teapot Documentation

Release 2.6

Julien Kauffmann

December 23, 2014

Contents

This documentation is about **teapot**, a multi-platform tool to ease fetching, organization and building of third-party softwares.

CHAPTER 1

What is teapot ?

teapot is a Python package that comes with *teapot*, a command-line interface tool. *teapot* reads a party file which defines the source, the properties, the environment and the build steps for all the third-party libraries to build.

The idea is to add a simple `Party` (or `.party`) file inside your project source tree that will describe which third-party libraries it depends on and how to build them.

The party file, describes the format of the *party file* and enumerates all the possible options.

CHAPTER 2

Why should I use teapot ?

Because you probably have more interesting things to do than dealing with third-party softwares.

Most of the time, people and companies end up writing their own set of scripts to build their dependencies. It can go from a simple *wget* call that fetches precompiled binaries from some server, to more complex systems that download and build them from source and try to do so as reliably as they can.

Writing a script that downloads a *.tar.gz* file, uncompresses it and builds it is really not difficult. But what if you want to handle dependencies between your third party libraries, or desire to support variant builds ? How do you deal with multiple platforms ? How can you react to changes and automatically rebuild what's necessary ? With **teapot**, you just have to write a simple *party file* once and call the *teapot* command once in a while. You can even integrate it into your usual build system since it automatically deals with dependencies and avoids unecessary rebuilds.

For instance, this *party file* downloads, unpacks and builds the popular `libiconv` on all UNIX platforms:

```
from teapot import *

Attendee('iconv').add_source('http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.14.tar.gz')
Attendee('iconv').add_build('default')
Attendee('iconv').get_build('default').add_command('./configure --prefix={{prefix}}')
Attendee('iconv').get_build('default').add_command('make')
Attendee('iconv').get_build('default').add_command('make install')
```

How simpler can it get ?

CHAPTER 3

So, will *teapot* build my third-party software for me ?

Yes it will, but you will still have to tell him how exactly.

There are just too many different ways of building software for this to be done without human guidance.

However, *teapot* will make this as painless as it can get by automating all the other steps that can be automated.

CHAPTER 4

Why this name ?

No good reason really. I just don't like spending too much time finding catchy names and a *teapot* is a nice object so... why not ? :)

CHAPTER 5

What's next ?

Here are the chapters you should read if you want to get familiar with *teapot*:

5.1 The *party file*

The *party file* is at the heart of **teapot**. It describes the different third-party softwares to build, and how to build them.

5.1.1 Structure

The *party file* is a regular Python file.

Whatever you write in the *party file* is declarative, meaning that you don't tell **teapot** to actually build things, you just tell it what to build, and how. The actual build process will take place later when you call the command-line tool. See the *party file* as a declaration file.

Attendees

attendees are the main element of the *party file*. An *attendee* represents a library/project to build. An *attendee* can have one or several *sources*, and one or several *builds*.

Here is an example that declares two *attendees*:

```
from teapot import *

Attendee('iconv').add_source('http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.14.tar.gz')
Attendee('curl').add_source('http://curl.haxx.se/download/curl-7.32.0.tar.gz')
```

This example, while perfectly valid, is not quite complete: as they are written, those *attendees* would be able to download and unpack the specified archives, but they don't know how to build the software they constitute.

Here is a more complete *party file* with an *attendee* that actually does something:

```
from teapot import *

Attendee('iconv').add_source('http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.14.tar.gz')
Attendee('iconv').add_build('default', environment='system')
Attendee('iconv').get_build('default').add_command('./configure --prefix={{prefix}}')
Attendee('iconv').get_build('default').add_command('make')
Attendee('iconv').get_build('default').add_command('make install')
```

This *party file* defines completely the way to build *libicon, version 1.14*. The archive will be downloaded from the specified URL, it will be extracted and built with the usuall autotools scenario (*./configure && make && make install*).

In the `./configure` command, you may notice the specific `--prefix={{prefix}}` syntax. This makes uses of an *extension* that will be replaced on runtime by the *prefix* path for this build.

You may find more information on *builds* in the *Builders* section.

