# A quick introduction to shell basics

Martin L. Purschke

# Later in my RCDAQ presentation I will show you this

```
$ rcdaq_client load librcdaqplugin_drs.so

$ rcdaq_client create_device device_drs -- 1 1001 0x21 -150 negative 140 3
```

You will see that we are using commands typed in a terminal application ALL THE TIME.

A terminal application is what runs the **shell**

A shell is taking your commands and acts on them (it's by far not the only program that does that)

I consider myself a master of the shell (and yet, I still learn things from others all the time)

The shell makes many repetitive tasks a breeze

And it is a lot faster to work with than a graphical environment

# Why you need to know the shell.

- You usually spend a lot of time using the shell
- If not so far, then you will during the exercises!
- The shell is the “face” of any Unix system, all attempts to make it like Windows aside
- You can do magic with a few lines of script that would take hours otherwise
- You can automate complex tasks with ease
- You can even automate your job to a large extent
- Let me show you in a few minutes.

**If you are a physicist, an engineer, a professional in a technical field and do not use the shell, you are missing out!**

# You need to accomplish some really complex task on the computer…

… is there a ready-made tool that does exactly that?

Probably not!

But each task breaks down into small steps.

You already have a myriad of “small” tools available that do a particular thing really, really well.

Think: You want to build a table. Do you have a tool that makes a table? No. But you have a hammer, a saw, screwdrivers, a drill… You use those generic tools to make your table

You use them one by one and end up with something that does exactly what you wanted/needed

Sounds too abstract? Here is an example…

# “I want to tag my pictures with the date they were taken!”

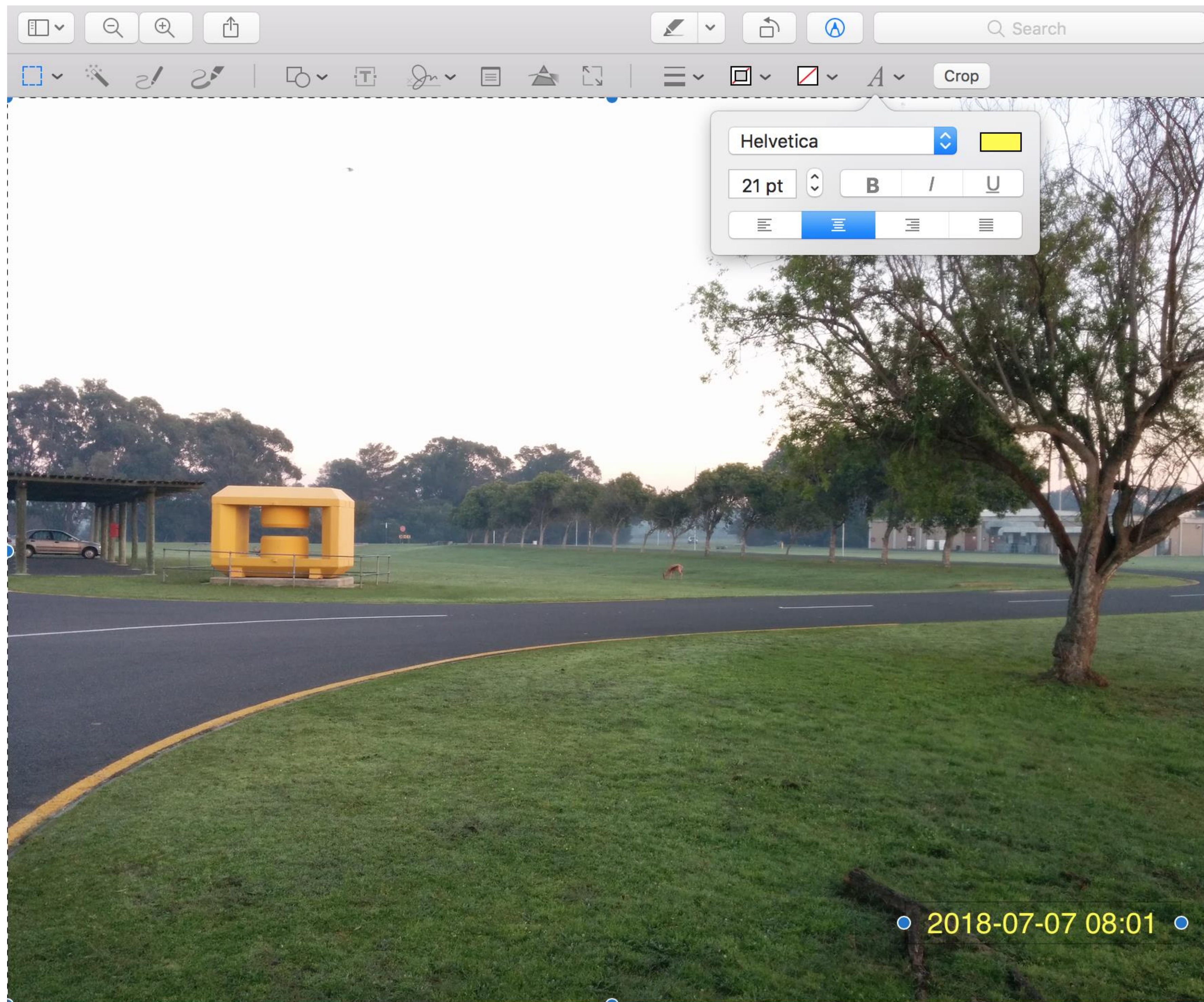
Figure out when a picture was taken and add some text on the right lower end:

That’s easy to do.

Right? *Right?*

Wait…

# Is it easy? You decide…




That's a lot of steps (I left several steps out)

But, yes, it gets the job done.

But wait! Now I have 1000 such pictures!

Does this really scale?

I'm going to be  at it all week!

I'll get (or buy) a program to do just that!

# No shortage of programs you can get buy to do that…

It adds some information to the picture

Formats the appearance in 20 or so different styles to choose from

And it costs you $30!!!

And what happens if you want a different style?

A different font?

Maybe you want the text on the left side?

A date formatted in a particular way? US/European/???

Maybe you want to add something completely different?

No monolithic tool can foresee all possible use cases the user might want

Even the most advanced program is limited in the end (and the really good ones cost money!)

**And that's we are getting to the shell and the tools.**

# In the shell you are stringing together small tools…

- No single “tool” needs to do all you want all on its own
- Each small tool is really, really good at a particular thing, such as:
- You have a tool to extract any information from a picture (data, geotags, lens info, ….)
- You have a tool to format the information in whatever format you need
- You have a tool to superimpose a ready-made string on a picture in whatever way you like ( position, size, color, transparency, font, background, ….)

- You string together small individual tools that are great at one particular thing
- No one utility needs to excel at all required tasks
- And this buys you the ultimate flexibility to accomplish the most esoteric tasks
- And not just what the designer of an all-integrated tool envisioned!

