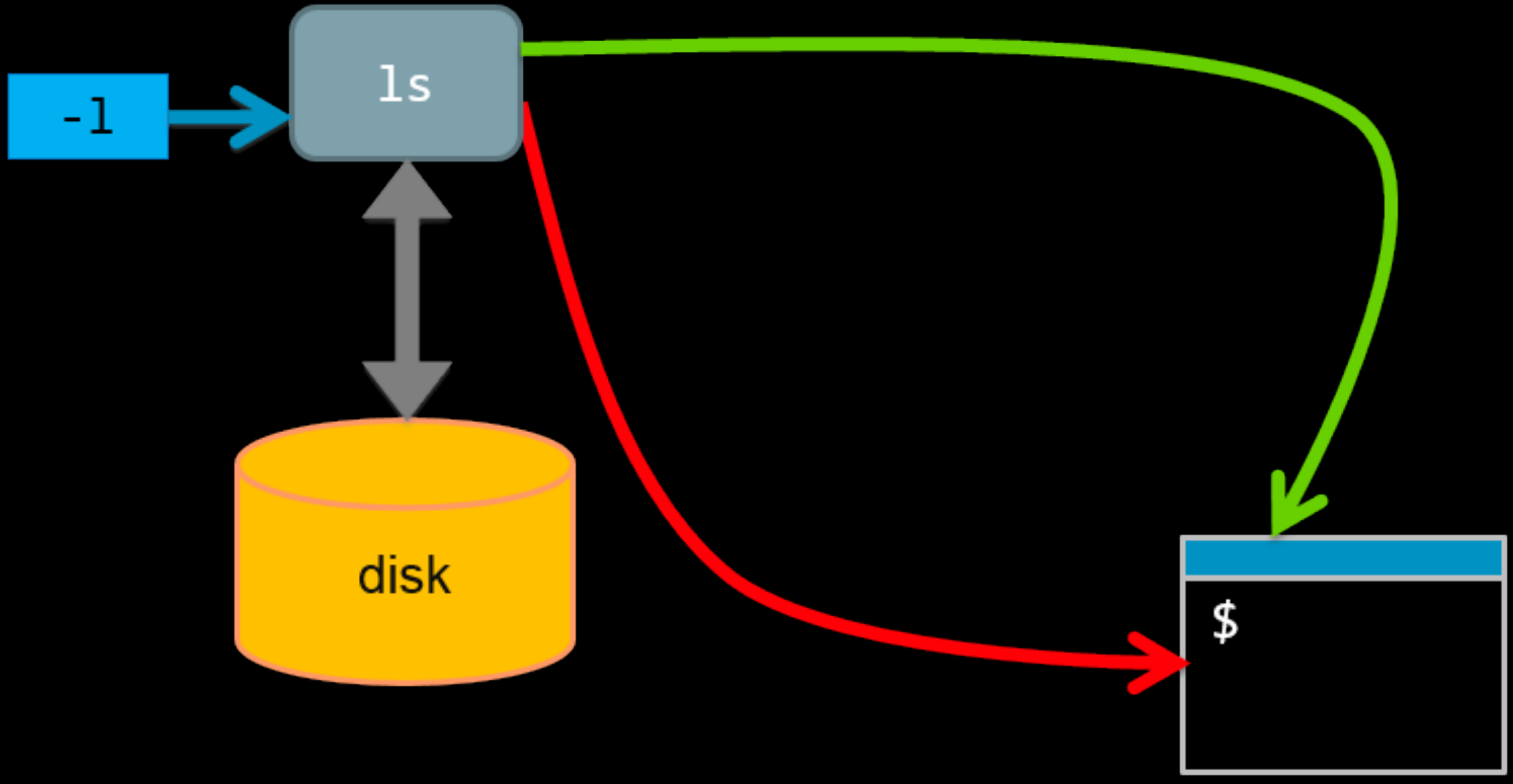


```
> ls ~/Desktop  # stdin is "~/Desktop"; stdout to screen

> ls ~/Desktop > mylist.txt  # stdout redirected to file

 > ls ~/Desktop | less  # sdout piped to less (no file created)

> ls /nodirexists/ 2> binlist.txt  # stderr to file
```