If you are used to Python development, you will notice something strange: we defined several times `Attendee('iconv')` yet it seems to refer to the same object. In **teapot**, instances of *Attendee* are memoized, meaning that any instanciation that uses the same name will actually refer to the same instance. The same goes for *Build* and some other classes. Obviously, this doesn't prevent you from assigning the instances to variables, like you would do in a regular Python script. So you may actually write the same script that way:

```
from teapot import *

iconv = Attendee('iconv')
iconv.add_source('http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.14.tar.gz')
iconv.add_build('default', environment='system')

iconv_default = Attendee('iconv').get_build('default')
iconv_default.add_command('./configure --prefix={{prefix}}')
iconv_default.add_command('make')
iconv_default.add_command('make install')
```

Instances of *Attendee* can be filtered. The *filter* can be specified either in the first instanciation of the *Attendee*, or later, using the `attendee.filter` property.

For instance, to make an *attendee* only exist on Windows, one could write:

```
from teapot import *

# During instanciation.
Attendee('iconv', filter='windows')

# Later.
Attendee('iconv').filter = 'windows'
```

You will learn more about filters in the *Filters* section.

Attendees can also depend on each other, using the `attendee.depends_on()` method.

```
from teapot import *

Attendee('a')
Attendee('b').depends_on('a')
Attendee('c').depends_on('a', 'b')
Attendee('d').depends_on('a', 'b', Attendee('c'))
```

The `depends_on()` method can take zero, one or several *attendee* names or instances.

> **Warning:** If the dependency graph is cyclic, *teapot* will notice it before even starting the build and will warn you about the problem.

Attendees can also have their custom prefix for installation. For instance, if one *attendee* needs to install inside a specific subfolder, you may write:

```
from teapot import *

set_option('prefix', '/tmp/output')
```

```
Attendee('iconv', prefix='subfolder')
# or
Attendee('iconv').prefix = 'subfolder'
```

If `prefix` is an absolute path, then the parent `prefix` is ignored.

Sources

A *source* can be anything you want. By default **teapot** supports three sources types:

http Fetches an archive from a web URL in a fashion similar to the **wget** command. This is the most commonly used fetcher.

Example formats:

- `http://host/path/archive.zip`
- `https://host/path/archive.zip`

file Fetches an archive from a filesystem path. The path can be either local or a network mount point.

Example formats:

- `file://~/archives/archive.tar.gz`
- `file://C:\archives\archive.zip`

folder Fetches an archive from a filesystem path. The path can be either local or a network mount point. The target must point to an already uncompressed source tree.

Example formats:

- `folder://~/archives/source`
- `folder://C:\archives\source`

github Generates and fetches an archive from a Github-hosted project.

Example formats:

- `github:user/repository/ref`

Sources are also filterable, following the same rules than for *attendees*.

teapot reads the mime type of the archives to extract them. If, for whatever reason, the mime type of the archive cannot be detected for a given source you may specify it in the `attendee.add_source()` method call, by specifying the `mimetype` named argument. This can happen for instance when a HTTP webserver is misconfigured and does not specify a `Content-Type` for a given archive.

Unpackers

At some point before the build, *teapot* must convert a downloaded (often compressed) archive into a source tree. This is what *unpackers* are for.

The unpacker selection is done automatically, depending on the mime type of the downloaded archive. That is, the only way to choose which unpacker to use, is to change the mimetype of the *source*.

By default, *teapot* provides the following unpackers:

Tarball unpacker An unpacker that can uncompress tarballs (*.tar.gz* and *.tar.bz2* files).

It recognizes the following mimetypes:

- *application/x-gzip*
- *application/x-bzip2*

Zipfile unpacker An unpacker that can uncompress zip archives (*.zip* files).

It recognizes only the *application/zip* mimetype.

Null unpacker An unpacker that does nothing. Useful for local files/directories.

It recognizes only the *(null, null)* mimetype.

You may also extend teapot and implement your own unpackers, should you have specific needs.

Note: You can specify some actions to perform after the unpacking process completed using the **:method:'teapot.attendee.Attendee.add_post_unpack_command'** method. These commands can have a filter.

Builders

One of the most important thing to declare into an *attendee*, is its *builds*. A *build* is responsible for taking an unarchived source tree and creating something by issuing a series of commands.

