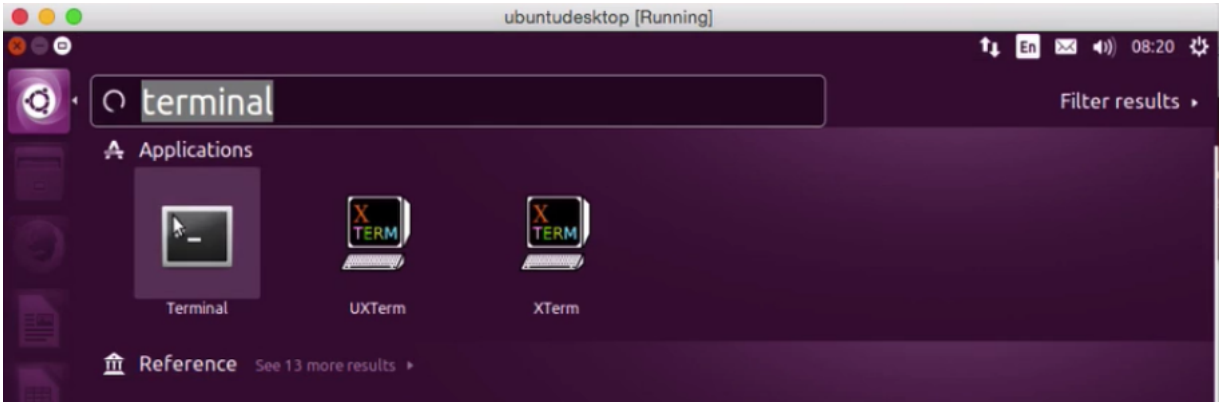
06

Globbing, quoting, history e a variável PATH

Navegando na linha de comando e o set

Vamos testar mais algumas coisas na linha de comando?

Só para garantir que estaremos iniciando do zero, vamos abrir um novo terminal:



Lembra-se que temos alguns arquivos já criados? Para verificar isso podemos dar um `ls` e veremos em nossa listagem, por exemplo, o arquivo `mostra_idade` :

```
> ls
Desktop         examples.desktop        mostra_idade        Public
Documents       loja                    Music               Templates
Downloads       mostra_idade            Pictures            Videos
```

Se quisermos editar o arquivo `mostra_idade` podemos modificá-lo usando diversos programas, inclusive, através de um editor gráfico, como o *gedit*.

Podemos digitar no terminal `gedit mostra_idade` . Mas, e se nós tivéssemos escrito apenas `gedit mostra` ?

```
> ls
Desktop         examples.desktop        mostra_idade        Public
Documents       loja                    Music               Templates
Downloads       mostra_idade            Pictures            Videos
> gedit mostra
```

Abriria um arquivo do editor, mas não é esse arquivo que estamos procurando, então, fecharemos esse editor e não salvaremos nada nele e podemos escrever agora, corretamente o `gedit mostra_idade` .

Ao em vez de escrevermos todo o `gedit mostra_idade` novamente, podemos utilizar um atalho. Atalhos são padrões no *Linux* e eles ocorrem quando estamos trabalhando com um terminal.

Se quero acessar o comando anterior, o último comando que digitamos, basta usar as setas do teclado, no caso, para cima, e teremos o `gedit mostra` . Se apertarmos para cima novamente, teremos o penúltimo comando, o `ls` . Se apertarmos para baixo, voltamos para o `gedit mostra` . E se apertarmos para baixo de novo? Não temos nenhum comando.

Imagine que agora digitamos um `cd loja` e usaremos também um `git status`, teremos:

```
> cd loja
~/loja$
```

E se por exemplo escrevermos também um repositório do `git`, um `git status`. Teremos ao todo:

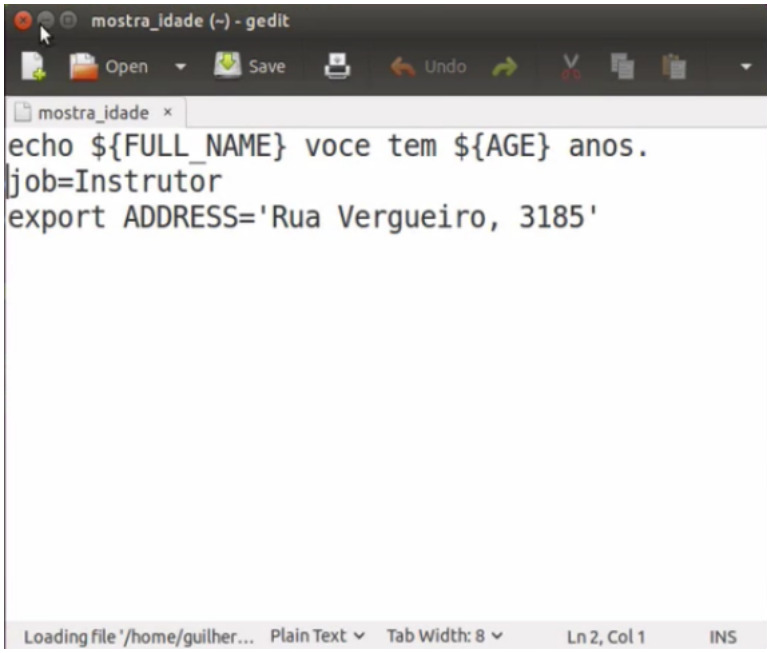
```
> ls
Desktop         examples.desktop        mostra_idade        Public
Documents       loja                    Music               Templates
Downloads       mostra_idade            Pictures            Videos
> gedit mostra

> cd loja
~/loja$ git status
On banch master

Initial commit

Chanes to be commited:
    (use "git rm --cached <file>..." to unstage)

    new file:   bemvindo.html
```

Repare que utilizamos diversos comandos: `ls`, `gedit mostra`, `cd loja`, `git status`. Então, podemos navegar entre esses comandos com as setas para cima e para o baixo.

Com isso, navegaremos pelo histórico de comandos utilizando esses atalhos, o para cima e o para baixo.

Vamos voltar para o diretório anterior antes de seguir. Utilizaremos para retornar o `cd ..`. E, como fazemos para voltar a executar aquele `gedit mostra` com mais um pouco, com o complemento que faltou no início, o `_idade`? Basta utilizar o para cima até encontrar o `gedit mostra` e resta apenas complementar digitando `_idade`.

Vai abrir o editor:



Pronto, o editor está aberto e podemos fechá-lo.

Vamos analisar, ainda, outros atalhos.

Assim como usamos o para cima e para baixo também podemos utilizar as setas de direita e esquerda. Podemos também ir para o começo da linha, para isso utilizaremos a tecla "home" e para nos direcionarmos ao final da linha usamos a tecla "end". Se o teclado for de *Mac* podem ser utilizados também o "Ctrl+A" para ir ao começo da linha e "Ctrl+E" para ir ao final da linha.

Podemos apagar um pedaço do comando com "backspace", por exemplo, ficaremos com o `gedit mostra` de novo.

Repare que se formos para baixo e não existir nenhum comando mais abaixo do `gedit mostra` e voltarmos usando para cima ele vai lembrar que tínhamos apagado o `_idade` e vai mostrar apenas o `gedit mostra` .

Claro, se executarmos abaixo do `gedit mostra` outro comando, como o `ls` por exemplo e executar dois para cima ele ainda vai lembrar que apagamos aquele pedaço.

```
> gedit mostra
> ls
Desktop         examples.desktop        mostra_idade        Public
Documents       loja                    Music               Templates
Downloads       mostra_idade            Pictures            Videos
> gedit mostra
```

Isso significa que este é o nosso histórico, podemos navegar nele, para baixo e para cima, direita e esquerda, "backspace", "Home" ou "Ctrl+A" e "End" ou "Ctrl+E". E, ainda, podemos apertar o "Delete". Se o teclado possui o "Delete" ele apaga o carácter atual.

Então, se você tiver um teclado com "Delete" ele apaga o carácter que está na sua posição, como era de se esperar.

Concluímos, que o teclado funciona como é de se esperar dele, por padrão, ou seja, como ele deveria funcionar.

E, agora, vamos analisar alguns outros detalhes que são comuns do dia a dia.

Imagina que você esteja escrevendo um arquivo e queremos escrever `gedit` , mas imagine que na verdade digitamos duas letras trocadas, o "t" por "i", teremos `gedti` . Para destrocar as últimas duas letras podemos utilizar o atalho "Ctrl+T".

E se queremos limpar a tela?

Se ela está cheia de informações podemos usar o `clear` , que é o que estamos usando desde o início, mas podemos utilizar também um atalho para fazer isso, o "Ctrl+L".

Lembre-se que já vimos outros "Ctrl". Quando digitávamos um comando e na verdade não queríamos executar ele, utilizávamos o "Ctrl+C" para quebrar o que estavámos fazendo. Existe, ainda, outra função no atalho "Ctrl+C", por exemplo, ao digitar `perl` e ele começar a ser executado podemos parar sua execução através do "Ctrl+C". Se usarmos o "Ctrl+C" teremos o seguinte:

```
> gedit^C
> perl
^C
```

Para outros programas, por exemplo, o `python`, se usarmos o "Ctrl+C", veremos que ele continuará executando. No caso do `phyton`, não temos mais informação nenhuma para mandar a partir do teclado, isto é, ele acabou, então, usaremos o "Ctrl+D". Teremos o seguinte:

```
> python

>>>
KeyboardInterrupt
>>>
```

O "Crl+D" diz que não temos mais informações para enviar. Repare que são coisas distintas. O "Ctrl+C" manda um sinal de parada, enquanto o "Ctrl+D" fala para o programa que não tem mais informações para ele.

Repare que todos os atalhos que vimos são atalhos padrão do *Linux*, mas podemos mudar as maneiras de utilizar esses atalhos, entre outras características. Para isso, utilizamos um programa chamado `set`, que é configurado para isso.

Por padrão a entrada de dados para navegar no comando é feita utilizando o `set -o emacs` que são os atalhos do `emacs` por padrão.

```
set -o emacs
```

Se executarmos esse comando continuaremos podendo utilizar os mesmo atalhos, para cima, para baixo, esquerda e direita e etc. Se alguém tiver a preferência pelo modo *vi* como editor pode alterar também para um modo do `vi`. Poderemos digitar, então, `set -o vi` e executar isso.

```
set -o emacs
set -o vi
```

O que o `set -o vi` faz?