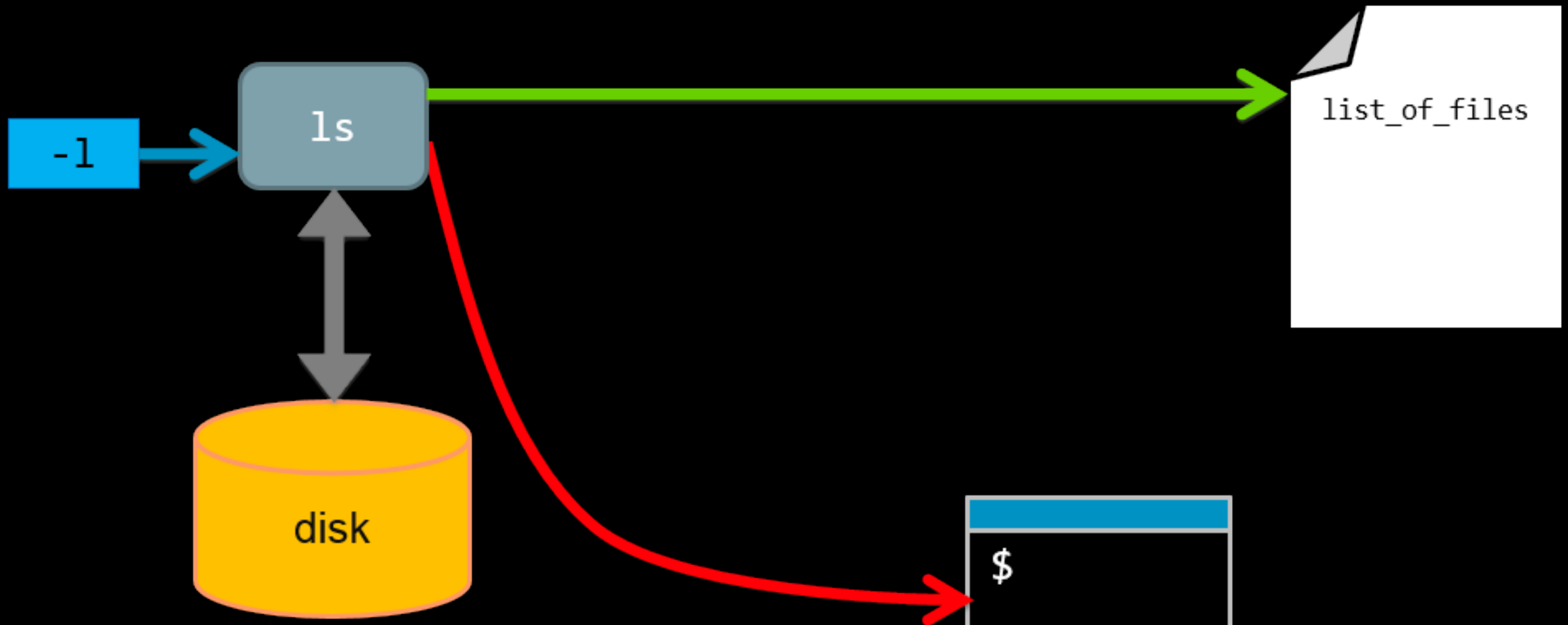
Output redirection summary



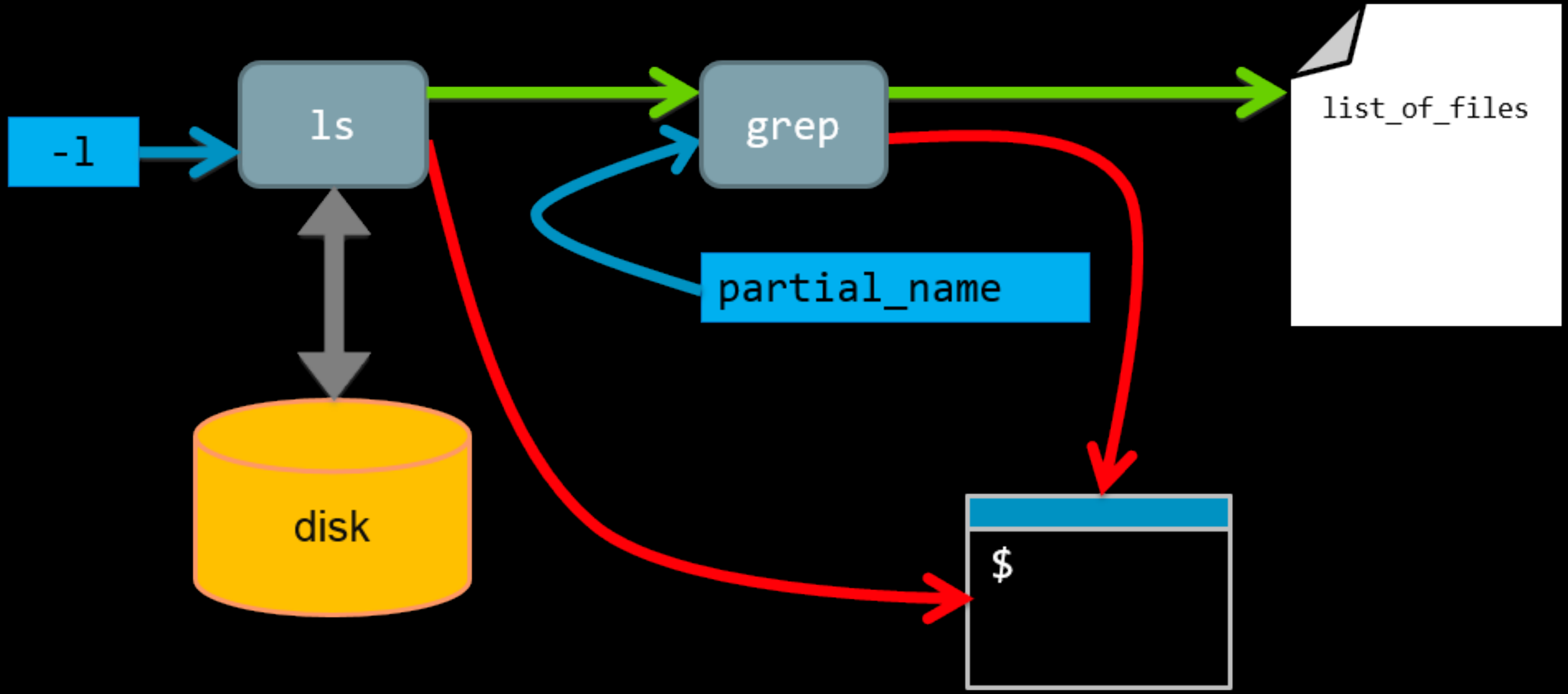
ls -l



ls -l > list_of_files



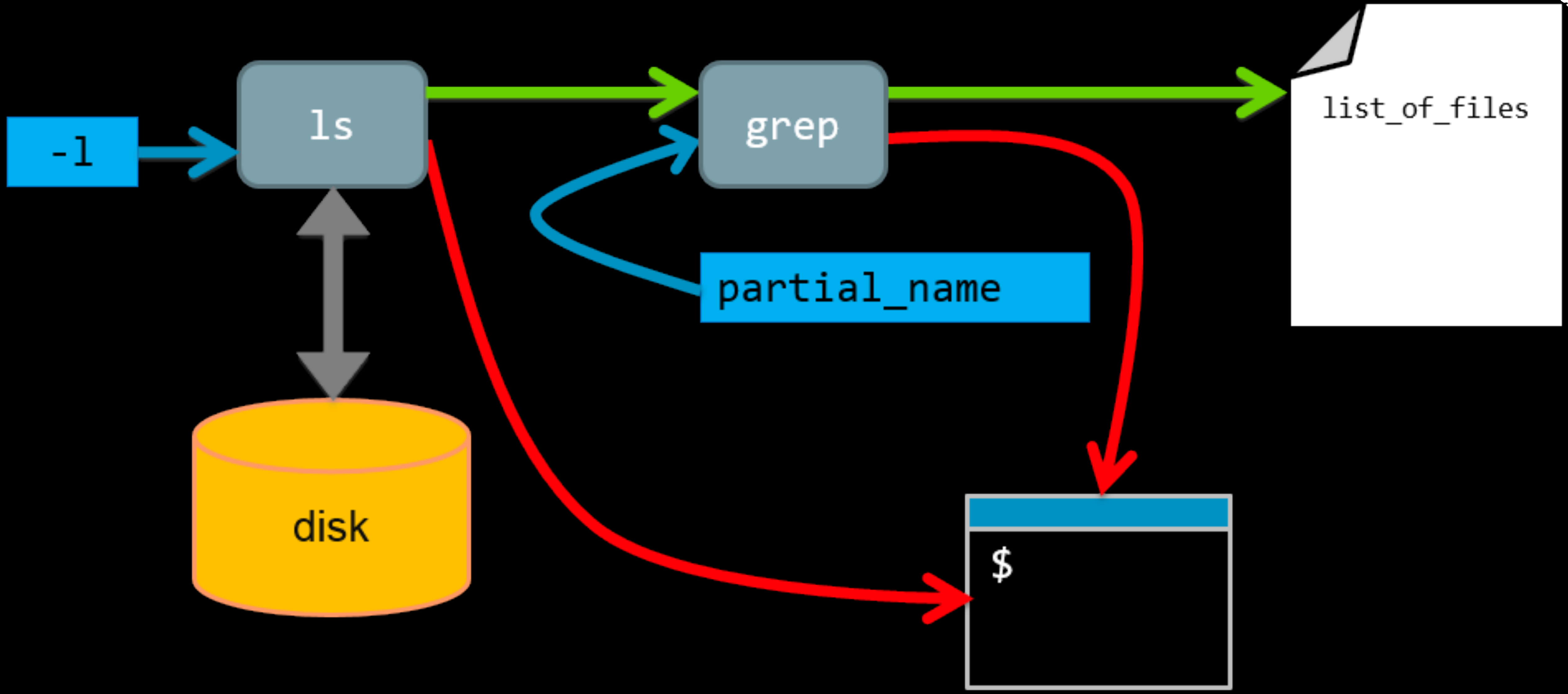
ls -l | grep partial_name > list_of_files



We have piped (|) the stdout of one command into the stdin of another command!

Do it Yourself!

ls -l /usr/bin/ | grep “tree” > list_of_files



grep: prints lines containing a string.
Also searches for strings in text files.

Basics	File Control	Viewing & Editing Files	Misc. useful	Power commands	Process related
ls	mv	less	chmod	grep	top
cd	cp	head	echo	find	ps
pwd	mkdir	tail	wc	sed	kill
man	rm	nano	curl	sudo	Crl-c
ssh	| (pipe)	touch	source	git	Crl-z
scp	> (write to file)		cat	R	bg
	< (read from file)		tmux	python	fg

Do it Yourself!

