# Readline Ninja Skills

**Jack Rosenthal**

2016-03-07
2018-03-08

Mines Linux Users Group
`https://lug.mines.edu`

## Readline

- A library for interactive line editing that your shell probably uses.
- Responsible for things like tab completion, history expansion, and all of those useful keystrokes
- Readline saves you keystrokes.
- Some readline things can make you look like a total ninja.
- Some readline things make you feel like a total ninja.

## Readline

- A library for interactive line editing that your shell probably uses.
- Responsible for things like tab completion, history expansion, and all of those useful keystrokes
- Readline saves you keystrokes.
- Some readline things can make you look like a total ninja.
- Some readline things make you feel like a total ninja.

## Readline

- A library for interactive line editing that your shell probably uses.
- Responsible for things like tab completion, history expansion, and all of those useful keystrokes
- Readline saves you keystrokes.
- Some readline things can make you look like a total ninja.
- Some readline things make you feel like a total ninja.

## Readline

- A library for interactive line editing that your shell probably uses.
- Responsible for things like tab completion, history expansion, and all of those useful keystrokes
- Readline saves you keystrokes.
- Some readline things can make you look like a total ninja.
- Some readline things make you feel like a total ninja.

## Readline

- A library for interactive line editing that your shell probably uses.
- Responsible for things like tab completion, history expansion, and all of those useful keystrokes
- Readline saves you keystrokes.
- Some readline things can make you look like a total ninja.
- Some readline things make you feel like a total ninja.

## Using Readline & History

## History

Readline can track your history, most shells let you use the `history` builtin to view your history.

You can navigate your history using the up and down keys.

## Tab completion

Most of us already know what this and would die without it.

## Event Designators

- `!` - begin history expansion
- `!!` - refer to the last command
- `!`$n$ - refer to the $n$-th command in history
- `!-`$n$ - refer to the current command minus $n$
- `!#` - refer to the current command you are typing
- `!`$search$ - refer to the last commmand that starts with $search$
  `!?`$search$`?` - refer to the last command with $search$ anywhere in the command

Examples:

- `sudo !!` - run the last command with sudo in front
- `!grep` - run the last command you typed beginning with grep

## Event Designators

- `!` - begin history expansion
- `!!` - refer to the last command
- `!`$n$ - refer to the $n$-th command in history
- `!-`$n$ - refer to the current command minus $n$
- `!#` - refer to the current command you are typing
- `!`$search$ - refer to the last commmand that starts with $search$
- `!?`$search$`?` - refer to the last command with $search$ anywhere in the command

Examples:

- `sudo !!` - run the last command with sudo in front
- `!grep` - run the last command you typed beginning with grep

## Event Designators

- `!` - begin history expansion
- `!!` - refer to the last command
- `!`$n$ - refer to the $n$-th command in history
- `!-`$n$ - refer to the current command minus $n$
- `!#` - refer to the current command you are typing
- `!`$search$ - refer to the last commmand that starts with $search$
- `!?`$search$`?` - refer to the last command with $search$ anywhere in the command

Examples:

- `sudo !!` - run the last command with sudo in front
- `!grep` - run the last command you typed beginning with grep

## Event Designators

- `!` - begin history expansion
- `!!` - refer to the last command
- `!`$n$ - refer to the $n$-th command in history
- `!-`$n$ - refer to the current command minus $n$
- `!#` - refer to the current command you are typing
- `!`$search$ - refer to the last commmand that starts with $search$

  `!?`$search$`?` - refer to the last command with $search$ anywhere in the command

Examples:

- `sudo !!` - run the last command with sudo in front
- `!grep` - run the last command you typed beginning with grep

## Event Designators

- ! - begin history expansion
- !! - refer to the last command
- !$n$ - refer to the $n$-th command in history
- !-$n$ - refer to the current command minus $n$
- !# - refer to the current command you are typing
- !*search* - refer to the last commmand that starts with *search*
- !?*search*? - refer to the last command with *search* anywhere in the command

Examples:

- sudo !! - run the last command with sudo in front
- !grep - run the last command you typed beginning with grep

## Event Designators

- `!` - begin history expansion
- `!!` - refer to the last command
- `!`$n$ - refer to the $n$-th command in history
- `!-`$n$ - refer to the current command minus $n$
- `!#` - refer to the current command you are typing
- `!`$search$ - refer to the last commmand that starts with $search$
- `!?`$search$`?` - refer to the last command with $search$ anywhere in the command

Examples:

- `sudo !!` - run the last command with sudo in front
- `!grep` - run the last command you typed beginning with grep