- **And most important: you can apply the same operation to thousands of images easily**

# We are getting way ahead of ourselves but I want to show you…

```
$ exiftool  IMG_20180707_080137.jpg
ExifTool Version Number         : 10.94
File Name                       : IMG_20180707_080137.jpg
Directory                       : .
File Size                       : 2.2 MB
File Modification Date/Time     : 2018:07:07 03:06:05-04:00
File Type                       : JPEG
File Type Extension             : jpg
MIME Type                       : image/jpeg
JFIF Version                    : 1.01
Exif Byte Order                 : Big-endian (Motorola, MM)
Make                            : LGE
Camera Model Name               : Nexus 5
Orientation                     : Horizontal (normal)
. . .
Date/Time Original              : 2018:07:07 08:01:37
Create Date                     : 2018:07:07 08:01:37
Shutter Speed Value             : 59.7
Aperture Value                  : 2.4
Flash                           : No Flash
Focal Length                    : 4.0 mm
. . .
GPS Latitude                    : 34 deg 1' 29.08" S
GPS Longitude                   : 18 deg 42' 59.24" E
GPS Position                    : 34 deg 1' 29.08" S, 18 deg 42' 59.24" E
```

By the way: If you upload such a picture from your camera to, say, Facebook, you give away all this information!

Strip this out!

This is what we actually want!

```
$ exiftool  -DateTimeOriginal -d "%Y-%m-%d %H:%M" -S IMG_20180707_080137.jpg | sed 's/DateTimeOriginal: //'
2018-07-07 08:01
```

# This is what saved me $30 ☺

```
#! /bin/sh

PIC="$1"
[ -z "$PIC"  ] && exit

DEST="$2"
[ -z "$DEST" ] && exit

DATE=$(exiftool  -DateTimeOriginal -d "%Y-%m-%d %H:%M" -S "$PIC" | sed 's/DateTimeOriginal: //')

NAME=$(basename $PIC)
NAME="$DEST/$NAME"
echo "new image = $NAME"

convert  $PIC -fill yellow  -pointsize 80  -undercolor '#00000080' -annotate +2200+2200  "$DATE"  $NAME
```

## This was just to get you into the right mindset…

You don't need to become a master at image processing, that was not the point…

I just wanted to show you that those tools act just like your hammer, saw, screwdriver, drill…

We will now do simple things with and in the shell.

# What the shell looks like

Normally you see a shell within a terminal window

In my younger days, a terminal was a big piece of hardware

5. pi@rpi3: ~ (ssh)

```
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Jul 19 22:17:29 2016 from 192.168.2.1
pi@rpi3:~ $
pi@rpi3:~ $
pi@rpi3:~ $
pi@rpi3:~ $
pi@rpi3:~ $
pi@rpi3:~ $
pi@rpi3:~ $
pi@rpi3:~ $
pi@rpi3:~ $
pi@rpi3:~ $
pi@rpi3:~ $
```

Prompt    Cursor    (and here you can type things)

# Directories and files

Directories == known as “folders” on other operating systems

We call them **directories**, sometimes also called “path”

Just a way to organize your work

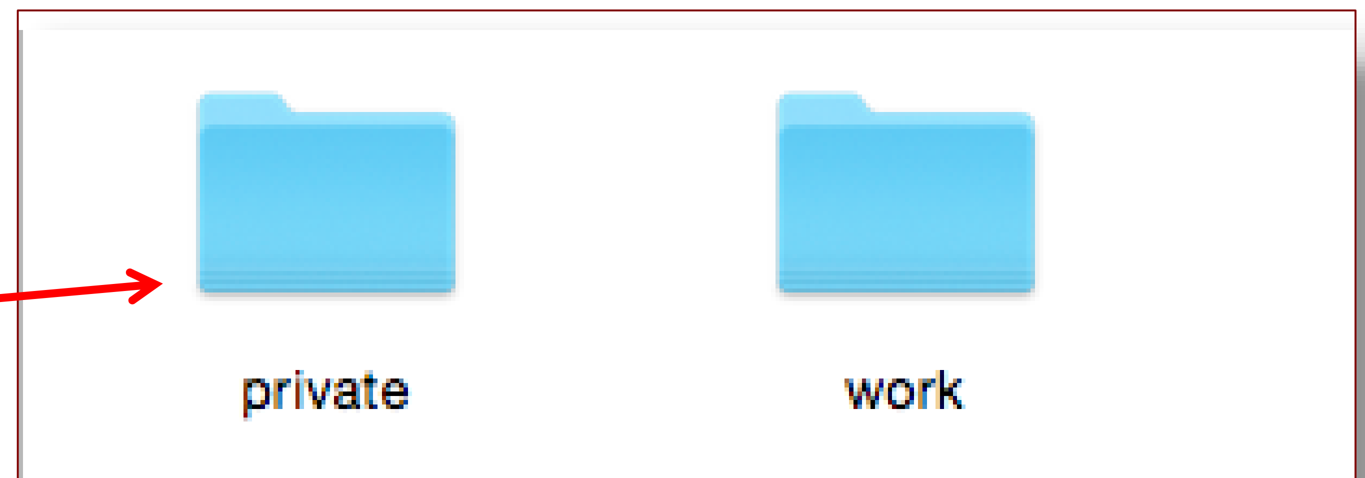
Like “private” and “work”

Use **cd** to navigate directories (“change directory”)

Use **pwd** to show where you are (“print work directory”)

Certain commands only make sense if we stick with “directory”

```
pi@rpi3:~ $ cd tree
pi@rpi3:~/tree $ pwd
/home/pi/tree
pi@rpi3:~/tree $ ls
private  work
pi@rpi3:~/tree $
```



# Directories and files

Use **pwd** to show where you are ("print work directory")

Most shell users picture the "pwd" as a physical place ("I *am* in the so-and-so directory")

The current directory acts like it is prepended to a file path

```
pi@rpi3:~/tree $ touch myfile
pi@rpi3:~/tree $ pwd
/home/pi/tree
pi@rpi3:~/tree $
```

I only typed "**myfile**"

The default directory acts as if it is prepended to the file name

**`myfile ->   /home/pi/tree + myfile  -> /home/pi/tree/myfile`**

It is an easy way to save a lot of typing

But you can still refer to the file by its full name **`/home/pi/tree/myfile`**

# Commands Utilities Programs

On a Unix system, they all mean pretty much the same

For **everything** you do, you execute a new program that does what you tell the shell to do

All the shell does is to call up those programs based on what command you type

We have seen the **`pwd`** command – print work directory

We can use the **`which`** command to find out what happens

```
pi@rpi3:~/tree $ pwd
/home/pi/tree
pi@rpi3:~/tree $ which pwd
/bin/pwd
pi@rpi3:~/tree $
```

```
pi@rpi3:~/tree $ which which
/usr/bin/which
pi@rpi3:~/tree $
```

So if you type “pwd”, the shell executes a program /bin/pwd that prints the current directory

# Navigating

The primary tool to “go to” a particular place in the directory structure is “cd”

“change directory”

