

---

# **termite**

***Release 0.0.1***

February 16, 2017



<b>1</b>	<b>Features</b>	<b>3</b>
<b>2</b>	<b>Alternatives</b>	<b>5</b>
<b>3</b>	<b>Why another build tool?</b>	<b>7</b>
<b>4</b>	<b>Requeriments</b>	<b>9</b>
<b>5</b>	<b>Installation</b>	<b>11</b>
5.1	Basic concepts . . . . .	11
5.2	Commands . . . . .	11
5.3	Tasks . . . . .	12
5.4	Calling external python functions . . . . .	13
5.5	Indices and tables . . . . .	14



A tool to automate your work.



---

### Features

---

- Runs tasks from the command line.
- Watches for file changes .
- Serves HTML files and reload them on changes.





---

### Alternatives

---

A list of alternatives to [Termite](#):

- [Grunt](#)
- [Brunch](#)
- [Mimosa](#)
- [LiveReload](#)



---

## Why another build tool?

---

After some frustration with the alternatives, I started to write [Termite](#).



---

## Requeriments

---

Termite needs Python 3.3 or better



---

## Installation

---

```
pip install termite
```

### Contents:

## Basic concepts

Termite uses the [yaml](#) format to define commands and tasks. The wikipedia has a [good description of the format](#).

The main entry point is a yaml file, called *termite.yaml*, which should be in your current working directory.

In termite we have two basic elements, the commands, and the tasks. A command is a list of tasks, and should have a name, which is basically an identifier. Let's see a basic *termite.yaml* file:

```
- command:
  name: dev
  tasks:
    - shell:
      command: echo "Hello world!!"
```

Run this in the command line to see the greeting:

```
termite dev
```

## Commands

Commands define the tasks to be run. An example of a *termite.yaml* file with 2 commands:

```
- command:
  name: hello
  tasks:
    - shell:
      command: echo "Hello world!!"

- command:
  name: bye
  tasks:
    - shell:
      command: echo "Goodbye!!"
```

Pass the name of the command to termite as its first argument. If you don't specify any command name in the command line, **Termite** runs the first command found. In this example, running in the command line:

```
termite hello
```

has the same effect as run just

```
termite
```

## Global tasks

It is possible to create a task globally and use it in several commands, an example:

```
- shell: &some_id
  command: echo "Hello world!!"

- command:
  name: hello
  tasks:
    - shell: *some_id

- command:
  name: bye
  tasks:
    - shell: *some_id

    - shell:
      command: echo "Goodbye!!"
```

## Tasks

There are 3 types of tasks in **Termite**, *shell*, *cp* and *server*

### Shell tasks

Shell tasks accept 3 options, *Command (Mandatory)*, *Cwd (Optional)* and *watch*.

#### Command (Mandatory)

Specifies the command to run. It is also possible to specify a list of commands. In this case, the commands are run sequentially.

#### Cwd (Optional)

The current directory will be changed to *cwd* before the command is executed.

#### Watch (Optional)

List of files to watch for modifications. After any change, the command is executed again. It is possible to use shell-style wildcards (\* or \*\*). It is also possible to specify folders to watch, in this case */some/path/* and */some/path/\*\** have the same effect. If *watch* is omitted, the command is run only once.



## Cp tasks

Copy files is a very common operation, that's the reason we have a task for this operation, although it would be possible to use a command task for copy files. For *cp* tasks there are 3 options, *Source (Mandatory)*, *Dest (Mandatory)* and *watch*.

### Source (Mandatory)

A file, or list of files to copy. Shell-style wildcards are allowed.

### Dest (Mandatory)

Where to copy the file or files. Should be a folder, if it doesn't exist it is created. Be careful, files are overwritten without any warning.

### Watch (Optional)

Specifies if the source files should be monitored. It is a boolean value, by default the value is set to *False*.

## Server task

This task starts an HTTP server. If you are watching any files, your browser is automatically refreshed after every change. Has only one option, *Path (Mandatory)*.

### Path (Mandatory)

Serves files from this directory.

## Calling external python functions

Command line is great, but sometimes it is useful to write python code to do some tasks. *Termite* provides a command line utility, called *tcli*, to help you with that.

First, write a python file with your utilities, call this file *termite\_cli.py*, and put this file in the same directory where your *termite.yaml* resides.

A simple *termite\_cli.py* file:

```
def hello(args):
    print('Hello, your arguments are: ', args)
```

Now, from the command line, run this:

```
tcli hello -x 5
```

*Termite* is going to call the function *hello* in the file *termite\_cli.py*. All the arguments after the function name, are saved in a python list and passed to the function. In our case the value of *args* is *['-x', '5']*

Call the *hello* function from a *Termite* file with this task:

```
- shell:
  command: tcli hello the arguments
```

Lets write a more complicated *termite\_cli.py* file:

```
import os
from docopt import docopt
from jinja2 import Environment, FileSystemLoader

def render(args):
    usage = '''Usage: render (--input IN) (--output OUT) [<vars>...]'''

    arguments = docopt(usage, argv=args)
    variables = dict([var.split('=') for var in arguments['<vars>']])

    env = Environment(loader=FileSystemLoader(os.getcwd()))
    template = env.get_template(arguments['IN'])
    with open(arguments['OUT'], 'w') as out:
        out.write(template.render(**variables))
```

And the associated **Termite** task:

```
- shell:
  command: tcli render --input app/index.html --output build/index.html dev=true
```

In this example we are rendering a HTML template using **Jinja**. To parse the command line arguments we are using **docopt**.

## Indices and tables

- [genindex](#)
- [modindex](#)
- [search](#)