## Event Designators

- `!` - begin history expansion
- `!!` - refer to the last command
- `!`$n$ - refer to the $n$-th command in history
- `!-`$n$ - refer to the current command minus $n$
- `!#` - refer to the current command you are typing
- `!`$search$ - refer to the last commmand that starts with $search$

`!?`$search$`?` - refer to the last command with $search$ anywhere in the command

Examples:

- `sudo !!` - run the last command with sudo in front
- `!grep` - run the last command you typed beginning with grep

## Event Designators

- `!` - begin history expansion
- `!!` - refer to the last command
- `!`$n$ - refer to the $n$-th command in history
- `!-`$n$ - refer to the current command minus $n$
- `!#` - refer to the current command you are typing
- `!`$search$ - refer to the last commmand that starts with $search$

  `!?`$search$`?` - refer to the last command with $search$ anywhere in the command

Examples:

- `sudo !!` - run the last command with sudo in front
- `!grep` - run the last command you typed beginning with grep

## Event Designators

- `!` - begin history expansion
- `!!` - refer to the last command
- `!`$n$ - refer to the $n$-th command in history
- `!-`$n$ - refer to the current command minus $n$
- `!#` - refer to the current command you are typing
- `!`$search$ - refer to the last commmand that starts with $search$

  `!?`$search$`?` - refer to the last command with $search$ anywhere in the command

Examples:

- `sudo !!` - run the last command with sudo in front
- `!grep` - run the last command you typed beginning with `grep`

## Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (`:`) and then a word designator.

- `:`*n* - select argument *n* (zero indexed)
- `:`*n*`-`*m* - select arguments *n* through *m*
- `:$` - select the last argument (think of a regex)
- `:*` - select all arguments, omitting the command name (equivalent to `:1-$`)
- `:%` - select the argument that matches `?`*word*`?`

Examples:

- `cd !!:1` - cd to the first argument of the last command
- `vim !-2:$` - edit the file that is the last argument of two commands ago

## Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (`:`) and then a word designator.

- `:`$n$ - select argument $n$ (zero indexed)
- `:`$n$`-`$m$ - select arguments $n$ through $m$
- `:$` - select the last argument (think of a regex)
- `:*` - select all arguments, omitting the command name (equivalent to `:1-$`)
- `:%` - select the argument that matches `?`*word*`?`

Examples:

- `cd !!:1` - cd to the first argument of the last command
- `vim !-2:$` - edit the file that is the last argument of two commands ago

## Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (`:`) and then a word designator.

- `:`$n$ - select argument $n$ (zero indexed)
- `:`$n$`-`$m$ - select arguments $n$ through $m$
- `:$` - select the last argument (think of a regex)
- `:*` - select all arguments, omitting the command name (equivalent to `:1-$`)
- `:%` - select the argument that matches `?`$search$`?`

Examples:

- `cd !!:1` - cd to the first argument of the last command
- `vim !-2:$` - edit the file that is the last argument of two commands ago

## Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (`:`) and then a word designator.

- `:`$n$ - select argument $n$ (zero indexed)
- `:`$n$`-`$m$ - select arguments $n$ through $m$
- `:$` - select the last argument (think of a regex)
- `:*` - select all arguments, omitting the command name (equivalent to `:1-$`)
- `:%` - select the argument that matches `?`$search$`?`

Examples:

- `cd !!:1` - cd to the first argument of the last command
- `vim !-2:$` - edit the file that is the last argument of two commands ago

## Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (`:`) and then a word designator.

- `:`$n$ - select argument $n$ (zero indexed)
- `:`$n$`-`$m$ - select arguments $n$ through $m$
- `:$` - select the last argument (think of a regex)
- `:*` - select all arguments, omitting the command name (equivalent to `:1-$`)
- `:%` - select the argument that matches `?`$search$`?`

Examples:

- `cd !!:1` - cd to the first argument of the last command
- `vim !-2:$` - edit the file that is the last argument of two commands ago

## Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (`:`) and then a word designator.

- `:`$n$ - select argument $n$ (zero indexed)
- `:`$n$`-`$m$ - select arguments $n$ through $m$
- `:$` - select the last argument (think of a regex)
- `:*` - select all arguments, omitting the command name (equivalent to `:1-$`)
- `:%` - select the argument that matches `?`$search$`?`

Examples:

- `cd !!:1` - cd to the first argument of the last command.
- `vim !-2:$` - edit the file that is the last argument of two commands ago

## Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (`:`) and then a word designator.