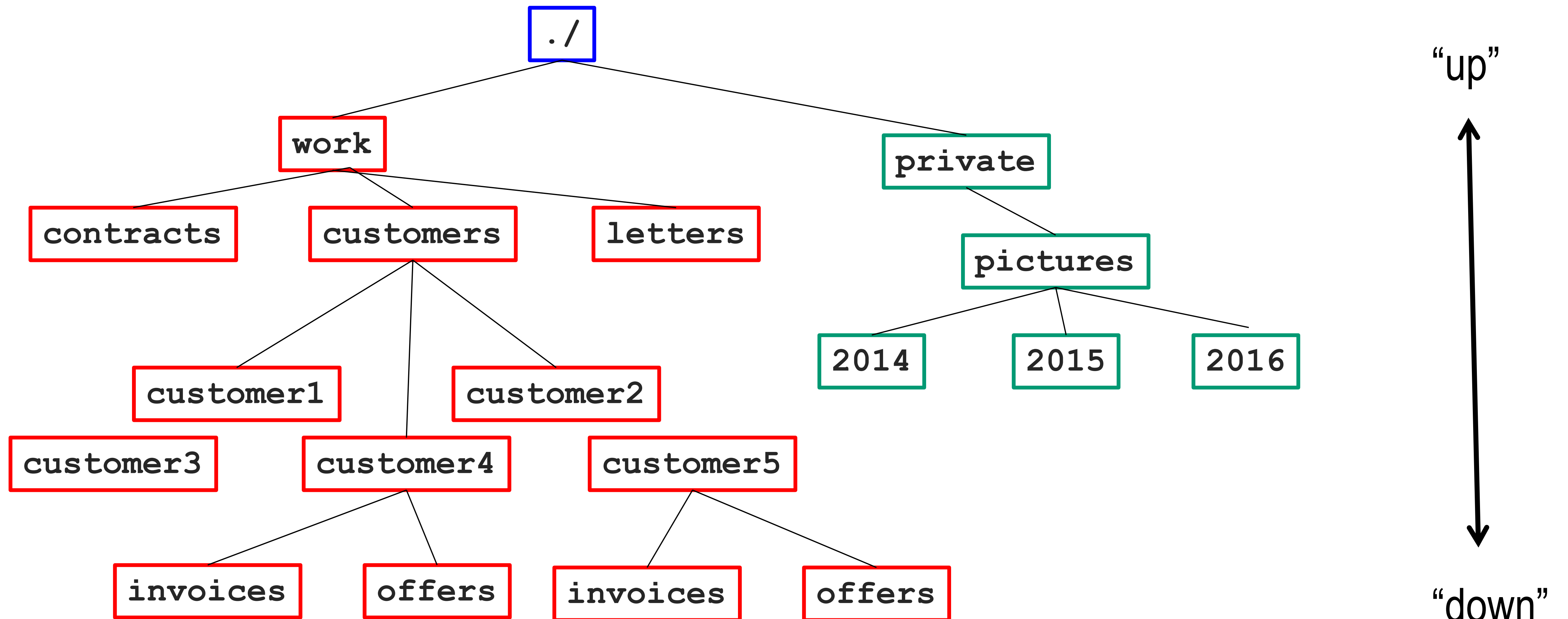
You have absolute moves and relative moves

Being able to navigate the directories with ease saves lot of time

After working on a Linux / Unix system for a while you develop some sort of a “map” for that

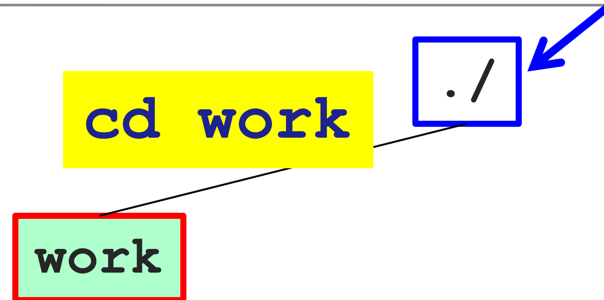
In the rare occasions I work on Windows machines, I still try to visualize such a map