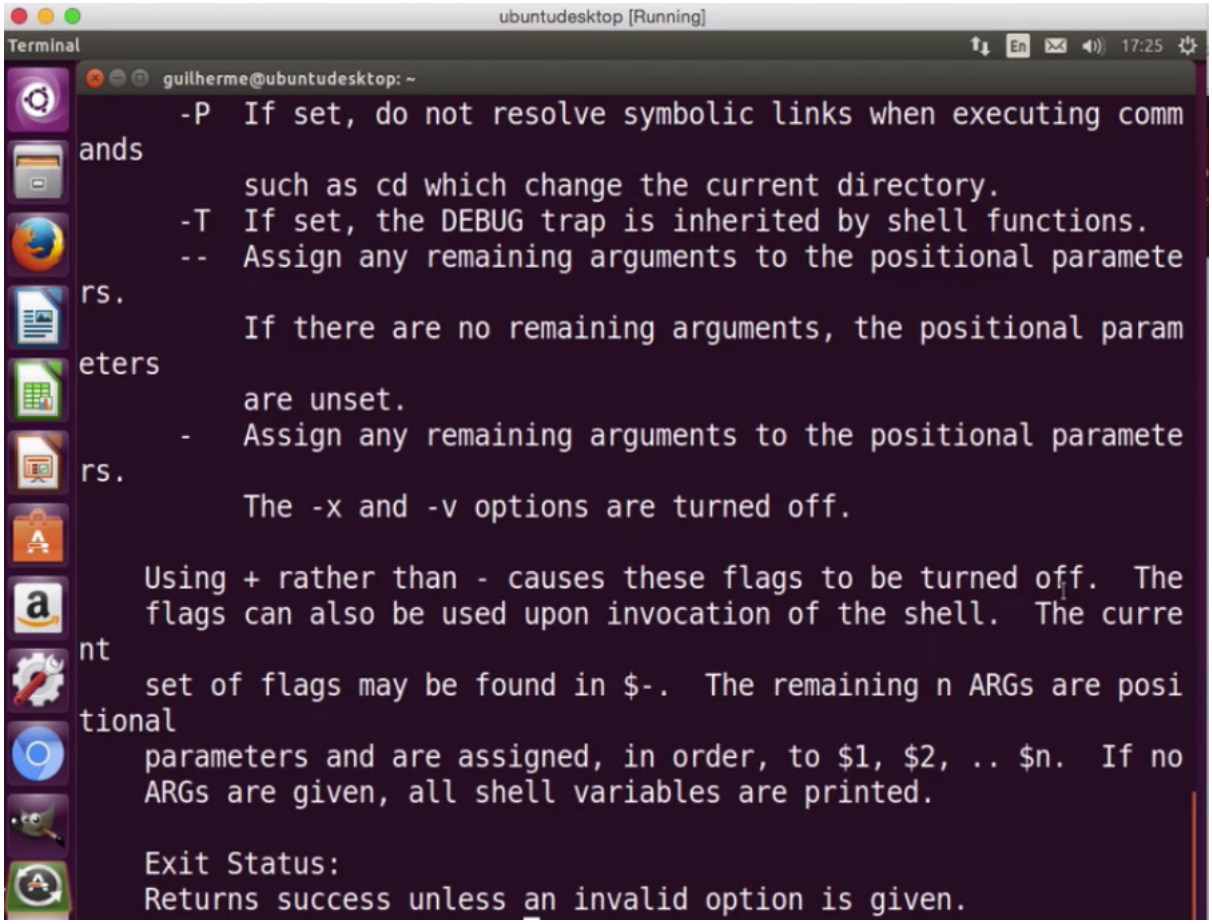
Ele altera os atalhos. Então, se nós voltarmos para os comandos que tínhamos executado antes, por exemplo, `gedit mostra_idade` e quisermos ir para o começo da linha usaremos outros atalhos.

No *vi* como fazemos para ir ao começo da linha?

Estando no modo de comando do *vi*, usaremos o "Esc" e digitaremos "0". Como estamos no modo de comando do *vi*, pois ainda estamos no "Esc" podemos ir para a direita ou esquerda utilizando as setas, e se quisermos inserir algum conteúdo no comando basta digitar "i" de inserir e adicionamos o que quisermos. Quando quisermos parar de inserir basta apertar "Esc", mas continuaremos no modo de comando, então, apertando novamente o "Esc" finalizamos o modo de comando. Por fim, podemos apertar o "Enter" ou fazer qualquer outra coisa.

Nós, particularmente, não utilizamos o modo do *vi*, portanto, vamos voltar atrás, para isso, digitaremos `set -o emacs`.

Vamos dar uma olhada no `help` do `set`, digitando, `help set`. Teremos a seguinte tela:



Aqui temos diversas opções de configurações do nosso terminal quando trabalhamos com esses comandos no terminal.

A prova, não irá cobrar, necessariamente, este conteúdo de nós. Mas, é importante conhecer o básico de um terminal de *shell*, de como interagimos com ele, por isso, é importante conhecermos esse atalhos. As setas para cima e para baixo para navegar no histórico, esquerda e direita, "Ctrl+A" para ir ao início da linha e "Ctrl+E" para ir ao final, "backscape" e "delete", "Ctrl T" para trocar letras e "Ctrl+L" para limpar a tela, "Ctrl+C" para enviar um sinal de interrupção para o programa e "Ctrl+D" para falar que acabou a entrada do usuário. Vimos, ainda, o `set -o vi` e o `set -o emacs`.

Veremos, na sequência, um pouco acerca do histórico.

History HISTFILE e o Ctrl r

Vamos olhar, agora, alguns comandos ligados ao que estávamos fazendo no terminal. É bem comum executarmos uma série de comandos.

Por exemplo, `ls` para ver o diretório, `cd loja` para entrar no diretório loja, `git status` para observar o status dele, `cd ..` para sair do diretório e podemos executar, novamente e damos, ainda, mais um `ls`.

```
> ls
Desktop         examples.desktop        mostra_idade~       Public
Documents       loja                    Music               Templates
Dowloads        mostra_idade            Pictures            Videos
> cd loja
~/loja$ ls
~/loja$ git status
On branch master

Initial commit
```

```
Changes to be commited
    (use "git rm --cached <file>..." to unstage)

        new file: bemvindo.html

~/loja$ cd ..
> ls
Desktop         examples.desktop        mostra_idade~       Public
Documents       loja                    Music               Templates
Dowloads        mostra_idade            Pictures            Videos
```

Vamos dar um `clear` para limpar nossa tela.

E se após tudo isso que fizemos, todos esses comandos que utilizamos, quiséssemos lembrar de algum comando específico? Perceba que nesse contexto de diversas alterações, o comando que estamos querendo lembrar já foi executado a algum tempo. Para buscar o comando usamos a seta para cima.

Usamos o para cima até encontrar o `gedit mostra_idade` . Esse comando está no armazenamento do histórico dos comandos que executamos.

Porém, existe alguma outra maneira de visualizar todos os comandos que já executamos, ou, pelo menos os últimos comandos que executamos? É possível ter acesso a isso de um jeito diferente?

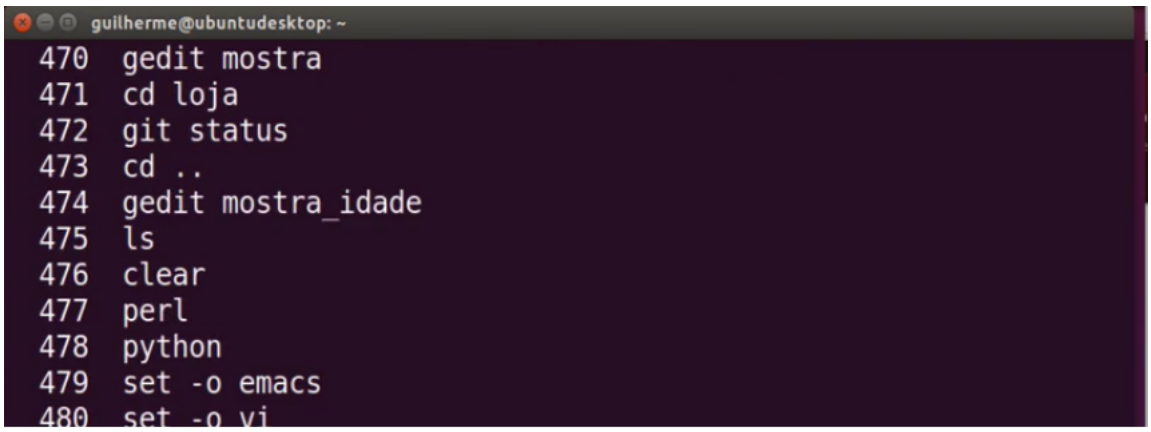
Temos no nosso terminal o `gedit mostra_idade` e damos um `Ctrl+C` para que o comando não seja executado.

```
gedit mostra_idade^C
```

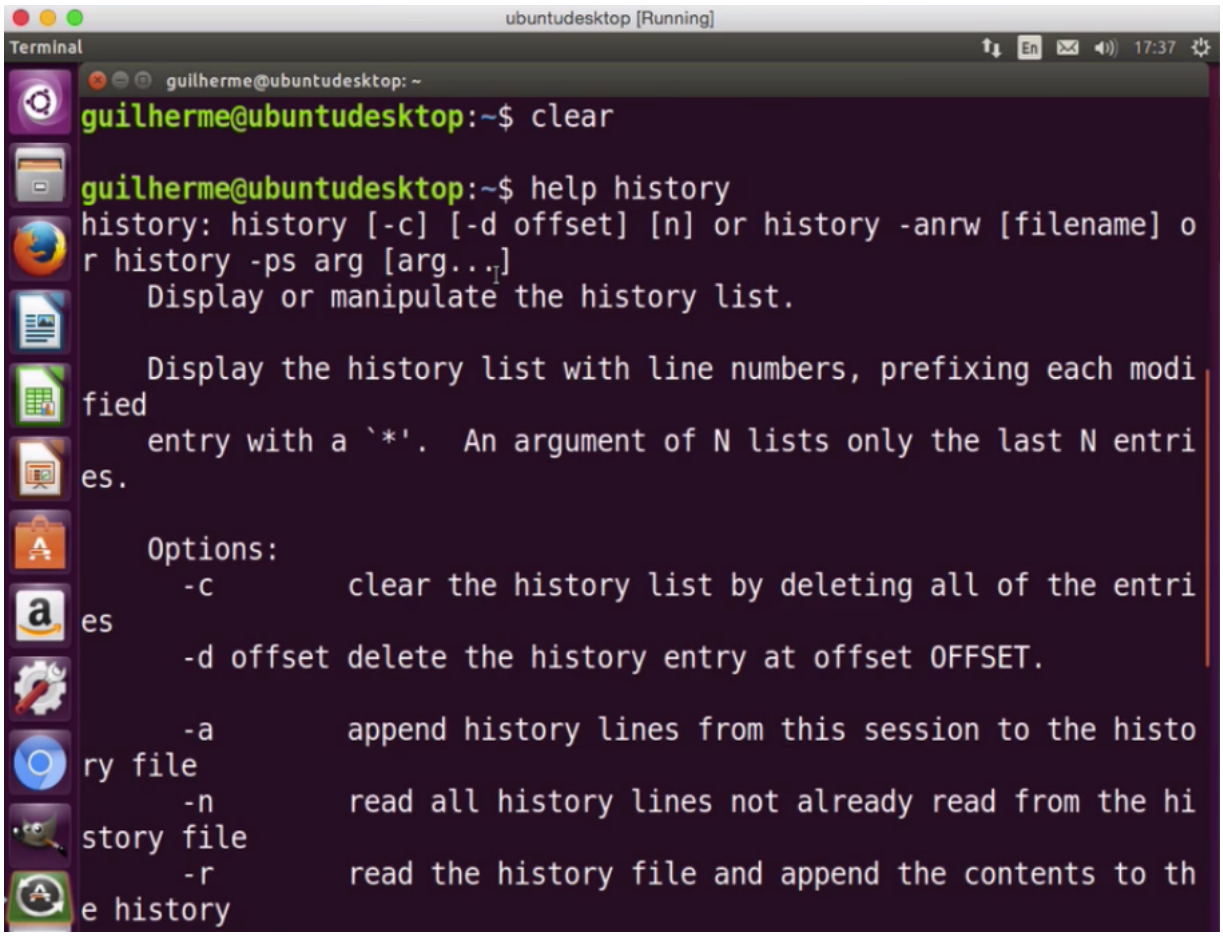
Na próxima linha podemos digitar `history` , que mostra o histórico de comando.

```
> gedit mostra_idade^C
> history
```

No nosso caso, temos uma lista de cerca de 495 comandos que já executamos:



Podemos digitar `help history` para compreender quais são as opções de uso do histórico:



O `-c`, por exemplo, é utilizado para limpar o histórico, assim, quando quisermos podemos realizar uma limpeza nele. O `-c` serve para realizar essa limpeza, entretanto, não é o que gostaríamos de utilizar nesse momento.