- `:`$n$ - select argument $n$ (zero indexed)
- `:`$n$`-`$m$ - select arguments $n$ through $m$
- `:$` - select the last argument (think of a regex)
- `:*` - select all arguments, omitting the command name (equivalent to `:1-$`)
- `:%` - select the argument that matches `?`$search$`?`

Examples:

- `cd !!:1` - cd to the first argument of the last command.
- `vim !-2:$` - edit the file that is the last argument of two commands ago

## Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (`:`) and then a word designator.

- `:`$n$ - select argument $n$ (zero indexed)
- `:`$n$`-`$m$ - select arguments $n$ through $m$
- `:$` - select the last argument (think of a regex)
- `:*` - select all arguments, omitting the command name (equivalent to `:1-$`)
- `:%` - select the argument that matches `?`$search$`?`

Examples:

- `cd !!:1` - cd to the first argument of the last command.
- `vim !-2:$` - edit the file that is the last argument of two commands ago

## Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (`:`) and then a word designator.

- `:`$n$ - select argument $n$ (zero indexed)
- `:`$n$`-`$m$ - select arguments $n$ through $m$
- `:$` - select the last argument (think of a regex)
- `:*` - select all arguments, omitting the command name (equivalent to `:1-$`)

  `:%` - select the argument that matches `?`$search$?

Examples:

- `cd !!:1` - cd to the first argument of the last command.
- `vim !-2:$` - edit the file that is the last argument of two commands ago

## Modifiers

Modifiers let you chop up the history expansion in ways that you like. You can chain any amount of modifiers that you would like onto your expansion.

- :r - Chop off the extension of a filename
- :h - Remove the filename component, leaving only the directory (think of head)
- :t - Remove the directory component, leaving only the filename (think of tail)
- :q - Quote each of the arguments
- :s/*search*/*replace*/ - sed style substitution
- :gs/*search*/*replace*/ - sed style substitution, globally
- :p - print the history expansion, don't execute quite yet

## Modifiers

Modifiers let you chop up the history expansion in ways that you like. You can chain any amount of modifiers that you would like onto your expansion.

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/`*search*`/`*replace*`/` - sed style substitution
- `:gs/`*search*`/`*replace*`/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

## Modifiers

Modifiers let you chop up the history expansion in ways that you like. You can chain any amount of modifiers that you would like onto your expansion.

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/`*search*`/`*replace*`/` - sed style substitution
- `:gs/`*search*`/`*replace*`/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

## Modifiers

Modifiers let you chop up the history expansion in ways that you like. You can chain any amount of modifiers that you would like onto your expansion.

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s`/*search*/*replace*/ - sed style substitution
- `:gs`/*search*/*replace*/ - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

## Modifiers

Modifiers let you chop up the history expansion in ways that you like. You can chain any amount of modifiers that you would like onto your expansion.

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s`/*search*/*replace*/ - sed style substitution
- `:gs`/*search*/*replace*/ - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

## Modifiers

Modifiers let you chop up the history expansion in ways that you like. You can chain any amount of modifiers that you would like onto your expansion.

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s`/$search$/$replace$/ - sed style substitution
- `:gs`/$search$/$replace$/ - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

## Modifiers

Modifiers let you chop up the history expansion in ways that you like. You can chain any amount of modifiers that you would like onto your expansion.

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s`/*search*/*replace*/ - sed style substitution
- `:gs`/*search*/*replace*/ - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

## Modifiers

Modifiers let you chop up the history expansion in ways that you like. You can chain any amount of modifiers that you would like onto your expansion.

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s`/*search*/*replace*/ - sed style substitution
- `:gs`/*search*/*replace*/ - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

## Modifiers

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/`*search*`/`*replace*`/` - sed style substitution
- `:gs/`*search*`/`*replace*`/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

Examples:

- `mv important.png !#:1:r.gif` - rename `important.png` to `important.gif`
- `touch mydir/file.txt`
- `cd !$:h`

## Modifiers

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/`*search*`/`*replace*`/` - sed style substitution
- `:gs/`*search*`/`*replace*`/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

Examples:

- `mv important.png !#:1:r.gif` - rename `important.png` to `important.gif`
- `touch mydir/file.txt`
- `cd !$:h`

## Modifiers

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/`*search*`/`*replace*`/` - sed style substitution
- `:gs/`*search*`/`*replace*`/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

Examples:

- `mv important.png !#:1:r.gif` - rename `important.png` to `important.gif`
- `touch mydir/file.txt`
- `cd !$:h`

## Modifiers

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/`*search*`/`*replace*`/` - sed style substitution
- `:gs/`*search*`/`*replace*`/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

Examples:

- `mv important.png !#:1:r.gif` - rename `important.png` to `important.gif`
- `touch mydir/file.txt`
- `cd !$:h`

## Abbreviations Allowed

- `!!:`… can be shortened to `!:`…
- The `:` can be removed from word designators where it is unambiguous. So `!$` and `!*` are allowed.
- The trailing / in a substitution can be omitted if it is unambigous that the substitution has ended.
- The trailing ? in a !?*search*? can be ommitted for the same reason.
- Any delimiter can be used in a substitution, so `!:sx`*find*x*replace*x is legal.