# A directory tree



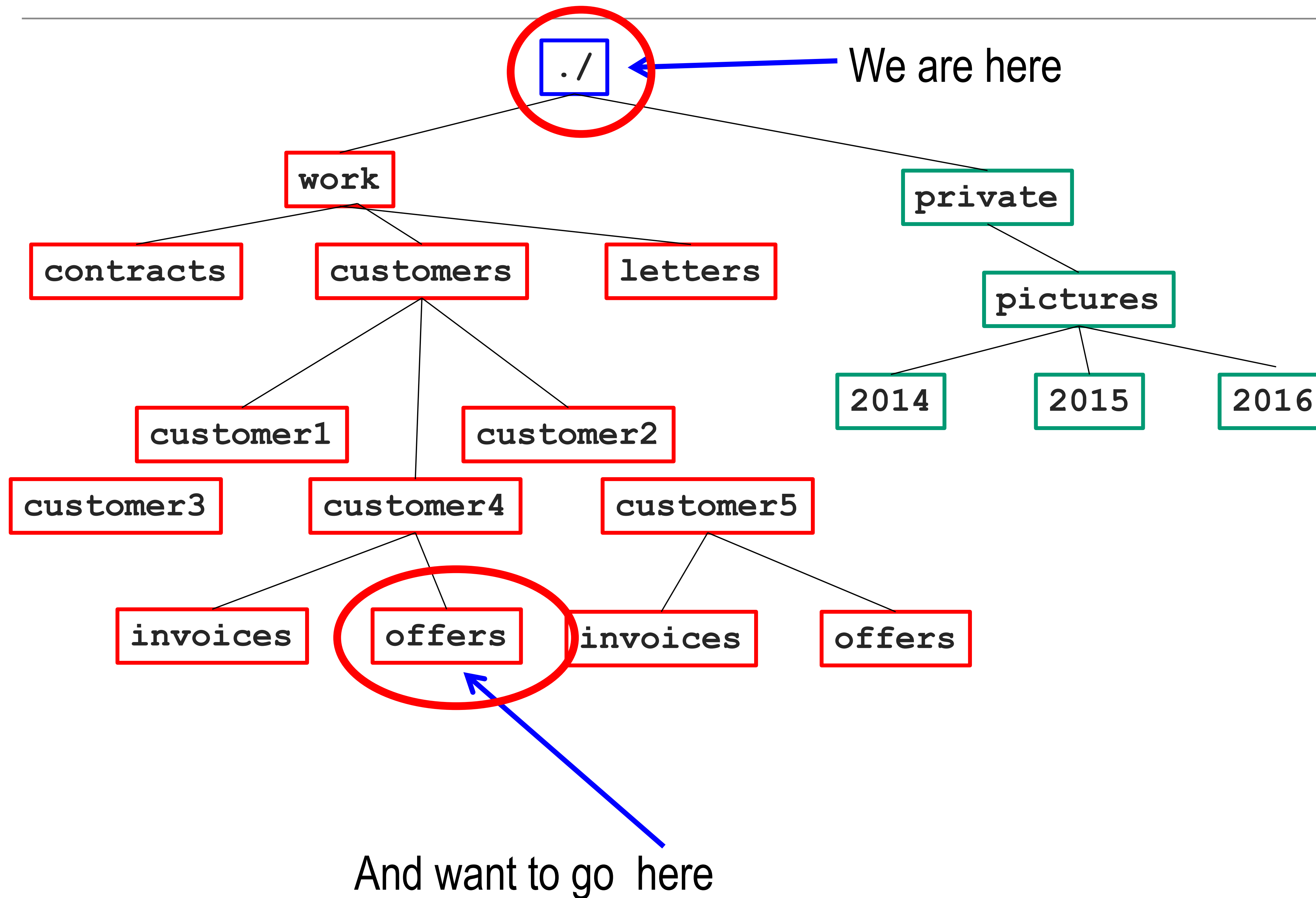
# Navigating

We start out here

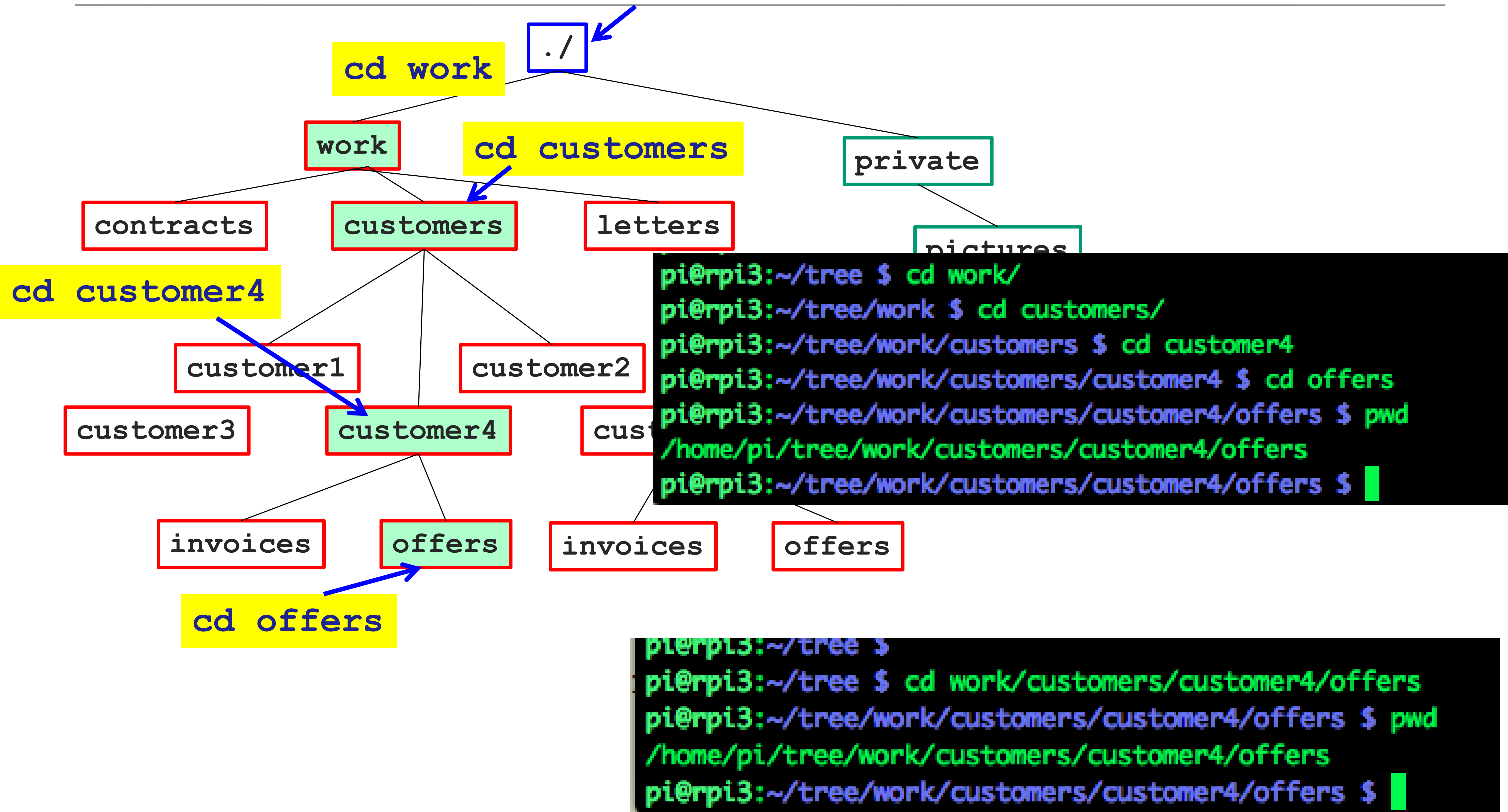



```
pi@rpi3:~/tree $ pwd
/home/pi/tree
pi@rpi3:~/tree $ cd work
pi@rpi3:~/tree/work $ pwd
/home/pi/tree/work
pi@rpi3:~/tree/work $
```