Side-Note: **grep** ‘power command’

- **grep** - prints lines containing a string pattern. Also searches for strings in text files, e.g.

```
> grep --color "GESGKS" sequences/data/seqdump.fasta
```

```
REVKLLLLGAGESGKSTIVKQMKIIHEAGYSEEECKQYK
```

- grep is a ‘power tool’ that is often used with pipes as it accepts **regular expressions** as input (e.g. **“G..GK[ST]”**) and has lots of useful options - see the *man page* for details.

grep example using regular expressions

Do it Yourself!

- Suppose a program that you are working with complains that your input sequence file contains non-nucleotide characters. You can eye-ball your file or …

```
> grep -v "^>" seqdump.fasta | grep --color "[^ATGC]"
```

Exercises:

(1). Use "man grep" to find out what the **-v** argument option is doing!

(2). How could we also show line number for each match along with the output?

(tip you can grep the output of "man grep" for 'line number')

grep example using regular expressions

Do it Yourself!

- Suppose a program that you are working with complains that your input sequence file contains non-nucleotide characters. You can eye-ball your file or …

```
> grep -v "^>" seqdump.fasta | grep --color -n "[^ATGC]"
```

- First we remove (with **-v** option) lines that start with a “>” character (these are sequence identifiers).

- Next we find characters that are *not* A, T, C or G. To do this we use **^** symbols second meaning: *match anything but* the pattern in square brackets. We also print line number (with **-n** option) and color output (with **--color** option).