## Abbreviations Allowed

- `!!:`... can be shortened to `!:`...
- The `:` can be removed from word designators where it is unambiguous. So `!$` and `!*` are allowed.
- The trailing / in a substitution can be omitted if it is unambigous that the substitution has ended.
- The trailing ? in a `!?`*search*? can be ommitted for the same reason.
- Any delimiter can be used in a substitution, so `!:sx`*find*`x`*replace*`x` is legal.

## Abbreviations Allowed

- `!!:`… can be shortened to `!:`…
- The `:` can be removed from word designators where it is unambiguous. So `!$` and `!*` are allowed.
- The trailing / in a substitution can be omitted if it is unambiguous that the substitution has ended.
- The trailing ? in a `!?`*search*? can be ommitted for the same reason.
- Any delimiter can be used in a substitution, so `!:sx`*find*x*replace*x is legal.

## Abbreviations Allowed

- `!!:…` can be shortened to `!:…`
- The `:` can be removed from word designators where it is unambiguous. So `!$` and `!*` are allowed.
- The trailing / in a substitution can be omitted if it is unambigous that the substitution has ended.
- The trailing ? in a `!?`*search*? can be ommitted for the same reason.
- Any delimiter can be used in a substitution, so `!:s`x*find*x*replace*x is legal.

## Abbreviations Allowed

- `!!:`… can be shortened to `!:`…
- The `:` can be removed from word designators where it is unambiguous. So `!$` and `!*` are allowed.
- The trailing / in a substitution can be omitted if it is unambigous that the substitution has ended.
- The trailing ? in a `!?`*search*? can be ommitted for the same reason.
- Any delimiter can be used in a substitution, so `!:sx`*find*`x`*replace*`x` is legal.

## Editing Modes

Readline provides editing modes similar to `vi` and `emacs`. Learn one and learn to love it. Most shells and programs have `emacs` as the default.

## History Incremental Search

`<C-r>` (`vi: <Esc>/`) brings you to an search of your history. `<C-s>` will reverse the direction of your search (You may need to `stty -ixon`).

## Readline Programming in C/C++

## C/C++ Readline Library

```
#include <stdio.h>
#include <readline/readline.h>
#include <readline/history.h>

char * readline(const char *prompt);
```

Allocates memory to read a line, reads it from standard input (displaying `prompt` as the prompt line). Returns the line you read. You really should `free` the memory it allocated.

# Using History Features

```c
void using_history(void);
```

Must be called before using history features.

```c
int read_history(const char *filename);
int write_history(const char *filename);
```

For reading/writing saved history. Returns non-zero on failure and sets `errno`.

```c
void add_history(const char *line);
```

Add a line to the history.

```c
HIST_ENTRY ** histlst = history_list();
for (int i = 1; *histlst; i++, histlst++)
    printf("%d %s\n", i, (*histlst)->line);
```

List history.

## Using History Features

```c
void using_history(void);
```

Must be called before using history features.

```c
int read_history(const char *filename);
int write_history(const char *filename);
```

For reading/writing saved history. Returns non-zero on failure and sets `errno`.

```c
void add_history(const char *line);
```

Add a line to the history.

```c
HIST_ENTRY ** histlst = history_list();
for (int i = 1; *histlst; i++, histlst++)
    printf("%d %s\n", i, (*histlst)->line);
```

List history.

## Using History Features

```c
void using_history(void);
```

Must be called before using history features.

```c
int read_history(const char *filename);
int write_history(const char *filename);
```

For reading/writing saved history. Returns non-zero on failure and sets `errno`.

```c
void add_history(const char *line);
```

Add a line to the history.

```c
HIST_ENTRY ** histlst = history_list();
for (int i = 1; *histlst; i++, histlst++)
    printf("%d %s\n", i, (*histlst)->line);
```

List history.

## Using History Features

```c
void using_history(void);
```

Must be called before using history features.

```c
int read_history(const char *filename);
int write_history(const char *filename);
```

For reading/writing saved history. Returns non-zero on failure and sets `errno`.

```c
void add_history(const char *line);
```

Add a line to the history.

```c
HIST_ENTRY ** histlst = history_list();
for (int i = 1; *histlst; i++, histlst++)
    printf("%d %s\n", i, (*histlst)->line);
```

List history.