Se estamos no `history` e gostaríamos de visualizar os cinco últimos comandos, então, poderíamos digitar `history 5`. Teremos:

```
> history 5
    497 help history
    498 clear
    499 history
    500 clear
    501 history 5
```

Podemos observar que são mostrados os cinco últimos comandos, inclusive, o comando `history 5`, que é o último que acabamos de chamar.

Então, o `history` que é acompanhado de um número, temos que esse número diz respeito a quantidade de comandos que foram utilizados por último. Por exemplo, em `history 5`, o "5" se refere aos cinco últimos comandos utilizados e que desejamos visualizar.

E, retomando, o *history -c* limpa os comandos.

Mas, onde esses comandos são armazenados? É preciso que esse histórico seja armazenado em algum local, então, para descobrir isso podemos digitar `echo $HISTFILE`. Teremos:

```
echo $HISTFILE
/home/guilhere/.bash_history.
```

Todos estes comandos estão sendo salvos em `HISTFILE`. Qual é o valor dessa variável? É `/home/guilhere/.bash_history`. E, no *bash*, `/home/guilhere/.bash_history`, é o padrão onde se armazenam esses arquivos. Vamos parar a execução do `HISTFILE` usando o "Ctrl c".

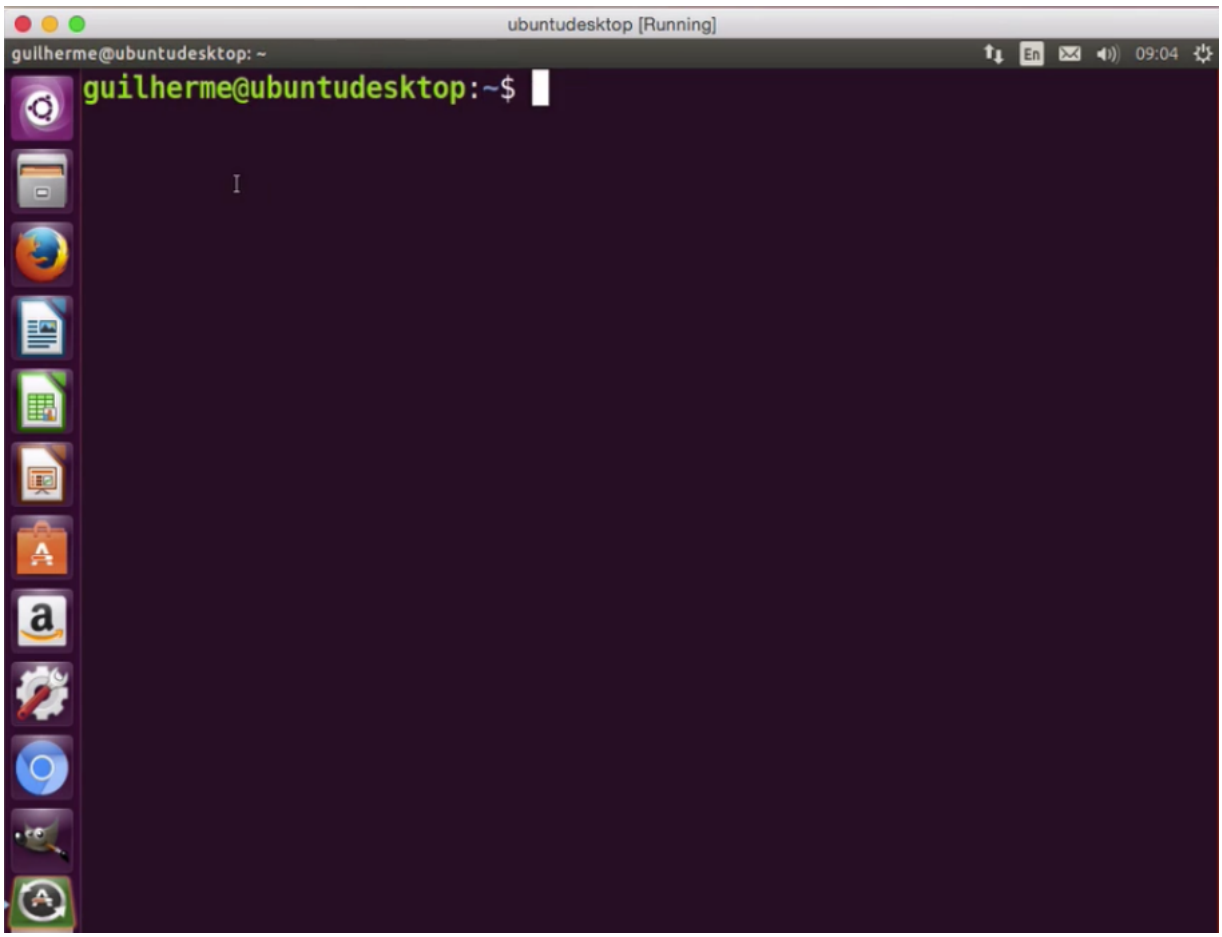
Vamos olhar esse arquivo?

Podemos copiar o `/home/guilhere/.bash_history` e como queremos editar esse arquivo digitaremos também `gedit`. Teremos:

```
> echo $HISTFILE
/home/guilhere/.bash_history.
> ^C
> gedit /home/guilherme/.bash_history
```

E abrirá um editor onde estarão discriminados todos os comandos que já utilizamos desde que o curso iniciou:

O `bash_history` mostra o histórico de todos os comandos que executamos no terminal, inclusive, vamos repara uma coisa! Se fecharmos o terminal e abrirmos um novo:



Teremos tudo novo, inclusive, um `bash` novo. Se pedirmos que `history 5` seja executado teremos a seguinte resposta:

```
history 5
    500 clear
    501 history 5
    502 echo $HISTFILE
    503 gedit /home/guilherme/.bash_history
    504 history 5
```

Temos nessa lista, inclusive o `gedit` que foi o último comando que executamos.

O que é importante lembramos?

O comando `history` que traz tudo.

O arquivo que armazena os comandos e que por padrão está na variável `HISTFILE` . E o padrão do arquivo é o diretório de base do seu usuário e `.bah_history` temos o seguinte padrão: `gedit /home/guilherme/.bash_history` .

Por fim, o último detalhe do `history` é que, se quero executar algum comando do `gedit` que executamos a muito tempo, como se ele estivesse muito para trás no nosso histórico, ou, "muitos para cima". Podemos usar o "Ctrl+R". O "Ctrl+R" começa uma busca reversa, uma busca de trás para a frente, isto é, do último comando para o primeiro.

Então, será que tem alguém que executou algum `gedit` ? Para verificar isso podemos dar um "Ctrl+R" e escrever qual o comando que quereremos buscar, no caso, o `gedit` .Por exemplo, observe:

```
gedit /home/guilherme/.bash_history
(reverse-i-search)`gedit`: gedit /home/guilherme/.bash_history
```

Repare que tínhamos acabado de executar um `gedit` na linha de cima e quando damos um "Ctrl R" é exatamente isso que ele mostra, `gedit gedit /home/guilherme/.bash_history` o que estava na nossa linha de cima.

Observe que para continuar buscando comandos podemos seguir dando "Ctrl R" e ele irá mostrar outros `gedit` :

```
gedit /home/guilherme/.bash_history
(reverse-i-search)`gedit`: gedit mostra_idade
```

E continuará mostrando os `gedit` que utilizamos se seguirmos apertando "Ctrl+R". Vamos só dar um "Ctrl+C" para não executar esse `gedit` .

```
> gedit /home/guilherme/.bash_history
(reverse-i-search)`gedit`: gedit mostra_idade: ^Cdit mostra
```

Então, revisando o uso do "Ctrl+R". Digitamos o atalho "Ctrl+R" para iniciar a busca e digitamos a parte que queremos buscar no histórico. É importante ressaltar que não é necessário escrever todo o carácter. Basta digitar uma parte, por exemplo, `dit` e teremos:

```
(reserve-i-search)`dit`: gedit /home/guilherme/.bash_history
```

Repare que apenas digitando o `dit` ele nos mostrou o `gedit` . Então, para voltar para trás usamos um "Ctrl+R", se queremos voltar mais, mais um "Ctrl+R" e mais para trás, mais um "Ctrl+R" e por aí em diante até que não tenham mais comandos para serem encontrados e ele nos indique `failed` :

```
(failed reserve-i-search)`dit`: ge^Ct mostra
```

E se quisermos executar algum comando que encontramos utilizando o "Ctrl+R"? Podemos, simplesmente, ao encontrá-lo, dar um "Enter". Por exemplo, usamos o "Crl+R" até encontrar o `gedit mostra_idade` e basta dar um "Enter" que ele começará a ser executado:

```
<img src="https://s3.amazonaws.com/caelum-online-public/essentiallinux3/essential4_2/essential4_2_4
```

Repare que poderíamos ter feito o seguinte, escrevermos primeiro o `gedit` e depois "Ctrl+R". Se fizermos apenas isso ele não começará a realizar a busca, e se dermos um espaço depois do `gedit` ele estará buscando o apenas o espaço, o começo que nós digitamos, no caso `gedit` , não será buscado. Mas, antes do comando vai ter o resto para trás.

```
(reserve-i-search)` `: gedit mostra_idade
```

Então, é importante lembrar a ordem: primeiro "Ctrl+R" e depois o que gostaríamos de buscar.

Então, vimos como funciona o histórico do nosso bash, do *shell*. Como ele é armazenado, onde ele é armazenado e como podemos acessar ele, comando a comando, utilizando o para cima e para baixo ou usando o "Crl+R", onde, ele realizará a busca para nós.

A variável de ambiente PATH

Já falamos sobre diversas variáveis, o `export`, o `history`, o `bash` e o `echo`.

Falta ainda a variável de ambiente, `PATH`. Vamos analisar esta última variável de ambiente.