Key Point: Pipes and redirects <u>avoid</u> unnecessary i/o

- Disc i/o is often a bottleneck in data processing!
- Pipes prevent unnecessary disc i/o operations by connecting the stdout of one process to the stdin of another (these are frequently called “**streams**”)

```
> program1 input.txt 2> program1.stderr | \
program2 2> program2.stderr > results.txt
```

- Pipes and redirects allow us to build solutions from modular parts that work with **stdin** and **stdout streams**.

Unix ‘Philosophy’ Revisited

“Write programs that do one thing and do it well. Write programs to work together and that encourage open standards. Write programs to handle text streams, because that is a universal interface.”

— Doug McIlory

Pipes provide speed, flexibility and sometimes simplicity…

- In 1986 "*Communications of the ACM magazine*" asked famous computer scientist Donald Knuth to write a simple program to count and print the *k* most common words in a file alongside their counts, in descending order.

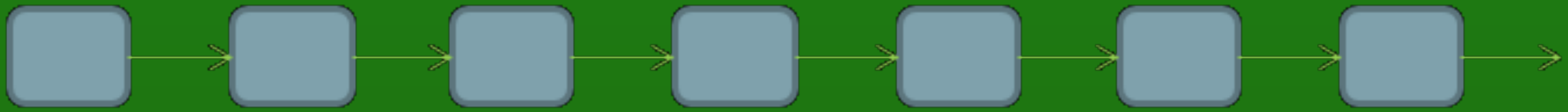
- Kunth wrote a literate programming solution that was 7 pages long, and also highly customized to this problem (e.g. Kunth implemented a custom data structure for counting English words).

- Doug McIlroy replied with one line:

```
> cat input.txt | tr A-Z a-z |  sort |  uniq -c |  sort -rn |  sed 10q
```

Key Point:

You can chain any number of programs together to achieve your goal!



This allows you to build up fairly complex workflows within one command-line.

Do it Yourself!

Shell scripting

```
#!/bin/bash
# This is a very simple hello world script.
echo "Hello, world!”
```

Exercise:

- Create a "Hello world"-like script using command line tools and execute it.
- Copy and alter your script to redirect output to a file using **>** along with a list of files in your home directory.
- Alter your script to use **>>** instead of **>**. What effect does this have on its behavior?

Do it Yourself!

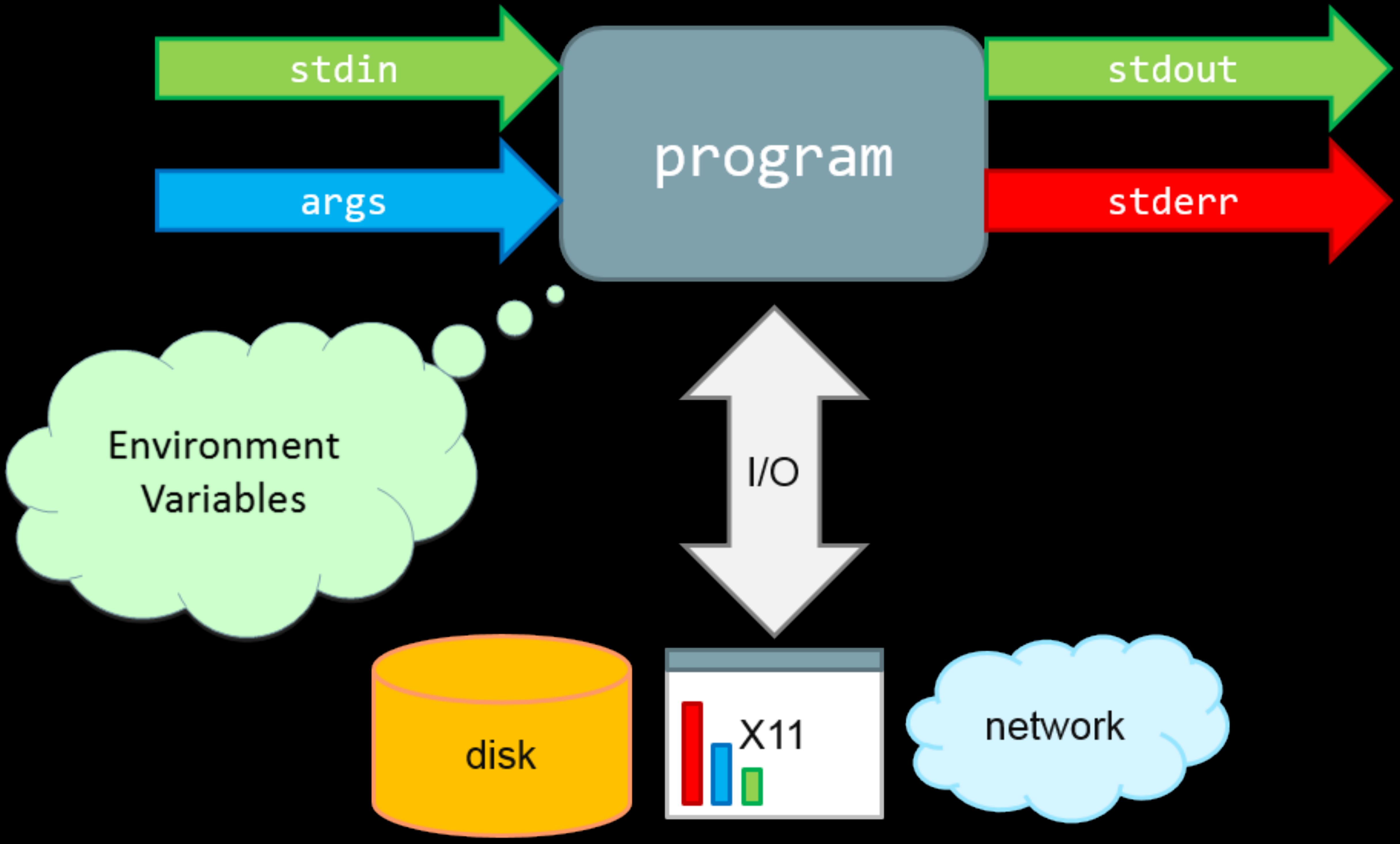
Variables in shell scripts

```
#!/bin/bash
# Another simple hello world script
message='Hello World!'
echo $message
```

