TinyMk Documentation

Release 0.4

Ryan Gonzalez

Aug 16, 2017

Contents

Contents:

CHAPTER 1

Introduction/FAQ

Why another build tool?

There's no real reason; I just wanted to experiment. After using build systems like Shake (very powerful but complex) and Cake (simple and elegant but very rebuild-the-wheel type), I wanted a mix. I like Haskell, but I don't like it enough to work with it that much. (**FUTURE EDIT:** I now like Haskell a lot more, but it's large size is a big problem when trying to deploy.) TinyMk is written in Python, which is pre-installed on most Unix-like platforms, and is very portable. One thing I especially liked about Cake was the use of creative prefixes (task *build* calls *build:objects* and *build:library*). TinyMk implements this using categories.

Is TinyMk better than *x*?

No. Every build system has different goals/concepts. It's useless to ask if *x* is better than *y* when they have completely different end goals.

Of course, when comparing *x* to CMake/autotools, *x* always wins.

Why is the documentation so out-of-date?

Work-in-progress. I hate writing API docs.

I have a question about using TinyMk. Where should I go?

Use the mailing list: tinymk@googlegroups.com.

CHAPTER 2

Tutorial

TinyMk may be small, but it's still quite powerful.

The basics

Here's a basic example:

```
from tinymk import *

@task()
def test():
    print('Hello, TinyMk!')

main()
```

This simple code defines a task named *test*. Save this into a file named *build.py* and run it:

```
$ python build.py
invalid number of args
usage: build.py [-h|--help] [--task-help] <task> [<args>]
$
```

To list the tasks, use *?* as the task name:

```
$ python build.py ?
Tasks:

test
$
```

Let's run our task:

```
$ python build.py test
Running task test...
Hello, TinkMk!
$
```

Running commands

Often, in a build script, you don't want to just print stuff. You want to be able to run programs. Look at this example:

```
from tinkmk import *

@task()
def test():
    run('cp x.in x.out')

main()
```

Now create a file named *x.in*:

```
$ echo 'Hi!' > x.in
$
```

Next, run the build script and check our new file *x.out*:

```
$ python build.py test
Running task test...
cp x.in x.out
$ cat x.out
Hi!
$
```

Dependencies

Let's say we have a big project, consisting of millions of files. It's likely that you don't want to rebuild everything when you modify one file. Build systems like make let you mention a command's dependencies:

```
x.out : x.in
    cp x.in x.out
```

TinyMk can do this, too, albeit slightly differently. You manually check if you need to update the files using *need_to_update*:

```
 from tinymk import *

@task()
def test():
    if need_to_update('x.out', 'x.in'):
        run('cp x.in x.out')

main()
```

Typing all of that is a pain, though. That's what the *run_d* function is for:

```
from tinymk import *

@task()
def test():
    run_d('x.out', 'x.in', 'cp x.in x.out')

main()
```

That's easier, isn't it?

Categories

In a large project, you might want to apply some method of organization. TinyMk lets you group tasks into *categories*. Here's an example:

```
from tinymk import *

@task('a:')
def b():
    print('Inside task a:b')

main()
```

Now you can use it like this:

```
$ python build.py a:b
Running task a:b...
Inside task a:b
$
```

Parallel execution

Sometimes you can run different tasks at the same time. For instance:

```
from tinymk import *

@task()
def build_object1():
    run_d('a.o', 'a.c', 'gcc -c a.c -o a.o')

@task()
def build_object2():
    run_d('b.o', 'b.c', 'gcc -c b.c -o b.o')

@task()
def build():
    if need_to_update('app', ['a.o', 'b.o']):
        qinvoke('build_object1')
        qinvoke('build_object2')
        run('gcc a.o b.o -o app')

main()
```

Notice the use of *qinvoke*. It's like invoke, but it doesn't print the name of the currently running task.