Repare que quando executamos um comando, por exemplo, o `pwd`, o *shell* busca essa variável em algum lugar, ele encontra ela em algum lugar. Vamos perguntar para o `type` qual é o tipo de comando do `pwd`. Para isso, usaremos o `type`:

```
> pwd
type pwd
pwd is a shell builtin
```

Temos a resposta de que o `pwd` é um `shell builtin`, isto é, um comando de *shell*. Isso significa que o *shell* sabe quando executar esse comando,ou seja, é como se ele fosse o dono do `pwd`. Mas, vimos outros comandos também, por exemplo, o `clear`, se digitarmos `type clear` teremos a seguinte resposta:

```
> type clear
clear is hashed (/usr/bin/clear)
```

O `clear` é um comando *hasheado*. Vamos observar que resposta temos se digitarmos `type -a clear`:

```
> type -a clear
clear is /usr/bin/clear
```

Temos a resposta de que o comando `clear` é `/usr/bin/clear`. Esse comando, portanto, não é um `builtin` do *shell*, ele não é uma coisa que já vêm junto com o *shell*, é um comando do *Linux* e esse programa do *Linux* está localizado no `/usr/bin/clear`.

Lembra-se que vimos algo similar quando utilizamos o `zip`? Vamos digitar `type zip` e ver o que nos é contestado.

```
> type zip
zip is /usr/bin/zip
```

E lembra-se também que passamos pelos editores? Como o editor `vi`:

```
> type vi
vi is /usr/bin/vi
```

E vamos observar também o que temos quando digitamos o `type nano`:

```
> type nano
vi is /bin/nano
```

Repare que o `nano` se encontra localizado em outro lugar que é diferente do `/usr/bin` e ele está no `/bin/nano`.

Todos os comandos que estávamos executando até esse momento ou eram *builtin* do *shell* ou eram programas que estavam no `/usr/bin`. Agora, vimos que o `nano` é distinto, ele é um programa que está no `/bin`, um diretório que é distinto.

Então, quer dizer que quando executamos um comando `nano`, `vi`, esses comando estão sendo procurados, encontrados e executados. Isso ocorre em pelo menos dois diretórios diferentes. No diretório `/bin` de binário e no diretório `/usr/bin`.

Em que diretórios o *shell* busca os comandos? Em que caminhos o *shell* busca esses lugares?

Vamos limpar a tela usando um `clear`.

Como é caminho em inglês? É `PATH`, vamos então, digitar o `echo $PATH` e vamos observar o que temos:

```
> echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/bin:/usr/games:/usr/local/games
```

O que está no caminho do *shell*? No `PATH` do *shell*?

Temos no caminho `/usr/local/sbin`, repare que os `:` é o que demonstra a separação dos diretórios. Temos também o `/usr/local/bin`, o `/usr/sbin`, o `/usr/bin`. Vamos parar um pouco aqui, lembra-se que o `vi` estava aqui?

```
> type vi
vi is hashed (/usr/bin/vi)
```

O `vi` está hasheado no `/usr/bin/vi`. Encontramos o `vi` no `PATH`.

O que mais?

Temos também o `/sbin` e na sequência temos o `/bin`. O `/bin` é onde está o `nano`. Vamos confirmar?

```
> type nano
nano is hashed (/bin/nano)
```

Observe que o nano está cacheado no `/bin/nano`. Temos diversos diretórios onde ele busca os comandos que queremos executar. Vamos reparar nosso diretório atual, para verificar isso, utilizamos o `pwd`:

```
> pwd
/home/guilherme
```

Nosso diretório atual é o `/home/guilherme`. É feita uma busca em cada um dos diretórios que o `PATH` mostra para encontrar o que vai executar. É por isso que é usado o `hashed` e também o `cached`. Pois, toda vez que tentamos executar um comando, por exemplo o `unzip`.

O que teve que ser feito?

O `unzip` está no *shell*? É uma função? É uma função do *shell*? Não! Ele está localizado no diretório `/usr/local/sbin` ? Está no `usr/local/bin` ? Está no `/usr/sbin` ? Está no `/usr/bin` ? Sim!!!!

A informação do diretório é armazenada em um um `cached` para que a próxima que executarmos o `unzip` ele não tenha que procurar em todos os cantos novamente. Esse é o papel do `hashed` e `cached` , o de procura. Então, como, toda vez que executamos um comando ele tem que procurar isso no `PATH` inteiro, para que não se perca tempo com isso se cacheia isso em uma tabela de `hashed` , em uma estrutura. Temos um curso de estrutura que fala sobre isso, inclusive. A informação é cacheada em uma estrutura de dados.

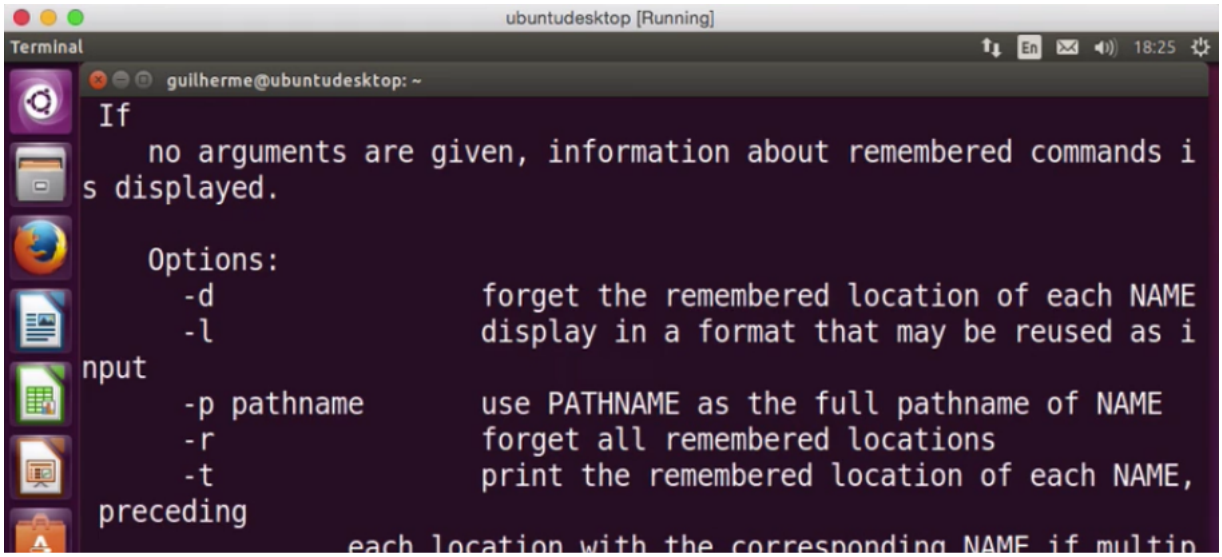
Para trabalhar com esse `cached` , vamos observar uma coisa. Por exemplo, vamos dar um `type zip` e um `type unzip` .

Teremos o seguinte:

```
> type zip
zip is /usr/local/bin
> type unzip
unzip is hashed (/usr/local/bin)
```

Repare que o `zip` está localizado no `/usr/local/bin` , mas quando perguntamos acerca do `unzip` , como ele já foi executado, essa informação acerca do lugar já está cacheada. Se quisermos trabalhar com o `hash` que é o `cache` , temos um programa chamado `hash` .

Vamos dar um `help` no `hash` e ver as explicações a repeito de seu funcionamento:



Existem várias opções para fazer o `hash`, e ele vai lembrar ou mostrar localizações de programas que ele está lembrando. E podemos através dele esquecer programas? Podemos!

Para isso utilizamos o `-d` para esquecer uma localização de cada nome que falarmos. E o `-r`, que serve para esquecer tudo.

Vamos testar isso! Já é conhecido o `type zip`, o `type unzip` e o `type nano`. Todos eles já estão cacheados! Ao final disso vamos usar o `hash -r`:

```
> type zip
zip is hashed (/usr/bin/zip)
> type unzip
unzip is hashed (/usr/bin/unzip)
> type nano
nano is hashed (/bin/nano)
> hash -r
```

O `hash -r` é responsável por limpar tudo. Vamos agora repetir os `type` e ver o que acontece:

```
> type zip
zip is (/usr/bin/zip)
> type unzip
unzip is (/usr/bin/unzip)
> type nano
nano is (/bin/nano)
```

Nada mais estará cacheado. Agora, se executarmos esses comandos novamente, eles aparecerão, mais uma vez cacheados:

```
> type zip
zip is hashed (/usr/bin/zip)
> type unzip
unzip is hashed (/usr/bin/unzip)
> type nano
nano is hashed (/bin/nano)
```

Qual era mesmo a opção do `hash` para remover um único nome? A opção `-d`. Vamos testar ela, pedindo `hash -d`, isto é, dizendo, `hash`, por favor, remove o `unzip`.

```
> type zip
zip is hashed (/usr/bin/zip)
> type unzip
unzip is hashed (/usr/bin/unzip)
> type nano
nano is hashed (/bin/nano)
> hash -d unzip
```

Como só removemos do `unzip` repare que se dermos um `type unzip` ele não estará mais cacheado. E se dermos um `type zip`, ele continuará cacheado.

```
> hash -d unzip
unzip is /usr/bin/unzip
> type zip
zip is hashed (/usr/bin/zip)
```

É importante saber que ele busca no `PATH` e armazena onde o programa ou o comando está localizado. Então, por exemplo `zip` está armazenado no `/usr/bin/zip` e já se sabe que está cacheado.

Vamos observar o que podemos realizar com o `PATH` ?

Vamos tentar colocar o nosso diretório atual no `PATH` , lembrando que para conferir o diretório basta utilizar o `pwd` , que nos informará que o nosso diretório atual é o `/home/guilherme/` .

Então, vamos ver qual é o `PATH` atual, para isso, basta digitar o `echo $PATH` .

Teremos:

```
> pwd
/home/guilherme/
> echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/bin:/usr/games:/usr/local/games
```

O que queremos fazer é que o `PATH` seja o `PATH` atual e que o diretório seja `home` , para isso, digitaremos `PATH=$PATH:/home/guilherme` . Teremos:

```
> pwd
/home/guilherme/
> echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/bin:/usr/games:/usr/local/games
> PATH=$PATH:/home/guilherme
```

Agora, temos o diretório `/home/guilherme` também no `PATH` . Podemos tentar executar o programa `mostra_idade` ?

Ao tentarmos executar o `mostra_idade` teremos a seguinte resposta:

```
> mostra_idade
bash: /home/guilherme/mostra_idade:Permission denied.
```

Ele nos dirá que não podemos executar o `mostra_idade` , isso ocorre pois não temos a permissão de execução desse programa.

Vamos falar mais acerca de permissões de execução mais adiante.

O que vamos fazer? Vamos adicionar a permissão de execução através de `chmod +x mostra_idade` . O `+x` é de executar. Agora sim, se chamarmos apenas o `mostra_idade` ele nos mostra `voce tem anos` :

```
> chmod +x mostra_idade
> mostra_idade
```

```
voce tem anos
```

Como já mencionado, falaremos sobre permissão mais adiante, teremos um setor só para falar sobre isso.