- “message” - is a **variable** to which the string 'Hello World!' is assigned
- echo - prints to screen the contents of the variable "$message"

See #6 @ https://swcarpentry.github.io/shell-novice/

Side-Note: Environment Variables



$PATH ‘special’ environment variable

- What is the output of this command?

```
> echo $PATH
```

- Note the structure: <path1>:<path2>:<path3>

- **PATH** is an environmental variable which the shell uses to search for commands typed on the command line without a full path.

- *Exercise*: Use the command **env** to discover more.

Summary

- Built-in unix shell commands allow for easy data manipulation (e.g. sort, grep, etc.)

- Commands can be easily combined to generate flexible solutions to data manipulation tasks.

- The unix shell allows users to automate repetitive tasks through the use of shell scripts that promote reproducibility and easy troubleshooting

- Introduced the 21 key unix commands that you will use during ~95% of your future unix work…

Basics	File Control	Viewing & Editing Files	Misc. useful	Power commands	Process related
ls	mv	less	chmod	grep	top
cd	cp	head	echo	find	ps
pwd	mkdir	tail	wc	sed	kill
man	rm	nano	curl	sudo	Crl-c
ssh	| (pipe)	touch	source	git	Crl-z
	> (write to file)		cat	R	bg
	< (read from file)			python	fg

Do it Yourself!

Hands-on time

Using Jetstream for Bioinformatics

- Running command-line BLAST
- Ortholog mapping (running large jobs)
- Visualizing results with R/RStudio

New commands

sudo	Execute a command with root permissions
apt-get	Package handling utility for updating & installing software
curl	Download data
gunzip	File compression and decompression
blastp	Command line BLAST
shmlast	Mapping orthologs from RNA-seq data

How to Get Working

Best practices for organizing your computational biology projects

Read: Noble *PLoS Comp Biol* (2009)

- "A Quick Guide to Organizing Computational Biology Projects"

http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1000424

All files and directories used in your project should live in a single project directory.

- Use sub-directories to divide your project into sub-projects.
- Do not use spaces in file and directory names!