Builders are declared like so:

```
from teapot import *

Attendee('iconv').add_source('http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.14.tar.gz')
Attendee('iconv').add_build('default', environment='system')
Attendee('iconv').get_build('default').add_command('./configure --prefix={{prefix}}')
Attendee('iconv').get_build('default').add_command('make')
Attendee('iconv').get_build('default').add_command('make install')
```

In this simple example, *teapot* will go into the source tree unpacked from *libiconv-1.14.tar.gz* and will issue the following commar

- `./configure --prefix={{prefix}}`
- `make`
- `make install`

If all of these commands succeed, the build is considered successful as well.

Note: Here `{{prefix}}` is an extension that resolves at runtime as the current prefix for the *build*. You can learn more about extensions in the *Extensions* section.

One *attendee* can have as many different *builds* as you want.

Here is an example of a more complex *attendee*:

```
from teapot import *

Attendee('iconv').add_source('http://ftp.gnu.org/pub/gnu/libiconv/libiconv-1.14.tar.gz')
Attendee('iconv').add_build('default_x86', environment='mingw_x86')
Attendee('iconv').get_build('default_x86').add_command('./configure --prefix={{prefix}}')
Attendee('iconv').get_build('default_x86').add_command('make')
Attendee('iconv').get_build('default_x86').add_command('make install')

Attendee('iconv').add_build('default_x64', environment='mingw_x64')
Attendee('iconv').get_build('default_x64').add_command('./configure --prefix={{prefix}}')
```

```
Attendee('iconv').get_build('default_x64').add_command('make')
Attendee('iconv').get_build('default_x64').add_command('make install')
```

In this example, we define two builds (*default_x86* and *default_x64*) that have exactly the same build commands.

Each *build* has another *environment*. The current example lacks the environments definitions for simplicity's sake. You will learn how to define your own environments in a further section.

Builds can be filtered like *attendees* and can also have a custom *prefix*.

Environments

Environments define the execution environment of a *build*.

An *environment* can inherit from another *environment*.

Here is an example of *party file* that defines environments:

```
from teapot import *

Environment('mingw_x86', shell=["C:\\MinGW\\msys\\1.0\\bin\\bash.exe", "-c"], variables={'PATH': "C:\
Environment('mingw_x64', shell=["C:\\MinGW\\msys\\1.0\\bin\\bash.exe", "-c"], variables={'PATH': "C:\
```

In this example, we define two environments that use the same *shell* (here, *bash* for Windows). They both inherit from the *system* environment and each (re)define the `PATH` environment variable.

An *environment* dictionary understands the following attributes:

shell The *shell* to use.

shell can be a list of command arguments (with the executable as the first argument). This is the recommended way of specifying the *shell* as it is unambiguous.

If *shell* is a string, it will be parsed and split into a list using `shlex.split()`. This method of defining the shell and its arguments can be ambiguous and is therefore **not recommended**.

shell can also be `True` (the default), in which case its value will be taken from the parent *environment*, if it has one.

If no *shell* is specified, the default one from the system will be taken as specified in `subprocess.call()`.

variables A dictionary of environment variables to set, remove or override.

Each variable can be set to either a string, or to `None`.

The behavior a null value depends on the value of *parent*.

If the *environment* inherits its attributes from another *environment*, a null value indicates that the environment variable should be **removed** from the environment. This is **not** equivalent to setting its value to an empty string (in this case the variable would still be part of the environment, but would just be empty).

If the *environment* does not inherit its attributes from another *environment*, a null value indicates that the value for this environment variable should be the one of the execution environment (the environment into which *teapot* was called). If the environment variable was not set within the execution environment, it won't be set in the new environment if its value was `null`.

parent *parent* can be `None` (the default), or it can be the name of a named *environment* to inherit from.

If *parent* is null, none of the existing environment variables are inherited and only the ones defined in the *variables* attribute will be set.

Note: By default, *teapot* exposes the execution environment through the name `system`.

This `system` environment has all the environment variables that were set right before the call to *teapot* and uses the default system *shell*.