Mas apenas retomando, adicionamos a permissão de execução no `mostra_idade`. Ao executar o `mostra_idade`, ele busca no `$PATH`. E onde será que ele achou o `mostra_idade`?

Vamos dar um `clear` e digitar `type mostra_idade`. Ele encontrou o `mostra_idade` no `(/home/guilherme/mostra_idade)`:

```
> type mostra_idade
mostra_idade is hashed (/home/guilherme/mostra_idade)
```

E onde o `zip` foi encontrado?

```
> type zip
zip is /usr/bin/zip
```

Repare que o `zip` ainda não está cacheado.

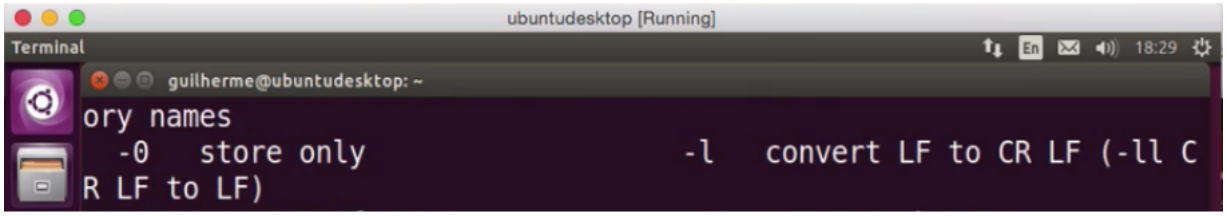
Vamos tentar fazer uma coisa? Vamos tentar criar um arquivo chamado zip no nosso diretório? Vamos editar um arquivo, para isso digitamos `gedit zip` e na tela do nosso editor escrevemos apenas `echo zip do Guilherme`, somente para mostrar que estamos executando esse arquivo, vamos salvar isso e fechar o editor.

Se dermos um `ls`, veremos que nosso arquivo recém criado está na lista de arquivos:

```
> gedit zip
> Desktop       examples.desktop        mostra_idade        Public      zip
Documents       loja                    Music               Templates
Dowloads        mostra_idade            Pictures            Videos
```

Vamos dar a permissão de execução, utilizando o `chmod +x zip`. E se executarmos `zip`, o que ele vai executar? Para executar escrevemos, simplesmente `zip` e damos um "Enter".

Vai executar o `zip` antigo:



Por que isso acontece? Lembra que quando vimos a variável ele primeiro procurá nos diversos diretórios. Lembra-se de como estava o `PATH` ? Para acessá-lo basta digitarmos `echo $PATH`

```
> echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/bin:/usr/games:/usr/local/games
```

O `zip` é encontrado no diretório `/usr/bin` , ele não chega ao `/home/guilherme` .

A ordem dos diretórios no `PATH` é extremamente importante.

Então, repare que agora, se dermos um `type zip` , como já o executamos, anteriormente, ele estará cacheado já no `usr/bin` :

```
> type zip
zip is hashed (/usr/bin/zip)
```

Então, vamos tentar alterar o `$PATH` . Faremos isso dizendo que o `$PATH` é primeiro `/home/guilherme` e depois o resto. Escreveremos isso da seguinte maneira: `PATH=/home/guilherme:$PATH` . Na sequência digitaremos um `echo $PATH` para observar como ele ficou:

```
>PATH=/home/guilherme:$PATH
echo $PATH
/home/guilherme:/usr/local/sbin:/usr/local/bin:/usr/local/bin:/usr/local/bin:/usr/sbin:usr/bin: /sbi
```

Temos que primeiro de tudo é o `/home/guilherme` e na sequência temos o restante.

Vamos ver o que acontece se tentamos executar o `zip` ?

```
> zip
Zip
do Guilherme
```

Repare que agora ele executa, primeiro, o `zip` do diretório `/home/guilherme` e não outro zip.

E se dermos um `type zip` teremos o seguinte:

```
> type zip
zip is hashed (/home/guilherme/zip)
```

Observe que ele está *hasheado*. Ele percebeu que o `PATH` alterou , procurou o `zip` , achou o `zip` novo e usou ele para cachear.

Então, finalmente, foi percebido que o `/home/guilherme/zip` é o programa que temos.

Então, vimos o `hash -r` para remover tudo, o `hash -d` para remover uma única parte. O `type` para saber de onde ele é e que tipo de comando ele é. Vimos o `PATH` onde podemos observar vários tipos de diretórios e utilizamos esses diretórios separados por dois pontos. É muito comum `setar` o `PATH` utilizando esse padrão:

```
PATH=diretorio: diretorio atual
```

Em geral, colocamos o `PATH` que queremos, novo, no final. Então, repare que não é o padrão do que utilizamos dessa última vez, mas sim, na primeira vez: `PATH=$PATH:/home/guilherme` . Então, frisando a ordem, primeiro o `$PATH` atual e depois o `/home/guilherme` , no geral adicionamos nossos diretórios no final e não, no começo.

E, o que acontece, se removermos o `zip` ? Vamos usar o `rm zip` para removê-lo e em seguida daremos um `type zip` :

```
> rm zip
> type zip
zip is hashed (/home/guilherme/zip)
```

Repare que ele continua cacheado no diretório.

Mas, então, vamos tentar executar o `zip?`

Não irá executar. Por mais que exista o `/home/guilherme` o *shell* cacheou essa informação, então quando tentamos executar o `zip` ele pega a informação do `cache` , mas por que agora ele pegou do `cache` ? E quando mudamos o `PATH` ele procurou novamente?

Porque é perceptível a mudança no `PATH` e, assim, aquele `cache` anterior não vale mais. Porque alteramos `PATH` , as coisas mudaram, a ordem de precedência é alterada. Toda vez que mudamos o `PATH` , tiramos do `cache` .

A prova provavelmente não focará na questão do `cache` . É mais provável que sejam cobrados conteúdos acerca do `PATH` . O `cache` pode ser secundário, mas o `PATH` é extremamente importante na prova. Pois ele é muito utilizado no dia a dia.

Então, `echo $PATH` para ver o `PATH` atual e `PATH=$PATH:diretóriosnovos que queremos`, lembrando que os diretórios devem ser separados utilizando-se os dois pontos.

Por que o `cache` é importante para nós? As vezes, quando não mudarmos o `PATH`, mas ficarmos movendo os programas de diretório talvez fique no `cache` alguma coisa e aí ele não encontrará, e isso acontecerá por que movemos as coisas de lugar. Observe:

```
> zip
bash: /home/guilherme/zip: No such file or directory
```

Nesse caso, o que temos que fazer é remover do `cache`, utilizando, para tanto, um `hash -d zip` e agora executamos o `zip` novo:

```
> hash -d zip
> zip
```

Ou, simplesmente fechamos o terminal e começamos um *enviroment* novo:

```
> type zip
zip is /usr/bin/zip
```

Podemos observar que nesse novo terminal o `zip` não está cacheado.

Mais um detalhe: no *Windows* é muito comum que coloquemos o diretório atual no `PATH`. O diretório atual no `PATH` é identificado usando-se simplesmente um ponto.

No *windows* é muito comum encontrarmos algo como: `PATH=$PATH:.`. Assim, o diretório atual está no `PATH`:

```
> PATH=$PATH:.
> ls
Desktop         Examples.desktop        mostra_idade        Public
Documents       loja                    Music               Templates
Dowloads        mostra_idade            Pictures            Videos
```

Se colocamos o diretório atual no `PATH`, podemos simplesmente executar o `mostra_idade`, pois ele está no diretório atual, e ele executa isso nos mostrando `voce tem anos`.

```
> PATH=$PATH:.
> ls
Desktop         Examples.desktop        mostra_idade        Public
Documents       loja                    Music               Templates
Dowloads        mostra_idade            Pictures            Videos
> mostra_idade
voce tem anos
```

Mas, se entrarmos no diretório `loja` utilizando o `cd loja`, o `mostra_idade` não funciona mais.

```
> cd loja
> ~/loja$ mostra_idade
mostra_idade: command not found
```

Ele não executa esse comando pois ele não está no diretório atual, não está `PATH` .

Se você digitar o `type mostra_idade` , perceba que ele não encontra o `mostra_idade` :

```
> type mostra_idade
bash: type: mostra_idade: not found
```

Se voltarmos para o diretório anterior, através do `cd ..` e dermos um `type mostra_idade` , teremos o `mostra_idade` *hasheado* no diretório atual:

```
~/loja$ cd ..
> type mostra_idade
mostra_idade is hashed (./mostra_idade)
```

Então, fica um comportamento bem estranho, colocar o ponto no diretório atual. Na verdade, isso é bastante complicado, no mundo *Linux*, não é nada comum, colocar o diretório atual no `PATH` , isto é, colocar o ponto no `PATH` . No *Windows* usar o `:.` é ok, mas no *Linux* isso não é um padrão comum.

Vamos fechar esse terminal. Repare que se fecharmos e abrirmos o terminal, estaremos criando um *shell* novo e essa variável será zerada, nosso `PATH` estará limpinho. Para testar isso basta digitar `echo $PATH` :

```
> echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

E como fazemos, então, para executar um arquivo desse diretório?

Vamos observar em qual diretório estamos, para saber isso usaremos o `pwd` :

```
> pwd
/home/guilherme
```

Repare que estamos no diretório `/home/guilherme/` . Então podemos executar o `/home/guilherme/mostra_idade` ? Sim! Pois temos permissão de execução! Ele nos responderá que temos: `voce tem anos` .

```
> /home/guilherme/mostra_idade
voce tem anos
```

Ou, podemos falar, usando um ponto, para indicar o diretório atual, `./mostra_idade` e pedir para executar.

São as diversas maneiras de chamar e executar um programa que possui permissão de execução. Vimos diversas configurações do nosso `PATH` e diversas maneiras de usar e evocar nossos programas.

Isso é extremamente importante no nosso dia a dia!

É sugerido que se teste todas essas variações que foram citadas anteriormente. Testar com um `PATH` adicionando diretórios que possuem o mesmo comando e observar qual terá a precedência, qual ele ir a preferir devido a ordem, podemos realizar diversas modificações no `PATH` .

Na prática colocamos apenas os diretórios absolutos para, justamente, não cair em confusão.

Na sequência veremos outros programas que nos ajudam a entender quem está sendo executado.

Whereis which builtin e look up de comandos e builtins no bash

Já vimos a importância da variável de ambiente, `PATH` . Toda vez que executarmos um comando, por exemplo, o comando `zip` , o *shell* busca em cada um desses diretórios se existe um comando `zip` . E se existir um comando `zip` ele será executado no primeiro diretório.

Lembrando o que o `PATH` nos mostra:

```
> echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/bin:/usr/games:/usr/local/games
```

Mas, quero saber qual dos `zip` , exatamente, ele vai executar. Vamos dar um `type zip` e teremos como resultado, `zip is hashed (/usr/bin/zip)` . Teremos o seguinte:

```
> type zip
zip is hashed (/usr/bin/zip)
```

E se tivéssemos um `zip` também em outro diretório? Por exemplo, no diretório atual. Qual dos dois seria executado? Vamos dar uma olhada?

Vamos verificar qual é o diretório do `zip` utilizando, para tanto, o `pwd` :