Now, *a.o* and *b.o* don't directly depend on each other. We can technically build those two at the same time. Look at this slightly modified code:

```
from tinymk import *

@task()
def build_object1():
    run_d('a.o', 'a.c', 'gcc -c a.c -o a.o')

@task()
def build_object2():
    run_d('b.o', 'b.c', 'gcc -c b.c -o b.o')

@task()
def build():
    if need_to_update('app', ['a.o', 'b.o']):
        p1 = pqinvoke('build_object1')
        p2 = pqinvoke('build_object2')
        p1.join()
        p2.join()
        run('gcc a.o b.o -o app')

main()
```

This time, we're using *pqinvoke*. *pqinvoke* is just like *qinvoke*, except that it return an object of type *multiprocessing.Process* (see the Python multiprocessing module). The next line does the same thing. The neat thing is that *pqinvoke* doesn't wait for the task to finish. It simply starts the task in a seperate process. That way, you can run multiple tasks at once.

However, there is a major issue: how do we know when *p1* and *p2* are done so we can finish building? Well, the *join* method simply pauses the current task until it's own task finishes running.

Also note that, just like *qinvoke* has it's counterpart *pqinvoke*, *invoke* has its own multiprocessing counterpart: *pinvoke*.

One more thing: you need to be careful when printing text to the screen when multiple tasks are running at once, or else their output will get all jumbled together. To fix the issue, simply enclose the code with a *with lock:* block:

```
with lock:
    print('Hello!')
# continue doing other stuff...
```

Pattern tasks

Well, what if you need to make a copy of every file in the directory? TinyMk has a feature for this: pattern tasks. A *pattern task* is the TinyMk equivalent to GNU make's pattern rules:

```
from tinymk import *

@ptask('%.in', '%.out', glob.glob('*.in'))
def copy_files(outs, dep):
    run_d(outs, dep, 'cp %s %s' % (out[0], dep))

main()
```

What the above code does is this:

For every file in the list returned by *glob.glob*:

- Match the file against the pattern *%.in*. Think about it like a regex: *(.+?).in*.
- Take the next that's in the place of the percent sign and replace the percent in *%.out* with it. For example, if *glob.glob* returned *['abc.in']*, then the pattern *%.in* matching against it would result in *abc*. Then, the percent sign in *%.out* is replaced with *abc* to result in *abc.out*.
- Create a task with those files.

outs is a list, which is why we index the 1st element.

Invoking categories

Pattern rules are great, but it's tricky to call them. The solution: put them all in a category and use *cinvoke*:

```
...

add_category('copy_stuff')

@ptask('%.in', '%.out', glob.glob('*.in'), 'copy_stuff')
...

@task()
def copy_stuff():
    cinvoke('copy_stuff')
```

cinvoke runs every task inside the category *copy_stuff* (except for copy_stuff itself).

Default tasks

You can have a task that will be run by default if no other task is specified:

```
@task()
def build():
    print("Inside build")

main(default='build')
```

Conclusion

That concludes this breif tutorial on TinyMk. There's much more that hasn't been discussed, however; you'll want to read the *API reference*. In addition, you should read the *command line interface reference*.

CHAPTER 3

Command-line reference

Usage:

```
build_script [-h|--help] [--task-help] <task> [<args>]
```

-h, --help
Show the help screen.

--task-help
Print information on specifying task names. See *Specifying task names*.

task
The task to call. See *Specifying task names*.

args
Arguments passed to the task.

Specifying task names

Tasks are organized into groups called categories. For example, this task name:

```
a:b:c
```

is referring to the task *c* inside the category *b* inside the category *a*.

If you do this:

```
a:b:?
```

the tasks belonging to the category *b* inside the category *a* will be printed.

If you do this:

```
a:b:c?
```

it will print information about the task *c* inside *b* inside *a*.

CHAPTER 4

API reference

`tinymk.`**`lock`**
An instance of *multiprocessing.Lock*. Use this when dealing with *stdout* and *stderr*.

`tinymk.`**`DBManager`**
The database manager. This is exported so that you can change the database path if desired:

```
from tinymk import *
DBManager.path = 'whatever-other-name.db'
```

The default path is `.tinymk.db`.

`tinymk.`**`file_digest`**(*path*, *hash=<unspecified>*)
Return the digest of the given path. *hash* is the hashing algorithm to use (you can subsitite in any hash in *hashlib*, e.g. `file_digest(..., hash=hashlib.sha224)`). The default hashing algorithms used are blake2b on 64-bit Python 3.6, blake2s on 32-bit Python 3.6, and SHA-256 on all other Python versions and architectures.