Filters

Filters are a way to differentiate *teapot* execution accross platforms and environments. A *filter* is basically a test whose result is boolean. It answers a simple question like: am on Windows ? Is MinGW available ?

teapot comes with several built-in filters:

Filter	Role
windows	Check that *teapot* is currently running on Windows.
linux	Check that *teapot* is currently running on Linux.
darwin	Check that *teapot* is currently running on Darwin (Mac OS X).
unix	Check that *teapot* is currently running on UNIX (Linux or Darwin).
msvc	Check that Microsoft Visual Studio is actually available in the current environment. It usually means *teapot* was started from a MSVC command shell.
msvc-x86	Check that Microsoft Visual Studio x86 is actually available in the current environment. It usually means *teapot* was started from a MSVC x86 command shell.
msvc-x64	Check that Microsoft Visual Studio x64 is actually available in the current environment. It usually means *teapot* was started from a MSVC x64 command shell.
mingw	Check that MinGW is available in the current environment. The filter will try to find *gcc.exe*.

All classes can refer to filters using their name (as a Python string) or directly (referring to a `teapot.filters.filter.Filter` instance).

teapot exposes two helper functions, *f* and *uf* which respectively stand for "filter" and "unnamed filter". Filters can be aggregated using standard bit-wise operators like so:

```
from teapot import *

# Define a new filter, named 'x64' that is verified if either of the filters 'mingw64' or 'gcc64' are
f('x64', f('mingw64') | f('gcc64'))

# Define a new filter, named 'foo' that is verified is we run on Windows and with MinGW or on UNIX bu
f('foo', (f('windows') & f('mingw')) | f('unix') & ~f('darwin'))

# Filters can also be created from variables or callables.
f('bar', uf(True) & uf(lambda: True))

# Finally, one can also use the 'named_filter' decorator to declare a custom filter.
@named_filter('has_foo')
def has_foo():
    return 'FOO' in os.environ()
```

Extensions

Extensions are simple functions, that optionally have parameters, which can occur in a *build* or post-unpack command.

For instance the *prefix* extension is resolved at runtime and replaced with the complete prefix (as defined at the root of the *party file*, the *attendee* and the *build*).

Valid syntaxes for calling extensions within commands are `{{extension}}` (no parameters) or `{{extension(1, 2, a=4, b="foo")}}` (parameters). Syntax for parametrized calls respect the Python function call syntax. That is, you can use positional arguments as well as named arguments.

teapot comes with several built-in extensions:

Extension	Parameters	Role
root	style	Get the absolute path to the root of the *party file*. Returns the complete path, in an operating system specific manner. On UNIX and its derivatives, forward slashes are used. On Windows, backwards slashes are used. If *style* is set to `unix`, forward slashes are used, even on Windows. This is useful inside MSys or Cygwin environments.
prefix	style	Get the complete prefix for the current attendee/build. Returns the complete path, in an operating system specific manner. On UNIX and its derivatives, forward slashes are used. On Windows, backwards slashes are used. If *style* is set to `unix`, forward slashes are used, even on Windows. This is useful inside MSys or Cygwin environments. *prefix* can contain extensions, as long as it doesn't call itself directly, or indirectly.
prefix_for	attendee, build, style	Get the complete prefix for the specified attendee/build. You must at least specify the *attendee* parameter. Returns the complete path, in an operating system specific manner. On UNIX and its derivatives, forward slashes are used. On Windows, backwards slashes are used. If *style* is set to `unix`, forward slashes are used, even on Windows. This is useful inside MSys or Cygwin environments. *prefix_for* can contain extensions, as long as it doesn't call itself directly, or indirectly.
attendee		Returns the current attendee name.
build		Returns the build name.
full_build		Returns the full build name, that begins with the *attendee*'s name.
archive_path	style	Returns the current archive path. On UNIX and its derivatives, forward slashes are used. On Windows, backwards slashes are used. If *style* is set to `unix`, forward slashes are used, even on Windows. This is useful inside MSys or Cygwin environments.
extracted_source_path	style	Returns the current source tree path. On UNIX and its derivatives, forward slashes are used. On Windows, backwards slashes are used. If *style* is set to `unix`, forward slashes are used, even on Windows. This is useful inside MSys or Cygwin environments. Since source trees are copied to a temporary location before the build, this is **not** the path were the build actually takes place.
msvc_version		Get the current Microsoft Visual Studio version, as a dotted version string. Example: "12.0"
msvc_toolset		Get the current Microsoft Visual Studio toolset. Example: "v120"

You may also define your own extensions, see `teapot.extensions.extension.register_extension()`.