```
> pwd
/home/guilherme
```

Vamos editar o `zip` , usando o `gedit zip` . Abrirá o editor e acrescentaremos apenas uma mensagem: `echo Meu zip` , salvaremos e fecharemos o editor. Vamos configurar para execução `chmod =x zip` e adicionaremos também o diretório fixo `PATH=$PATH:/home/guilherme` :

```
> gedit zip
> chmod =x zip
> PATH=$PATH:/home/guilherme
```

E agora, se dermos o `type` do `zip` ? Teremos o seguinte:

```
> type zip
zip is /usr/bin/zip
```

A resposta que temos é que o `zip` é `/usr/bin/zip`. Mas, você pode estar se perguntando por que ele perdeu o `hasheado`?

Bom, isso acontece porque trocamos `PATH=$PATH:/home/guilherme` e o *shell*, ou seja, o *bash*, teve que recalcular onde estava o `zip` que queremos. Então, ele nos aponta onde está o `zip` que queremos.

Se quisermos saber todos as opções de onde está o `zip` podemos utilizar o `type -a`, o `a` é de `all`. Digitaremos, `type zip -a` e teremos o seguinte:

```
> type -a zip
zip is /usr/bin/zip
zip is /home/guilherme/zip
```

Agora, temos a localização de todos os comandos definidos como `zip`.

Voltamos a nossa pergunta inicial: Qual o `zip` que vai ser executado?

A essa pergunta podemos responder utilizando o `which`, que em inglês significa "qual". Então, digitamos `which zip` e teremos o seguinte:

```
> which zip
/usr/bin/zip
```

O `which` nos responde qual o `zip` que será executado, nesse caso, é o que está em `/usr/bin`.

O `which`, portanto, informa qual é o *script* ou o programa que será executado. Se escrevermos um comando que não existe ele não irá se pronunciar, pois, ele não irá encontrar o que digitamos. Esse é o comportamento do `which`.

E o `pwd`, lembra-se dele? Se digitarmos `type pwd` teremos que ele é um `shell builtin`.

```
> type pwd
pwd is a shell builtin
```

E se dermos `type -a pwd`? Teremos que ele é um `shell builtin`, mas ele também é `/bin/pwd`:

```
> type -a pwd
pwd is a shell builtin
pwd is /bin/pwd
```

E se perguntarmos `which pwd`, para saber qual ele executará?

```
> which pwd
/bin/pwd
```

Ele executará o `/bin/pwd`.

Além do `pwd` que outros comandos vimos?

O `ls`. Vamos perguntar `which ls`:

```
> which ls
/bin/ls
```

Teremos que o `ls` que será executado é o `/bin/ls`.

E se dermos um `which` em um outro comando, um comando que também é um `shell builtin` e que possui um arquivo, por exemplo, como é o `pwd`.

Existem alguns comandos que são apenas *shell builtin* como, por exemplo, o `help`. Se dermos um `type help`, veremos que ele é um `shell builtin`. E se dermos um `type -a help` veremos que ele é só um `shell builtin`.

```
> type help
help is a shell builtin
> type -a help
help is a shell builtin
```

Já o `pwd`, se dermos um `type -a pwd` veremos que ele é `shell builtin` e é, também, `/bin/pwd`:

```
> type -a pwd
pwd is a shell builtin
pwd is /bin/pwd
```

Vemos que além dele ser um `shell builtin` ele possui um *script* binário no `/bin/pwd`.

Então, se pedirmos `which help`, não teremos resposta, pois não temos nenhum `help` no `PATH`.

Mas, qual a diferença entre o `type` e o `which`?

O `type` procura funções, programa e nos informa o que ele é e onde ele pode estar. Por exemplo, o `zip`:

```
> type -a zip
zip is /usr/bin/zip
zip is /home/guilherme/zip
```

Ele nos mostra que o `zip` está em dois diretórios diferentes, o `/usr/bin/zip` e `/home/guilherme/zip`.

E agora, o que o `which` faz?

O `which` ignora os `shell builtin` e outras coisas. Ele procura arquivos executáveis no `PATH`. Ele utiliza o mesmo algorítimo do *bash* em todos os diretórios do `PATH`. Então, se procuramos algo, como o `zip`, por exemplo, ele encontra o primeiro `zip` e nos passa a informação dele, de onde ele está localizado. E isso ocorre também com o `pwd`, mas se testarmos com o `help`, ele não nos falará nada.

```
> which zip
/usr/bin/zip
> which pwd
```

```
/bin/pwd
> which help
```

Não temos informações acerca do `help`, pois, ele não é um programa, não é um arquivo. É apenas um `shell builtin` e mais nada.

E se quiséssemos ver as duas variações de `zip` que ele encontra e que vimos um pouco mais a cima com o `type -a zip`? Teremos que digitar, `which -a zip` e ele nos mostrará os dois `zips` que ele encontra:

```
> which -a zip
/usr/bin/zip
/home/guilherme/zip
```

Ele mostrará os dois programas encontrados. Repare que, o que nós criamos é o `/home/guilherme/zip`, e o outro, é o programa binário, o `/usr/bin/zip`.

Esse é o uso do `which` no dia a dia, ele nos informa onde está o programa que desejamos executar e ele acha esse programa.

Repare que colocamos um arquivo chamado `zip` no diretório `/home/guilherme/zip`, mas o `PATH`, dá preferência para outros diretórios, observe que `/home/guilherme/zip` é o último.

```
> echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/bin:/usr/games:/usr/local/games:/home/
```

Vamos fechar esse terminal e vamos abrir um novo terminal. E, agora, vamos editar um outro arquivo, o arquivo chamado `help`. Vamos digitar `gedit help` e quando abrir o editor vamos escrever apenas `echo help do Guilherme` no editor.

Voltando no terminal escreveremos a permissão da execução `chmod +x help`:

```
> gedit help
> chmod +x help
```

E vamos também colocar isso no `PATH`, vamos escrever `PATH=/home/guilherme:$PATH`.

Com isso estamos dizendo que primeiro virá o `/home/guilherme` e depois o resto. Quem é o `help` que será executado?

Vamos observar o que o `type -a help` nos informa:

```
> type -a help
help is a shell builtin
help is /home/guilherme/help
```

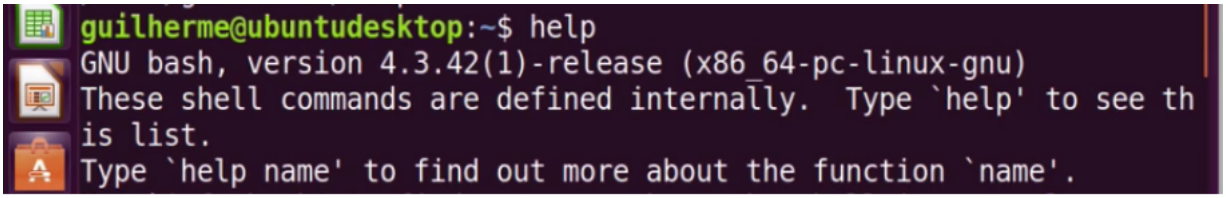
Ele encontrou um `shell builtin` e `/home/guilherme/help`.

Vamos dar um `which` do `help` e veremos que agora ele encontrará algo, diferente de antes, quando não encontrava nada:

```
> which help
/home/guilherme/help
```

Ele encontra o `/home/guilherme/help` .

E se evocarmos o `help` ? Qual dos dois ele irá evocar? Para evocar basta digitar `help` e dar um "Enter"



Ele evoca o `help` original, aquele que é o `shell builtin` e ignora o `/home/guilherme/help` .

Repare que, por mais que tenhamos esse arquivo `/home/guilherme/help` e mesmo que ele esteja no `PATH` , o `help` é um `shell` builtin e o *bash* da preferência para o `shell builtin` .

Se agora executarmos novamente o `type -a help` teremos:

```
> type -a help
help is a shell builtin
help is /home/guilherme/help
```

Repare que essa informação não é nem cacheada, continua sendo `shell builtin` e, a cima de tudo, um `shell builtin` . Lembre-se que o *shell* primeiro faz uma busca pelos `shell builtin` . Se colocarmos apenas o nome do comando ele procura, primeiro, o `shell builtin` e depois busca no `PATH` .

Mas, se quisermos executar o `help` que é `/home/guilherme/help` teremos que digitar seu nome completo:

```
> /home/guilherme/help
Help do Guilherme
```

Assim, estaremos falando explicitamente que gostaríamos de executar o comando que está nesse diretório `/home/guilherme/help` e, então, ele executa o *script* para nós.

O *bash*, primeiro, busca os `builtin` do *shell* e quando ele os encontra ele executa esses *bash* e, se ele não encontra, ele tenta executar o nosso comando.

Agora, e se quiséssemos forçar alguma das duas coisas? Por exemplo, queremos que seja executado o `builtin` e não o comando.

Se quero que o `builtin` é que seja executado podemos digitar `builtin help`.

```
> builtin help
```

Teremos o seguinte:

Podemos forçar o comando para que ele seja executado. E da mesma maneira podemos tentar executar um `builtin` de `pwd`, por exemplo:

```
> builtin pwd
/home/guilherme
```

Mas, se tentassemos executar o `builtin` do `zip` veremos que ele não será executado. É só para confirmarmos isso, podemos verificar através do `type -a zip` que o `zip` não é um `builtin` do `shell`:

```
> type -a zip
zip is /home/guilherme/zip
zip is /usr/bin/zip
```

O `zip` é um programa que possui vários diretórios, mas vamos observar também o `pwd`:

```
> type -a pwd
pwd is a shell builtin
pwd is /bin/pwd
```

O `pwd` é um `builtin` e ele executa `builtin pwd`. Os outros, não são.

Precisamos observar aqui um detalhe importante. Quando é necessário se buscar algo isso é feito passando um diretório de cada vez. E quem implementa isso é o `which`. Podemos perguntar para ele onde está o `pwd` no `PATH`:

```
> which pwd
/bin/pwd
```

Podemos perguntar também onde está o `help`:

```
> which help
/home/guilherme/help
```

Podemos perguntar também acerca do `zip`:

```
> which zip
/home/guilherme/zip
```

O `which` no responde mostrando o primeiro diretório que encontra, mas se queremos visualizar todos, basta utilizar o `which -a` que ele nos mostrará todos que forem encontrados no `PATH`. E esses diretórios serão exibidos na ordem em que são encontrados.

E, na hora que executarmos um `zip`, isso significa que ele vai executar o primeiro `zip` que aparece listado no `which`? Não temos como saber. Pode ser que o `zip` possa ter um `builtin` e sabemos que o `bash` dá preferência ao `builtin` por padrão. Vamos ver:

```
> zip
Meu zip
```

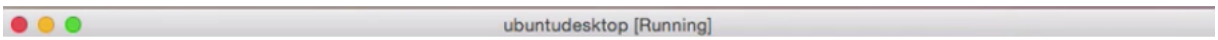
Veja que o `zip` que foi executado é o `Meu zip`, que não é um `builtin`. Basta para conferir isso que testemos usando o `type -a zip`:

```
> type -a zip
zip is /home/guilherme /zip
zip is /usr/bin/zip
```

E se dermos um `type -a help`, verificamos que ele é um `builtin`:

```
> help is a shell builtin
help is /home/guilherme/help
```

Então, já sabemos que no caso do `help` ele irá, primeiro, buscar aquele que é um `builtin`. Vamos executar só o `help`? Bom, para isso, basta escrever `help` e dar "Enter" que teremos o seguinte:



Então, a ordem de procura do *bash* é primeiro `builtin` e depois programas.

Para `builtin` ele já sabe quais são os `builtin`, como o `echo`, o `pwd`, o `cd` e assim por diante... Mas para os arquivos que queremos executar o `which` é capaz de procurar no `PATH` quem é que ele vai executar. O `which -a` procura todos eles.

Por fim, podemos querer saber em que diretório estão os arquivos relacionados a um programa que estamos executando. Podemos utilizar o `whereis`, isto é, onde está, o nome de um comando ou programa. Podemos, por exemplo, digitar, `whereis zip`, que significa, onde está o zip?