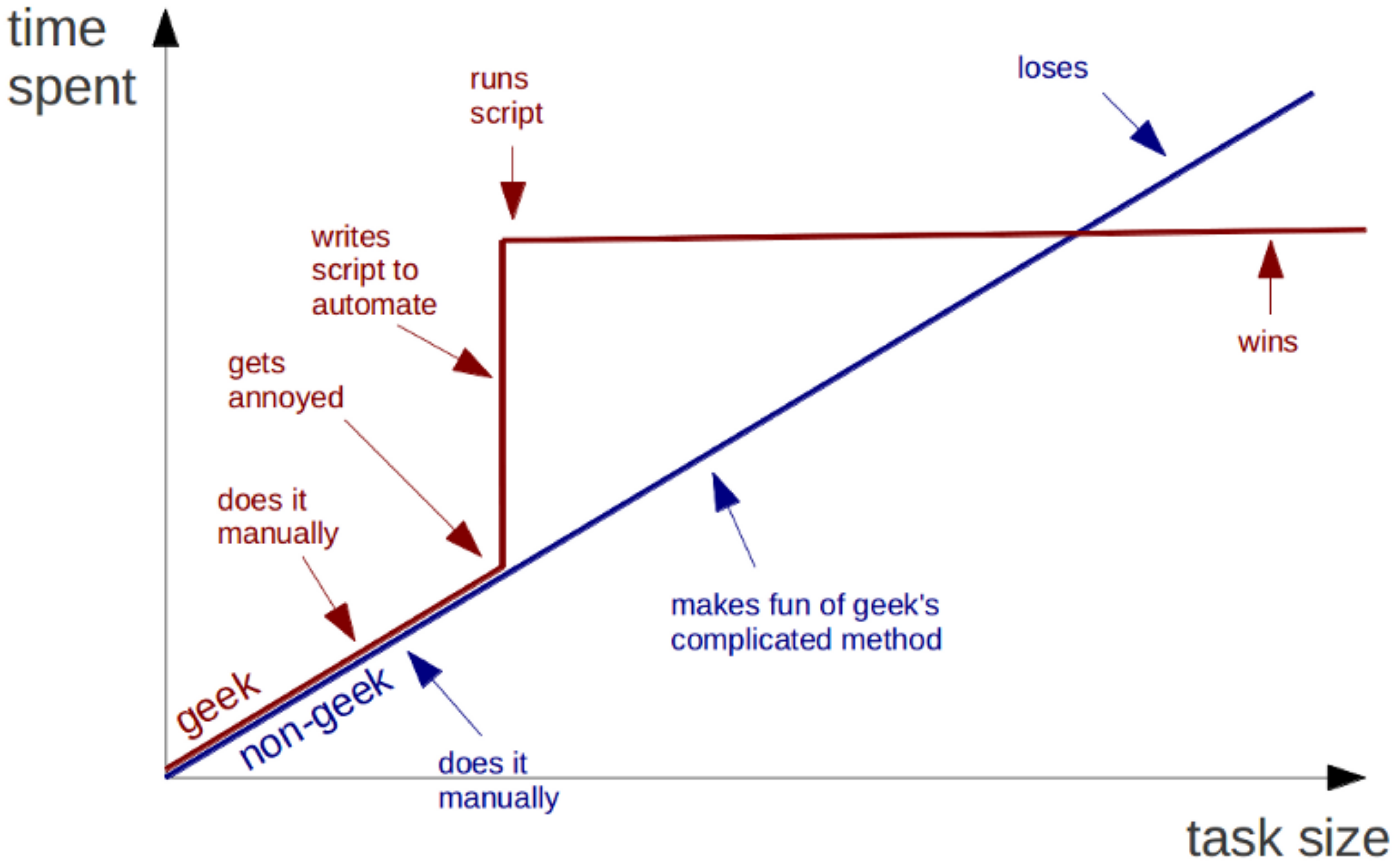
Document your methods and workflows with plain text README files

- Also document the origin of all data in your project directory
- Also document the versions of the software that you ran and the options you used.
- Consider using Markdown for your documentation.

Use version control and backup to multiple destinations!

Be reproducible:
http://ropensci.github.io/reproducibility-guide/sections/introduction/

Geeks and repetitive tasks


time spent
runs script
loses
writes script to automate
gets annoyed
wins
does it manually
makes fun of geek's complicated method
geek
non-geek
does it manually
task size