`tinymk.`**`add_category`**(*name*)
Add a new category. You can create multiple categories at once by separating the name with colons(*:*):

```
add_category('a') # add a category named a
add_category('a:b:c') # add a category named c inside a new category in b inside a
```

Deprecated since version 0.4: *`task()`* will automatically create any categories in its path.

`tinymk.`**`task`**(*tname=None*)
A decorator to create a new task with the name *tname*.

Parameters **`tname`** – If *None*, the task will carry the name of the function. In addition, if *tname* ends with a colon, *tname* will be used as the category, and the function's name will be the task name. For example:

```
@task('a:b:') # task name will be a:b:c
def c(): pass

@task('a:b:d') # task name will be a:b:d
```

```
def abc(): pass

@task() # task name will be xyz
def xyz(): pass
```

tinymk.**ptask**(*pattern*, *outs*, *deps*, *category=None*)

A decorator to create a set of pattern tasks. Pattern tasks are the TinyMk equivalent of GNU Make's *pattern rules*. Here's an example:

```
@ptask('%.in', '%.out', glob.glob('*.in'))
def copy_files(outs, dep):
    run_d(outs, dep, 'cp %s %s' % (dep, outs[0]))
```

That's roughly equivalent to this GNU make rule:

```
%.out : %.in
    cp $< $@
```

Parameters

- **pattern** – The pattern that *deps* will be matched against.
- **outs** – The output file patterns.
- **deps** – The input files.
- **category** – The category to place the tasks in.

tinymk.**need_to_update**(*outs*, *deps*)

Returns *True* if the oldest file in *outs* is newer than the newest file in *deps*. If either *outs* or *deps* is a string, it will be converted to a list using *shlex.split*.

tinymk.**digest_update**(*outs*, *deps*)

Returns *True* if any of the files in *deps* have been modified since the last time the function was called. The SHA1 hashes are stored in an SQLite3 database.

Parameters

- **outs** – Ignored. Only here so it can be used with *run_d()*.
- **deps** – The dependencies.

tinymk.**invoke**(*name*, **args*, ***kw*)

Calls the task named *name*.

Parameters

- **name** – The task to call.
- ***args** – The positional arguments passed to the task.
- ****kwargs** – The keyword arguments passed to the task.

tinymk.**qinvoke**(*name*, **args*, ***kw*)

The same thing as *invoke()*, but doesn't print the task that is executing.

tinymk.**pinvoke**(**args*, ***kw*)

The same thing as *invoke*, but, instead of running the task, launches it in a seperate process and returns a *multiprocessing.Process* object. See *invoke()*.

tinymk.**pqinvoke**(**args*, ***kw*)

The same thing as *pinvoke*, but doesn't print the task that is executing.

tinymk.**cinvoke**(*category*, *invoker=invoke*)
Call *invoker* for every task contained within *category*. Note that, if the category itself is a task, it will not be called.

tinymk.**run**(*cmd*, *write=True*, *shell=False*, *get_output=False*)
Run *cmd*.

Parameters

- **cmd** – The command to run. If it is a string and *shell* is False, it will first be converted to a list.
- **write** – If *True*, the command will be printed to the screen before it's run.
- **shell** – If *True*, the command will be run in the shell.
- **get_output** – If *True*, a tuple consisting of *(stdout, stderr)* containing the command's output will be returned.

tinymk.**run_d**(*outs*, *deps*, *cmd*, *func=need_to_update*, ***kw*)
Call *run* with *cmd* if *func*, when called with *outs* and *deps*, returns *True*. Doing:

```
run_d('x.out', 'x.in', 'cp x.in x.out', func)
```

Is equivalent to:

```
if func('x.out', 'x.in'):
    run('cp x.in x.out')
```

Parameters

- **outs** – The output files.
- **deps** – The dependencies.
- **cmd** – The command to run. See *run()*.
- ****kw** – Keyword arguments passed to *run*. See *run()*.

tinymk.**main**(*no_warn=False*, *default=None*)
Run the main driver. If *no_warn* is *True*, then no deprecation warnings will be displayed. If *default* is not *None*, it is assumed to be a string holding a task to run if no tasks were given on the command line.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

Symbols

A

C

D

F

I

L

M

N

P

Q

R

T