# Navigating a directory tree



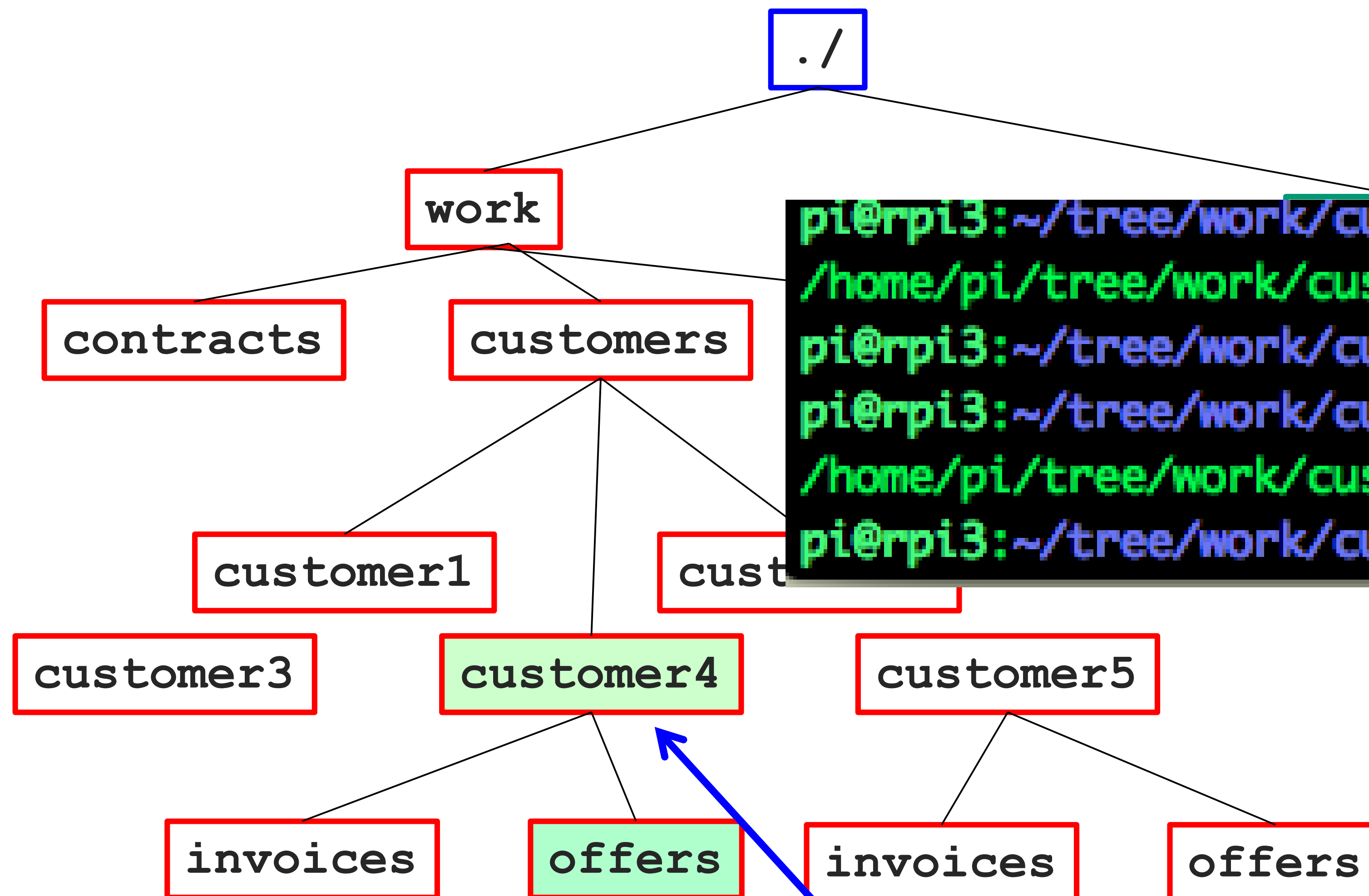
# Navigating



# Navigating: . and ..

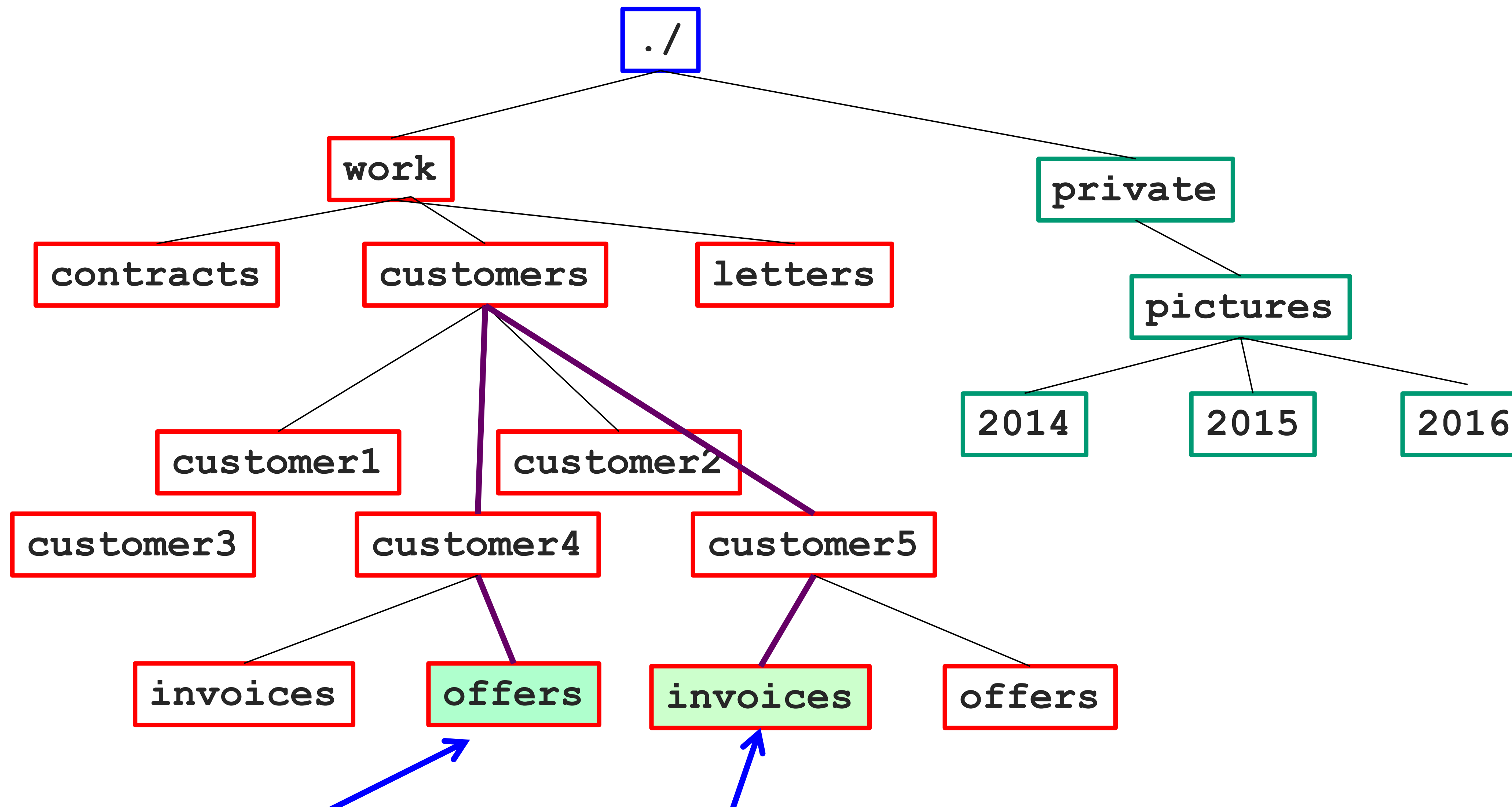
"." is a shorthand for "here"

".." is "one level up"




You are here… and want to go here: `cd ..`

# Navigating: combining “up” and “down”

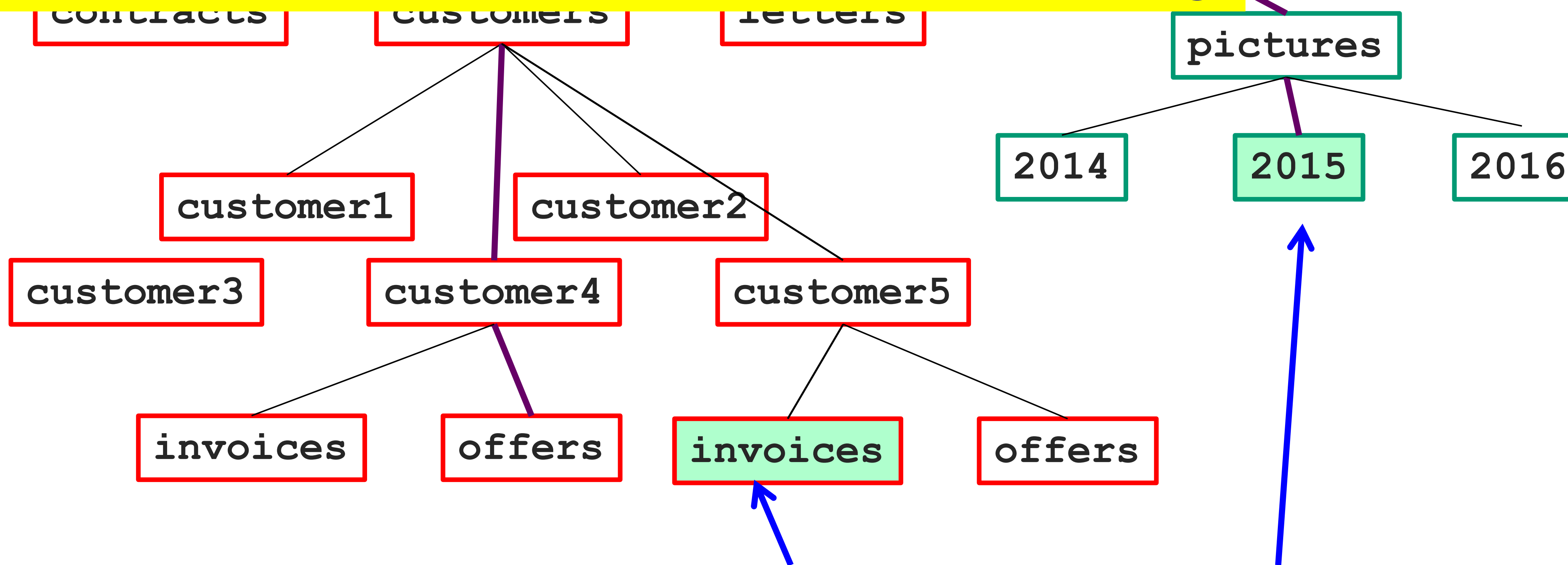



You are here… and want to go here:

```
pi@rpi3:~/tree/work/customers/customer4/offers $ pwd
/home/pi/tree/work/customers/customer4/offers
pi@rpi3:~/tree/work/customers/customer4/offers $ cd ../../customer5/invoices/
pi@rpi3:~/tree/work/customers/customer5/invoices $ pwd
/home/pi/tree/work/customers/customer5/invoices
pi@rpi3:~/tree/work/customers/customer5/invoices $
```