## History Expansion (for free!)

```
int history_expand(char *string, char **output);
```

Expand string, placing the result into output, a pointer to a string. Returns:

|  |  |
|---|---|
| 0 | If no expansions took place |
| 1 | If expansions did take place |
| -1 | If there was an error in expansion |
| 2 | If the line should be displayed, but not executed (`:p`) |

If an error occurred in expansion, then output contains a descriptive error message.

## A Complete Example: 31-line UNIX shell

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <readline/readline.h>
#include <readline/history.h>

int main(void) {
    char *line = NULL, *expn = NULL;
    int status;
    using_history();
    for (;;) {
        free(line), free(expn);
        line = readline("prompt> ");
        if (!line) return 0; /* ^D to exit */
        int expn_result = history_expand(line, &expn);
        if (expn_result) puts(expn);
        add_history(expn);
        if (expn_result == 0 || expn_result == 1) {
            int pid = fork();
            if (pid < 0) return 1;
            if (pid == 0) {
                char ** arg = history_tokenize(expn);
                execvp(*arg, arg);
                return 1;
            }
            waitpid(pid, &status, 0);
        }
    }
    return 0;
}
```

## Readline Programming in Python

## import readline

To use Readline from Python, type `import readline`, and the `input` function will magically become readlineifyed.

```python
import sys
import readline

while True:
    try:
        cmd = input(">>> ")
    except KeyboardInterrupt:
        continue
    except EOFError:
        sys.exit(0)
    print(exec(cmd))
```

## Tab Completion

The `readline` module provides an interface for you to add your own completer:

```
readline.set_completer(function)
```

`function` should be a function which takes two parameters:

| | |
|---|---|
| `text` | The current completion text |
| `state` | `0`, 1, ... |

Then, set your delimiters and completion keys:

```
readline.set_completer_delims(' ')
readline.parse_and_bind("tab: complete")
```

## Tab Completion

The `readline` module provides an interface for you to add your own completer:

```
readline.set_completer(function)
```

`function` should be a function which takes two parameters:

| | |
|---|---|
| `text` | The current completion text |
| `state` | `0, 1, ...` |

Then, set your delimiters and completion keys:

```
readline.set_completer_delims(' ')
readline.parse_and_bind("tab: complete")
```

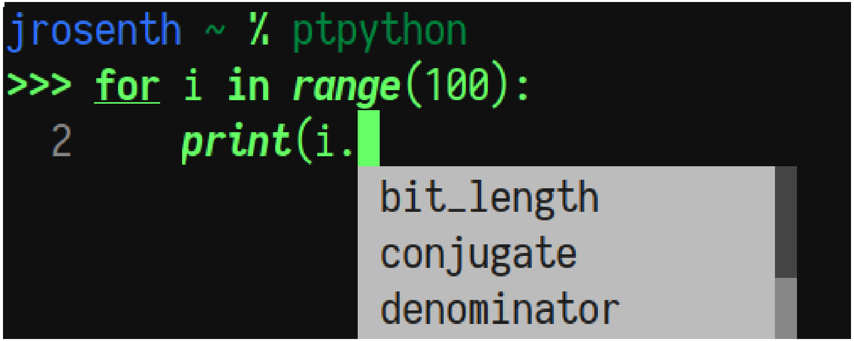
## Custom Completion in the Wild: `iels`

```python
def completer(text, state):
    def gen():
        variables = reduce(set.union, map(dict.keys, els.vars), set())
        for s in '%', '$':
            for v in variables:
                if (s + v).startswith(text):
                    yield s + v
        for op in els.operators:
            if op.startswith(text):
                yield op
        for syntax in 'begin', 'end':
            if syntax.startswith(text):
                yield syntax

    if state == 0:
        completer.it = gen()

    try:
        return next(completer.it)
    except StopIteration:
        return None
```

## Alternatives to Readline for Python

While Readline is a well written piece of software, it feels a little bit out of place in Python, with the bindings reflective of the state-maintaining C code they talk to.

Prompt Toolkit is a pure-Python alternative with fancy features:



```
jrosenth ~ % ptpython
>>> for i in range(100):
  2     print(i.
                 bit_length
                 conjugate
                 denominator
```

## Further Resources

## More Info

1. `man 3 readline`
2. `man 3 history`
3. `pydoc readline`
4. RTFM: Read The *Fine* Manual

## Questions?