```
> whereis zip
zip: /usr/bin/zip /home/guilherme/zip /usr/share/man/man1/zip.1.gz
```

Ele nos mostra que o `zip` está em `usr/bin/zip`, e em `/home/guilherme/zip` e temos também um manual do `zip`, o `/usr/share/man/man1/zip.1.gz`.

Podemos perguntar acerca do `pwd` também:

```
> whereis pwd
pwd: /bin/pwd /usr/include/pwd.h /usr/share/man/man/pwd.1.gz
```

Ele nos responde que está em `/bin/pwd`, em um cabeçalho `/usr/include/pwd.h` e também em um manual, `/usr/share/man/man/pwd.1.gz`

O `whereis` procura onde estão os arquivos relacionados ao arquivo ou a função que estamos executando.

Então, `whereis help` nos diz o seguinte:

```
> whereis help
help: /usr/share/help /home/guilherme/help
```

E ele nos fornece as informações do `help`. Então, temos o programa em si e temos sua documentação.

Então o conjunto `which` que mostra quais são os que serão executados e o `whereis` que mostra onde estão os arquivos ao comando que estamos executando. Vejamos, por exemplo, como isso funciona para o `zip`

```
> which -a zip
/home/guolherme/zip
/usr/bin/zip
> whereis zip
zip: /usr/bin/zip /home/guilherme/zip /usr/share/man/man1/zip.1.gz
```

Veremos mais adiante sobre o manual e sobre como buscar ajuda para comandos que não são `shell builtin`. Por enquanto, vimos apenas o `help`. E o `help`, é apenas para `shell builtin`. Para o `pwd` o `help` pode ser utilizado. Mas, o `help` para o `zip` ou para o `whereis` não funciona, pois eles não são `builtin`, eles são comandos e programas que temos para executar.

E, para isso, temos outro tipo de ajuda que veremos mais adiante.

Observe o que aparece se digitamos `help pwd`:

Mas teremos o seguinte se pedimos `help zip` e `help whereis`:

```
> help zip
bash: help: no help topics match `zip`. Try `help help` or `man -k help zip` or `info zip`.
> help whereis
bash: help: no help topics match `whereis`. Try `help help` or `man -k help whereis` or `info where
```

Então, vimos o `which` e o `whereis`, além disso, comentamos também sobre o `builtin` para executar explicitamente ele quando quisermos. Mais adiante, veremos, ainda, outros aspectos do *shell*, para depois partirmos para as demais "ajudas".

Autocomplete

Já vimos uma série de termos e suas utilidades, o `bash`, o `echo`, o `history`, o `PATH`, o `export`, o `type`, o `basic shell`, o `command line synthax` e algumas variáveis. Bom, ainda falta analisarmos o `globbing` e o `quoting`:

Antes de falarmos sobre esses dois itens, vamos falar de mais alguns detalhes que usamos no dia a dia de um *shell*. A medida que estamos trabalhando com o *shell* é bem comum que queiramos utilizar comandos. Por exemplo, queremos executar um comando, o `whereis`. Podemos digitar `whereis` letra por letra, ou, podemos utilizar o "Tab" para completar esse nosso comando. Digitaremos apenas `whe` e "Tab".

A tecla "Tab" funciona como um *autocomplete*, ela busca completar automaticamente aquilo que estamos digitando.

A primeira palavra costuma ser um comando, então, usando a tecla "Tab" que tenta completar a palavra usando os nomes de comandos disponíveis. Por exemplo, quais serão os comandos existentes que iniciam por "whe"? Temos entre os `builtin` `shell` e outros programas, arquivos e comandos o `whereis`. Então, utilizando o "Tab", `whereis` já é preenchido automaticamente.

O que se busca mostrar aqui é que a primeira palavra, isto é, as sugestões vêm dos programas que podemos executar. Por exemplo, queremos executar um `builtin` que se chama `help`. Vamos escrever, apenas para testar, `hel` e damos um "Tab". Ele completa e temos `help`. Podemos dar um "Enter" que ele executará esse comando, poderíamos, ainda, completar esse comando.

Teremos a seguinte tela após digitar `help` e dar um "Enter":

O "Tab" tenta completar o que digitarmos. Se existe apenas uma palavra ele completa inteira essa palavra, por exemplo, se digitamos apenas o `whe` ele completa com o `reis` e temos `whereis`.

Lembra-se que vimos o `builtin`?

Vamos lembrar qual era o `type` do `bultin` digitando `builtin type` teremos a resposta de que ele é um `shell builtin`.

Teremos:

```
> type builtin
builtin is a shell builtin
```

E se digitarmos apenas o `bui` e dermos um "Tab" ele completa `ltin` e teremos `builtin`. Ele completa pois só existe um comando que começa com `bui` e que está como co-função do `shell` ou arquivo, programa a ser executado. Então, além de completar ele ainda coloca um espaço, pois já sabe que temos que completar o `builtin` com um parâmetro.

Vamos colocar um parâmetro nesse `builtin` que o "Tab" completou para nós, escreveremos `builtin pwd`. Ficaremos com:

```
> builtin pwd
/home/guilherme
```

O "Tab" é extremamente importante para completar o nome dos comandos. Pois, nem sempre o nome do comando é tão simples, as vezes o comando pode ser bem mais extenso. Mas, nem sempre, queremos executar um comando que é solto.

Vamos digitar `ls` e lembrar do que temos aqui:

```
> ls
Desktop         examples.desktop        mostra_idade        Pictures        Videos
Documents       help                    mostra_idade        Public          zip
Dowloads        loja                    Music               Templates
```

Em nosso diretório atual temos o `mostra_idade`, vamos lembrar qual é nosso diretório através do `pwd`?

```
>pwd
/home/guilherme
```

Então queremos executar no `/home/guilherme` o `mostra_idade`. Basta escrever uma parte do comando, por exemplo após o `/home/guilherme` digitando apenas um `most` e dando um "Tab" ele já completa com o restante, `ra_idade`. E ele executará `/home/guilherme/mostra_idade`.

```
> /home/guilherme/mostra_idade
voce tem anos.
```

Então, o *bash* tenta buscar primeiro o comando, então, se começamos a digitar o `whe` ele vai buscar no `PATH` a continuação, `whereis`. Mas, se digitamos um diretório específico como o `/home/guilherme/` e tentarmos dar um "Tab" após digitar `whe`, não vai acontecer nada, ele não vai completar com `reis`. Isso acontece, pois, nesse diretório nós não temos o `whereis`. Se digitamos, inicialmente, um diretório, o "Tab" procura o que tem de executável nesse diretório. Nesse caso, nós só temos no `/home/guilherme/` o `help`, o `zip` e o `mostra_idade` que você deve lembrar-se, críamos para testar outros *scripts*.

Se digitarmos `/home/guilherme/mostra` e dermos um "Tab", teremos a complementação disso e ficaremos com `/home/guilherme/mostra_idade`.

O "Tab" tenta preencher um comando seja do `shell builtin` seja um comando do `PATH`.

Por exemplo, podemos digitar, `whe` e dar um "Tab", tecla de *autocomplete*, que ele completará o comando e ficaremos com `whereis` e ele permite que executemos esse comando após preenche-lo.

Se digitarmos o `/home/guilherme/` e dermos um "Tab" ele mostra todos os scripts desse diretório. Completaremos usando o `mostra_idade`.

Repare que digitar `/home/guilherme/mostra_idade` ou mesmo `/home/guilherme/m` é grande, então, o que podemos fazer é digitar apenas o `/h`. O que estamos digitando aqui? Não é um diretório? Sabe-se disso porque iniciamos com uma barra (`/`).

Como ele sabe que é um diretório, usando o "Tab", após o `/h` teremos o restante completado e ficaremos com `/home/`, digitando apenas o `gui` e dando um "Tab" teremos `guilherme/` e, por fim, escrevendo apenas `mo` e pressionando um "Tab" teremos `mostra_idade`. Ele reconhece que nesse instante estamos digitando um diretório, ele sabe que temos que preencher a partir do diretório e o "Tab" nos auxilia nisso, independente de ser o primeiro ou segundo argumento.

Mas, temos um caso mais detalhado. Repare que escrevemos o `whereis` digitando `whe` e dando um "Tab". Se digitarmos apenas o `wh`, ele poderia ser completado com o `which` ou o `whereis`, pelos menos.

Nesse caso, se dermos um "Tab" após o `wh`, ele não vai fazer nada, mas se dermos mais um "Tab", ele vai mostrar tudo o que tem disponível de sugestão que começa com o `wh`. Como é um primeiro argumento, são só `builtin` e os comandos que são executáveis e estão no `PATH`:

```
> wh
whatis          while           whoami
whereis         whiptail        whoopsie
which           who             whoopsie-preferences
```

É mostrado tudo o que está configurado no *Ubuntu Desktop*, no `PATH` ou no `builtin` do `bash`, do *shell*.

Então, reparem que se dermos dois "Tabs", a medida que começamos a dar "Tabs" ele nos ajuda. Então, a partir de agora se digitarmos um `whi` e dermos dois "Tabs" ele nos mostrará quais as opções que temos. O primeiro "Tab" ele ignora, o

segundo ele já nos mostra quais são as opções para completar o `whi`:

```
> whi
which       while       whipail
```

Aí, podemos completar com o `ch`, ficaremos com `which` e seguimos adiante.

O "Tab" ajuda bastante quando sabemos que o comando que queremos digitar é único, ou aquele diretório é único no local que estamos buscando. Mas, se não temos certeza disso podemos dar dois "Tab" e temos uma dica sobre o que tem disponível que comece com as iniciais que você digitou.

Vamos testar com algumas letras, `/h` e "Tab", teremos `/home`, `/g` e "Tab" ficaremos com `/guilherme` e, por fim, `m` e "Tab", completa-se com o `mostra_idade`. Assim, ficamos com `/home/guilherme/mostra_idade`.

Podemos digitar apenas uma barra `/` e se dermos dois "Tab" teremos uma lista:

```
> /
bin/        home/           mnt/        sbin/       var/
boot/       lib/            opt/        srv/
cdrom/      lib64/          proc/       sys/
dev/        lost+found/     root/       tmp/
etc/        media/          run/        usr/
```

Quando digitamos duas vezes o "Tab" é mostrado tudo o que temos no diretório raiz e que pode ser completado.

E se digitarmos `/h`, ele completa através do "Tab" o `ome`, ficando `/home/guilherme/`. Ele já traz o `guilherme` pois é o único diretório que existe no diretório `home`. E se dermos um "Tab" depois de `/home/guilherme/`? Ele não nos mostra nada. E se dermos mais um "Tab"? Ele nos mostra quais são as opções que temos para completar isso:

```
> /home/guilherme/
.cache          .gimp-2.8/          Music/          .thunderbird/
.config/        help                .nano/          Videos/
Desktop/        .local/             Pictures/       zip
Documents/      loja/               .pki/
Dowloads/       mostra_idade        Public/
.gconf/         .mozilla/           Templates/
```

No segundo "Tab" ele nos mostra tudo o que podemos fazer, temos todos esses diretórios que aparecem para navegar ou podemos executar o `mostra_idade`, o `help` ou o `zip`. O restante são diretórios.

Vamos executar o `mostra_idade`, então damos apenas um "Enter" e teremos:

```
> /home/guilherme/mostra_idade
voce tem anos
```

Um "Tab" funciona como *autocomplete* e dois "Tabs" servem para mostrar no *autocomplete* todas as opções que temos. Como o primeiro argumento, a primeira palavra que escrevemos no *Bash*, permite que digitemos uma barra, `/`, e a partir daí os diretórios que queremos executar ou simplesmente o nome de um comando de um `shell builtin`.

E no caso de um segundo argumento?

Se digitarmos `echo` o que será que ele mostra como *autocomplete*? Por exemplo, se digitarmos `mo`, ele mostra um arquivo, o `mostra_idade`.Em um segundo argumento ele permite qualquer arquivo que tenhamos no diretório atual.

Não significa que se tentarmos completar com o `echo whe` e dermos um "Tab" que isso será completado. Não será mostrado nada, pois, não tem um `whereis` que possa ser completado nesse segundo argumento. A partir do segundo argumento, ele sabe que é um argumento para o programa e que nesse caso não cabe um `whereis`, e sim, qualquer outra palavra que queremos digitar. Mas, se essa palavra se assemelhar ao nome de algum arquivo, então, temos que, provavelmente, será o nome desse arquivo, por exemplo, no caso de digitarmos `echo most`, isso será completado com `ra_idade` e ficaremos com `mostra_idade`. Muitos comandos executamos com o nome de arquivos logo em seguida, então, ele já dá um *autocomplete*. Perceba que o `bash` com *autocomplete* é razoavelmente inteligente.

Ele sabe que se é a primeira palavra ele precisa completar com o nome do comando ou com o nome de um diretório que vai encontrar um comando mais adiante. Do segundo argumento em diante ele sabe que pode ser o nome de um arquivo, mas, em geral, não vai ser o nome de um comando. Ele só mostrará a palavra `mostra_idade`.

Se quiséssemos realmente executar, poderíamos digitar `bash`, um espaço, `mo` e "Tab" para completar e aí, teremos o `mostra_idade` e executaremos isso:

```
> bash mostra_idade
voce tem anos.
```

Então, aqui vimos o uso do "Tab" e também dos dois "Tab" que mostra as opções. Na primeira palavra ele dá um *autocomplete* para comandos, diretórios, para poder encontrar o comando que estamos buscando para executar. Na segunda palavra ele dá um *autocomplete* para arquivos e diretórios a partir do diretório atual.

E se digitarmos o `bash` e `/`? Ele reconhece que é um diretório raiz. E se dermos dois "Tabs" ele nos mostra a raiz:

```
> bash
bin/        home/           media/      run/        usr/
boot/       initrd.img      mnt/        sbin/       var/
cdrom/      lib/            opt/        srv/        vmlinuz
dev/        lib64/          proc/       sys/
etc/        lost+found/     root/       tmp/
```

Se deixarmos uma barra estamos dando a dica para o *bash* que pode acontecer um *autocomplete* em diretório pois começamos com uma `/`. Ele sabe que a dica é essa.

Podemos seguir com isso, estender bastante esse tema. Podemos fazer *autocomplete* em vários comandos e programas específicos. Por exemplo, no `git` que instalamos podemos fazer *autocomplete* em seu segundo argumento. É possível digitar `git cl` e se dermos dois "Tab" teremos as seguintes opções para completar:

```
> git cl
clean clone
```

Temos as opções `clean` e `clone` como argumentos para completar esse `git`. O `git` foi configurado para trabalhar em conjunto com o `bash`, para que o `autocomplete` seja super inteligente. Assim, ele já sabe quais são os parâmetro possíveis ou os modos possíveis de execução de um comando do `git`.

Os programas que críamos podem se comunicar com o `shell` para dizer qual é o *autocomplete* que fornecemos.

Bom, esse é o *autocomplete*.

Mas temos ainda pequenos detalhes para analisarmos.

Um primeiro exemplo, é como executamos dois comandos em uma mesma linha. Por exemplo, temos o `ls`, que nos mostra os diretórios que temos:

```
> ls
Desktop     examples.desktop        mostra_idade        Pictures        Videos
Documents       help        mostra_idade~       Public      zip
Dowloads        loja        Music       Templates
```

Mas, e se quisermos executar o `ls` e também quisermos mostrar uma mensagem? Digitaremos o `ls`, espaço, `;` que serve para falar que o comando deve ser executado e mais um espaço e `echo Mostrei os diretorios e arquivos`. Vamos e executar isso e ver o que temos:

```
> ls ; echo Mostrei os diretorios e arquivos
Desktop         examples.desktop        mostra_idade        Pictures        Videos
Documents       help                    mostra_idade~       Public          zip
Dowloads        loja                    Music               Templates
Mostrei os diretorios e arquivos
```

Ele fez o `ls` e depois que terminou, mostrou a mensagem `Mostrei os diretorios e arquivos`. O `;` pode ser usado para separar dois comandos em uma mesma linha. Repare que usar `;` no *bash*, no *shell* em geral, na linha de comando, parece bem feioso de usar na prática. No *script* pode ser até interessante seu uso, uma vez que por algum motivo podemos querer utilizar dois comandos em uma mesma linha e podemos utilizar o `;`.

Repare que ele poderia ser utilizado sem digitarmos os espaços, `ls;echo Terminei`:

```
> ls;echo Terminei
Desktop     examples.desktop        mostra_idade        Pictures        Videos
Documents       help        mostra_idade~       Public      zip
Dowloads        loja        Music       Templates
Terminei
```

Mas, ele utilizado dessa maneira acaba confundindo um pouco. Ele parece deixar mais difícil de compreender o que está sendo executado. Entretanto, ele pode ser utilizado para separar dois comandos em uma mesma linha. É como se ele dissesse: "primeiro faça isso e depois faça esse outro".

Por exemplo, se tentarmos mostrar a seguinte mensagem: `echo Sou dono de um computador; tambem sou dono de um chocolate` e dermos um "Enter", vamos observar o que acontece:

```
> echo Sou dono de um computador; tambem sou dono de um chocolate
Sou dono de um computador
tambem: command nout found
```

Ele ficou confuso com o `;`. Não mostrou a mensagem como se fossem duas partes. Ele reage dessa maneira por que o `;` não é algo que ele entende que nós queiramos que seja impresso na mensagem. O nosso *bash*, o nosso *shell* interpreta o `;` como a indicação de que vai começar um novo comando e que esse comando seria `tambem` e resposta que nos dá é que esse comando não é passível de ser encontrado, `command not found`.

Assim, o `;` separa comandos, e se usarmos ele solto em um texto achando que nós estamos dando um `echo` não vai funcionar da maneira que queríamos.

Temos que tomar cuidado quando utilizamos vários argumentos. O `;` talvez não funcione da maneira como gostaríamos. O `;` é até algo especial. Aliás, temos vários caracteres que são especiais.

Por exemplo, vamos dar um `ls` e observar o que ele nos mostra. Observe que dentre os arquivos temos um `mostra_idade`:

```
> ls
Desktop     examples.desktop        mostra_idade        Pictures        Videos
Documents       help        mostra_idade~       Public      zip
Dowloads        loja        Music       Templates
```

Repare que além do `mostra_idade` também temos um arquivo chamado `mostra_idade~` que se diferencia por ter sido criado pelo `gedit` de Backup. Esse arquivo também se distingue pois contêm o símbolo til, `~`, que o diferencia.

O `gedit` criou esse arquivo `mostra_idade~` de Backup. Podemos digitar o seguinte, `echo Gosto de mo*` e teremos o seguinte:

```
> echo Gosto de mo*
Gosto de mostra_idade mostra_idade~
```

O uso do asterisco em `mo` expandiu o uso para os nomes dos arquivos. E a mesma coisa acontece se usarmos `ls mo*`, ele vai pegar tudo o que começa com `mo` e colocar no comando. É como se tivéssemos digitado `ls mostra_idade mostra_idade~`, isto é, `ls mo*` vai ser equivalente a `ls mostra_idade mostra_idade~`.

O nosso *shell* transforma esse comando de cima, o `ls mo*` no comando de baixo, o `ls mostra_idade mostra_idade~` e, então, ele é executado. É, como o cifrão das variáveis, o `$`, primeiro é interpretado e depois o programa é executado. O asterisco é primeiro interpretado e depois os comandos são executados.

Observe a equivalência de digitarmos `ls mo*` e `ls mostra_idade mostra_idade~`:

```
> ls mo*
mostra_idade mostra_idade~
ls mostra_idade mostra_idade~
mostra_idade mostra_idade~
```

Vamos observar algo, imagine que tivéssemos o seguinte:

```
> echo Primeiro; Segundo
Primeiro,
Segundo: command not found
```

Primeiro, o `;` é interpretado e depois esses comandos são executados. Mas, esses comandos seriam o `echo` e o `Segundo`, onde este último não existe, `Segundo: command not found`

Repare que em diversos momentos o que queremos fazer é fugir dessa situação em que os caracteres especiais, como o `*`, o `;` e diversos outros que veremos mais adiante. Esses caracteres especiais serão interpretados pelo nosso *shell* e isso pode dar em um erro.

O parâmetro que queremos passar, por exemplo, ao usar o `Gosto de mo*` é que estamos tentando dizer ao *shell* para não interpretar o `*`, o `;` ou, qualquer outro carácter especial.

Temos diversas maneiras de fazer isso, uma primeira delas é colocando tudo entre aspas simples.

Por exemplo, `echo` Primeiro; Segundo `` `. O que estamos falando para o *bash* é que o parâmetro começa na aspas e terminas nas aspas, então, temos que nosso parâmetro é `` Primeiro;Segundo`. teremos o seguinte:

```
> echo `Primeiro; Segundo`
Primeiro; Segundo
```

Temos impresso o parâmetro `Primeiro; Segundo`, assim como queríamos e nosso `;` não aparece em nossa mensagem.