# Navigating: combining “up” and “down”

**You must think that this is really complicated, counting the number of “..” required. The shell makes this easy**




You are here… and want to go here:

```
pi@rpi3:~/tree/work/customers/customer5/invoices $ pwd
/home/pi/tree/work/customers/customer5/invoices
pi@rpi3:~/tree/work/customers/customer5/invoices $ cd ../../../../private/pictures/2015
pi@rpi3:~/tree/private/pictures/2015 $ pwd
/home/pi/tree/private/pictures/2015
pi@rpi3:~/tree/private/pictures/2015 $
```

# The tab key is your best friend

When you type something, the tab key will expand this as much as possible

It saves you tons of typing! And saves mistakes!

You will see seasoned shell users hit tab all the time

Let's say you have a file called

**Invoice_March27_2016_work_done_inFebruary_by_Martin_version7**

**Here I hit tab**

```
$ open Inv
$ open Invoice_March27_2016_work_done_inFebruary_by_Martin_version7
```

# ls –l

“ls” lists the files in a directory

ls –l  adds more information  “ls dash ell”

```
mlpmac:tree purschke$ ls
private work
mlpmac:tree purschke$ ls -l
total 0
drwxr-xr-x  3 purschke  staff  102 Jul 17 22:49 private
drwxr-xr-x  5 purschke  staff  170 Jul 17 23:24 work
```

# Command options

Many commands have a large number of options that modify the behavior

As we have seen with ls –l

Commands can have “short” (one-letter) options and “long options”

For example ls –group-directories-first

Long options should start with two dashes, but many utilities don’t adhere to that

# Scripts

We have so far executed each of our commands by typing it

Now I put those exact commands in a file, one line after the other. I call this file my_script.sh

The file can have any name, but the .sh extension is a good convention

```
pi@rpi3:~/tree $ pwd
/home/pi/tree
pi@rpi3:~/tree $ cd work
pi@rpi3:~/tree/work $ pwd
/home/pi/tree/work
pi@rpi3:~/tree/work $ cd ..
pi@rpi3:~/tree $ pwd
/home/pi/tree
pi@rpi3:~/tree $
```

Now I can execute those commands in one fell swoop:

```
pwd
cd work
pwd
cd ..
pwd
```

```
pi@rpi3:~/tree $ sh my_script.sh
/home/pi/tree
/home/pi/tree/work
/home/pi/tree
pi@rpi3:~/tree $
```

We add one more line

```
#! /bin/sh

pwd
cd work
pwd
cd ..
pwd
```

```
pi@rpi3:~/tree $ chmod +x my_script.sh
pi@rpi3:~/tree $ ./my_script.sh
/home/pi/tree
/home/pi/tree/work
/home/pi/tree
pi@rpi3:~/tree $
```

# Before we go on – “the other kind of text processing”

On the following slides I will show you some concepts and commands that give you a jump-start with the shell.

This may not be obvious, but pretty much every command does some kind of **text manipulation**

Think of the Unix shell and the utilities as the

**Most powerful text processor you can find**

And a lot of work you do is text manipulation whether you realize this or not – that includes numbers

So this is orders of magnitude more powerful than any monolithic “text processor”.

Keep that in mind as we go along…

# Some useful programs “wc” – “word count”

wc   counts lines, words, and characters in a file

```
$ wc script.sh
       5       8      41 script.sh
```

5 lines, 8 words, 41 characters

“Please explain what you liked about your experience (not more than 200 words)”

Useful to know home many words you actually used….

Just count the lines:

```
$ wc -l script.sh
       5 script.sh
```

# Shell variables

You can store values in “environmental variables”

```
$ VARIABLE=value
```

You can then retrieve the stored value by $VARIABLE

They can have any name and case, but usually one writes then in all-uppercase

```
$ printenv
TERM_PROGRAM=iTerm.app
TERM=xterm-256color
SHELL=/bin/bash
TMPDIR=/var/folders/7j/91pk1g_144b5y9vgy3gfy4gw0000gq/T/
Apple_PubSub_Socket_Render=/private/tmp/com.apple.launchd.s6dk
LBBPc2/Render
TERM_PROGRAM_VERSION=3.1.6
OLDPWD=/Users/purschke/presentations/RealTimeSchool/2018/photo
editing/copy1
TERM_SESSION_ID=w11t0p0:FFEA3644-74C7-4BEA-BB8E-C583B6227C06
USER=purschke
```

# Echo! Echo!

The echo command simply prints the arguments to “stdout”

```
$ echo Martin
Martin
```

Ok, that’s pretty lame… we knew the answer beforehand!

But not always! This is a convenient way to see the value of a variable.

```
$ VARIABLE=Martin
$ echo $VARIABLE
Martin
```

# Pipes

Whatever one command "prints to the screen" can be "piped" to another program with the | symbol

So, how many files/directories do I have here? As many as I have lines (ok, one less):

```
$ ls -l
total 26184
-rw-r--r--@  1 purschke  staff  12092605 Jul 17 21:11 RCDAQ.pptx
drwxr-xr-x  16 purschke  staff       544 Jul 17 21:57 pictures
drwxr-xr-x  21 purschke  staff       714 Jul 17 22:18 pictures2
-rwxr-xr-x   1 purschke  staff        41 Jul 17 20:57 script.sh
-rwxr-xr-x   1 purschke  staff       154 Jul 17 21:00 script2.sh
-rw-r--r--   1 purschke  staff        41 Jul 17 20:58 script2.sh~
-rw-r--r--@  1 purschke  staff   1297508 Jul 17 23:57 shell.pptx
drwxr-xr-x   4 purschke  staff       136 Jul 18 00:13 tree

$ ls -l | wc -l
       9
```

This is a fantastically powerful concept

# Wildcards

Wildcards allow you to select files with common parts in their name (and may other things)

Here: Show me the PowerPoint files in this directory

```
$ ls -l *.pptx
total 1231
-rw-r--r--@  1 purschke  staff  12092605 Jul 17 21:11 RCDAQ.pptx
-rw-r--r--@  1 purschke  staff   1297508 Jul 17 23:57 shell.pptx
```

It is important to understand that it is not “ls” that expands the wildcards but the shell – it gives the two names to ls, and ls does its thing

I can have more than one wildcard:

```
$ ls -l *e*.pptx
total 988
-rw-r--r--@  1 purschke  staff   1297508 Jul 17 23:57 shell.pptx
```

# “Pipe into”

Different people call it by different names – I like “**pipe into**” - the “bar” symbol **|**

```
$ ls -l | wc -l
       9
```

So I would tell you to type

**“ eles dash el pipe into wc dash el”**

# Standard in and standard out

You have already seen this with the "pipe into" –

Many programs have a concept of "stdin" and "stdout"

They take some input from "stdin" (usually from your terminal)

and write something out (usually to your terminal)

Here: the "cat" program (con**cat**enate) – it *can* take its stdin (has other uses, hence my careful wording ☺ ) and puts stuff out to stdout. Here:

I type → 1234
It prints out → 1234

```
$ cat
1234
1234
```

Ok, that's a silly example, but you get the idea

You can think of such a program as a **filter** (ok, "cat" doesn't actually do anything to the input, but just wait)

If a program doesn't "do" stdout naturally, you can usually force it – we'll see an example later

# Keep on piping …

"sed" is the **"streamline editor"**

It is an enormously powerful editor that takes the input, does something to it, outputs – that's an actual filter! (you would not use it for editing some big thing….)

Remember echo? It takes the argument and prints it to stdout:

```
$ echo Martin
Martin
```

Here we go – we tell sed to substitute "a" for an "u":

```
$ echo Martin | sed 's/a/u/'
Murtin
```

But sed again prints its output to stdout, so we can go on:

```
$ echo Martin | sed 's/a/u/' | sed 's/i/e/'
Murten
```

And we can go on like this – "tr" translates one group or characters to another one (here: make everything uppercase)

```
$ echo Martin | sed 's/a/u/' | sed 's/i/e/' | tr a-z A-Z
MURTEN
```

# A super-useful program: “awk” – but what does that even mean?

It is a utility named after the initials of the authors - Alfred **A**ho, Peter **W**einberger, and Brian **K**ernighan

It is a swiss army knife for text parsing.

My full desktop at BNL is called “mlp.rhic.bnl.gov”. What if I want just the “mlp” name?

```
$ hostname
mlp.rhic.bnl.gov
$ hostname | awk -F. '{print $1}'
mlp
```

Sum something per-line up:

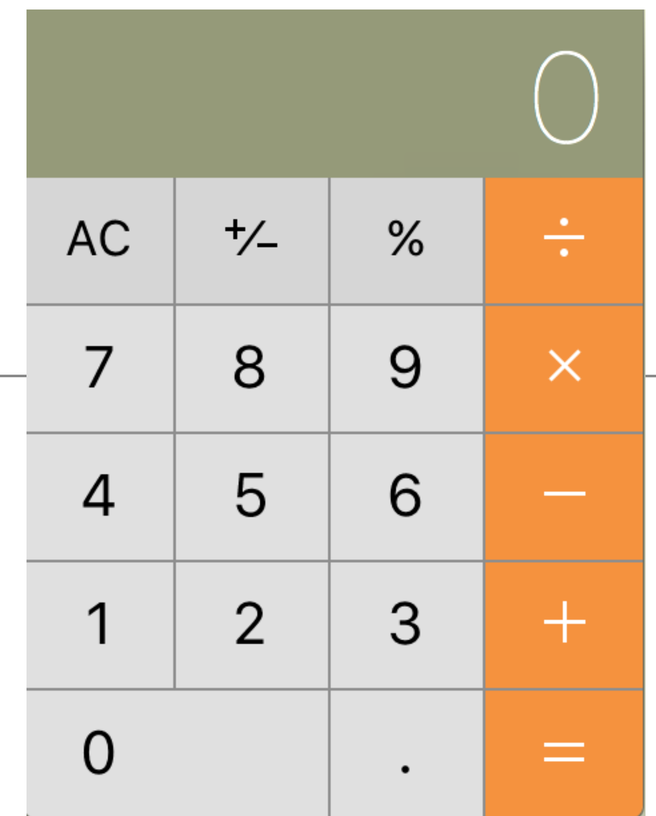
```
$ ls -l | awk '{sum+=$5} END {print sum}'
13391743
```

# sort

We list our files sorted by file size ( numerically sort by the 5th column):

```
$ ls -l | sort -n -k 5
total 26184
-rw-r--r--   1 purschke  staff        41 Jul 17 20:58 script2.sh~
-rwxr-xr-x   1 purschke  staff        41 Jul 17 20:57 script.sh
drwxr-xr-x   4 purschke  staff       136 Jul 18 00:13 tree
-rwxr-xr-x   1 purschke  staff       154 Jul 17 21:00 script2.sh
drwxr-xr-x  16 purschke  staff       544 Jul 17 21:57 pictures
drwxr-xr-x  21 purschke  staff       714 Jul 17 22:18 pictures2
-rw-r--r--@  1 purschke  staff   1297508 Jul 17 23:57 shell.pptx
-rw-r--r--@  1 purschke  staff  12092605 Jul 17 21:11 RCDAQ.pptx
```

# bc




Each computer and smartphone has a virtual pocket calculator these days

There is no slower way to calculate something… IMHO

Throw in “bc” - “binary calculator”

Arbitrary precision, text input, easy, takes stdin/out,….

```
$ bc -lq
1.234 * 2812894
3471111.196
3.78532 * 1276 / 9 * 15
8050.11386666666666666655

$ echo "1.234 * 2812894" | bc -lq
3471111.196
```

# Where did I put *that*????   -- find is your friend

Let's say I remember that I had at some point written a program in C++ and I vaguely remember its name (but not fully)

It was reading out a device called an Xbee. I would lIke to copy–paste some code from it

But I have no idea where I put that. I have more then 1800 directories on my Mac…

Am I looking through 1800+ directories?

It is a good guess that the program had the word "xbee" in its name… so I go

```
$  find . -name "*xbee*.cc"
./muell/heatserver/software/read_xbee.cc
./muell/software/read_xbee.cc
./softwarerepo/software/read_xbee.cc
$
```

See? I found not only one but 3… They are likely copies of the same file that I worked on

# find all files that were modified in the last day

Let's see how busy I have been (ok, not all file modification represent work by me, there are mail files, web browser cache, etc etc)

```
$  find . -type f -mtime -1
. . . Many lines deleted . . .
./presentations/RealTimeSchool/2018/photoediting/IMG_20180707_080137-6.jpg
./presentations/RealTimeSchool/2018/photoediting/IMG_20180707_080137-7.jpg
./presentations/RealTimeSchool/2018/photoediting/IMG_20180707_080137-8.jpg
./presentations/RealTimeSchool/2018/photoediting/IMG_20180707_080137-9.jpg
./presentations/RealTimeSchool/2018/raspberryPi_instructions.pptx
./presentations/RealTimeSchool/2018/RCDAQ.pptx
./presentations/RealTimeSchool/2018/scaledowns.pptx
$
```

You can see that I have been working on my presentations here. But how many files are that, total?

```
$ find . -type f -mtime -1 | wc -l
    1218
```

# find all files that contain a certain string

Let's say that I worry about files where I have inadvertently stored a password. Just making sure that I'm in the clear… My password is "secret1234" ( no it is not ☺ )

We look for all actual files (e.g. not directories)

-exec executes the command for each entry

The {} is replaced with each file

```
$  find . -type f -exec grep -l secret1234 {} \;
./muell/mypassword.txt
$
```

The ; (semicolon) terminates the command

So I go through ALL files and look for the presence of that string. grep –l … prints the name of the string.

That was easy!

# bc - arbitrary precision. Printing 1000 digits of pi

Arctan(1) = pi/4

```
$ bc -lq
4*a(1)
3.14159265358979323844

scale=1000
4*a(1)
3.141592653589793238462643383279502884197169399375105820974944592307\
8164062862089986280348253421170679821480865132823066470938446095505\
82231725359408128481117450284102701938521105559644622948954930381964\
42881097566593344612847564823378678316527120190914564856692346034861\
04543266482133936072602491412737245870066063155881748815209209628292\
54091715364367892590360011330530548820466521384146951941511609433057\
27036575959195309218611738193261179310511854807446237996274956735188\
57527248912279381830119491298336733624406566430860213949463952247371\
90702179860943702770539217176293176752384674818467669405132000568127\
14526356082778577134275778960917363717872146844090122495343014654958\
53710507922796892589235420199561121290219608640344181598136297747713\
09960518707211349999998372978049951059731732816096318595024459455346\
90830264252230825334468503526193118817101000313783875288658753320838\
14206171776691473035982534904287554687311595628638823537875937519577\
818577805321712268066130019278766111959092164201988
```

# Shamelessly Showing off!

SIMULATING 5 TRILLION EVENTS ON THE OPEN SCIENCE GRID

Martin L. Purschke, BNL

This came up in the context of a super-sized physics simulation project

In the end, I ran a half-million jobs on the Computing Grid over the course of ~7 weeks

Every Friday, the OSG admins wanted a status report – "how much CPU time did you use"?

For each job on the grid you get a log file with the relevant CPU usage numbers – but how to extract them? Pocket calculator? Nah. Excel? Nah. But what is the most powerful text processing tool?

So in the end I had 500,000 log files that look like this:

```
… lines deleted
005 (4972.000.000) 08/05 14:07:27 Job terminated.
   (1) Normal termination (return value 1)
      Usr 0 18:27:09, Sys 0 00:09:11  -  Run Remote Usage
      Usr 0 00:04:27, Sys 0 00:01:18  -  Run Local Usage
      Usr 0 18:27:09, Sys 0 00:09:11  -  Total Remote Usage
      Usr 0 00:04:27, Sys 0 00:01:18  -  Total Local Usage
   0  -  Run Bytes Sent By Job
… lines deleted
```

So this job used 18h 27m 09s

But we have ~500,000 such files!

Can we do the calculation on one line?

# Text processing…

First, find all log files ( I copied only 2 over here – remember we had 500,000)

```
# find . -name "*.log"
./condor_171878.log
./condor_171879.log
```

Find the one line per file that we need by executing “grep” on each file:

```
# find . -name "*.log" -exec grep 'Total Remote Usage' {} \;
  Usr 0 22:11:04, Sys 0 00:11:11  -  Total Remote Usage
  Usr 0 18:27:09, Sys 0 00:09:11  -  Total Remote Usage
```

We remove the commas (that would get in the way later) with sed:

```
# find . -name "*.log" -exec grep 'Total Remote Usage' {} \;
   | sed 's/,//'
  Usr 0 22:11:04 Sys 0 00:11:11  -  Total Remote Usage
  Usr 0 18:27:09 Sys 0 00:09:11  -  Total Remote Usage
```

# Text processing…

Now we extract the 3rd parameter which is the CPU time:

```
find . -name "*.log" -exec grep 'Total Remote Usage' {} \;
  | sed 's/,//’
  | awk '{print $3}'
22:11:04
18:27:09
```

Now we are using awk again to get hours, minutes, seconds (we just print them here):

```
find . -name "*.log" -exec grep 'Total Remote Usage' {} \;
    | sed 's/,//’
    | awk '{print $3}’
    | find . -name "*.log" -exec grep 'Total Remote Usage' {} \;
    | sed 's/,//' | awk '{print $3}’
    | awk -F: '{print "hours:", $1, " Minutes:", $2, " seconds: ", $3}'
hours: 22  Minutes: 11  seconds:  04
hours: 18  Minutes: 27  seconds:  09
```

# And finally we add it all up …

```
find . -name "*.log" -exec grep 'Total Remote Usage' {} \;
    | sed 's/,//'
    | awk '{print $3}'
    | find . -name "*.log" -exec grep 'Total Remote Usage' {} \;
    | sed 's/,//' | awk '{print $3}'
    | awk -F: '{X += ($1 *3600 + $2*60 + $3)/3600} END {print X}'

40.6369
```

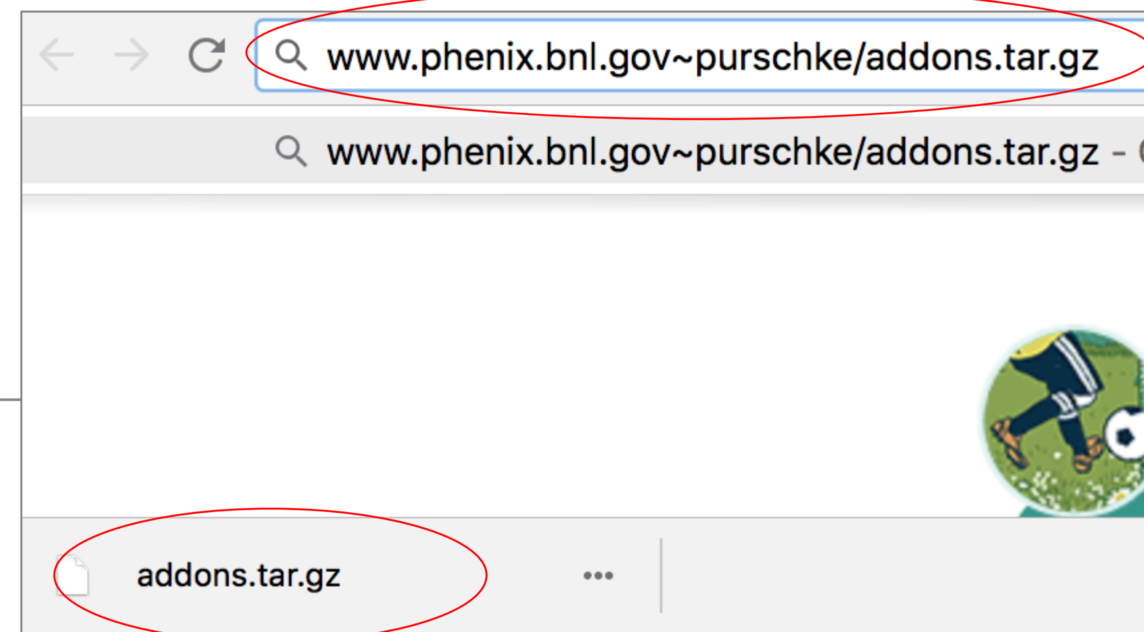
```
$ find log/ -name ``*.log'' -exec grep 'Total Remote Usage' {} \; | \
   sed -e 's/,//g' | awk '{print $3}' | \
   awk -F: '{X += ($1 *3600 + $2*60 + $3)/3600} END {print X}'

1.34523e06

$ bc -l
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
1345230 / 24
56051.25000000000000000000
1345230 / 24/365
153.56506849315068493150
```

1345230 hours
56051 days
153 years

# One last thing:




We will use the wget utility to download and unpack a file from my web server

wget is like a command-line web browser: It downloads the content of a URL to your computer

But we don't want to fire up the web browser

and we don't want to store the file first but use pipes to extract the info right away

```
wget http://www.phenix.bnl.gov/~purschke/addons.tar.gz
```

would download **and store the file** on your disk (but we don't want that!)

So we force wget to operate silently, AND force it to print out the contents to stdout (don't do that, you'll see a lot of gibberish – we don't want that typed to the terminal!)

```
wget -q -O - http://www.phenix.bnl.gov/~purschke/addons.tar.gz
```

And we process the data from a pipe (this is one line):

```
wget -q -O - http://www.phenix.bnl.gov/~purschke/addons.tar.gz
   | tar xvz
```

# The End