Other settings

teapot runs with the following defaults:

Parameter	Default value	Meaning

cache_root `~/.teapot/cache` (UNIX) The path where the archives are downloaded to.

`%APPDATA%/teapot/cache` (Windows)

sources_root `~/.teapot/sources` (UNI The path where the sources are unpacked.

`%APPDATA%/teapot/sources` (Windows)

builds_root `~/.teapot/builds` (UNIX) The path where the builds take place.

`%APPDATA%/teapot/builds` (Windows)

prefix `~/.teapot/install` The default *party file* prefix that gets prepended to all *attendees* prefixes.

`%APPDATA%/teapot/install` (Windows)

These settings are to be set use the *set_option()* method, like so:

```
from teapot import *

set_option('prefix', 'install')
print get_option('prefix')
```

Note: When setting options, note that you can also specify a *filter* to restrict its effect on some platforms/in some environments.

Depending on your project, you may want to set the *cache_path* to a more local location (you may choose to add them to version control for instance).

5.1.2 Using *teapot*

teapot is the command line tool that ships with *teapot*.

```
$ teapot --help
usage: teapot [-h] [-d] [-v] [-p PARTY_FILE]
              {clean,fetch,unpack,build} ...

Manage third-party software.

positional arguments:
  {clean,fetch,unpack,build}
                        The available commands.
    clean               Clean the party.
    fetch               Fetch all the archives.
    unpack              Unpack all the fetched archives.
    build               Build the archives.

optional arguments:
  -h, --help            show this help message and exit
  -d, --debug           Enable debug output.
  -v, --verbose         Be more explicit about what happens.
  -p PARTY_FILE, --party-file PARTY_FILE
                        The party-file to read.
```

By default, *teapot* looks for a file named `Party` in the current directory. You may change the location of this file by using the `--party-file` option.

The *clean* command

teapot fetches the sources archives and stores them in the *cache* directory. It unpacks those archives in the *sources* directory. It also build attendees and stores the temporary results inside the *builds* directory.

Use `teapot clean` to clean either the *cache*, *sources* or the *builds* directory (or all of them).

The use of this command in normally not needed as *teapot* knows how to compute dependencies and detect changes automatically.

```
$ teapot clean --help
usage: teapot clean [-h] {cache,sources,builds,all} ...

positional arguments:
  {cache,sources,builds,all}  The available commands.
    cache            Clean the party cache.
    sources          Clean the party sources.
    builds           Clean the party builds.
    all              Clean the party cache, sources and builds.

optional arguments:
  -h, --help         show this help message and exit
```

The *clean cache* command

Cleans the *teapot* cache directory, where the source archives are stored.

Use this command if, for whatever reason you think the archive cache was corrupted.

If no *attendee* is specified, all the attendees are cleaned.

```
$ teapot clean cache --help
usage: teapot clean cache [-h] [attendee [attendee ...]]

positional arguments:
  attendee    The attendees to clean.

optional arguments:
  -h, --help  show this help message and exit
```

The *clean sources* command

Cleans the *teapot* sources directory, where the unpacked archives are stored.

Use this command if, for whatever reason you think the sources were corrupted.

If no *attendee* is specified, all the attendees are cleaned.

```
$ teapot clean sources --help
usage: teapot clean sources [-h] [attendee [attendee ...]]

positional arguments:
  attendee    The attendees to clean.
```

```
optional arguments:
  -h, --help  show this help message and exit
```

The *clean builds* command

Cleans the *teapot* builds directory, where the build results are stored.

Use this command if, for whatever reason you think the build results were corrupted.

If no *attendee* is specified, all the attendees are cleaned.

```
$ teapot clean builds --help
usage: teapot clean builds [-h] [attendee [attendee ...]]

positional arguments:
  attendee    The attendees to clean.

optional arguments:
  -h, --help  show this help message and exit
```

The *clean all* command

Cleans the *teapot* cache, sources and builds directories.

Use this command if, for whatever reason you want to reset the status of your current *teapot* project.

If no *attendee* is specified, all the attendees are cleaned.

```
$ teapot clean all --help
usage: teapot clean all [-h] [attendee [attendee ...]]

positional arguments:
  attendee    The attendees to clean.

optional arguments:
  -h, --help  show this help message and exit
```

The *fetch* command

Fetches the source archives of the specified *attendees*.

`teapot fetch` makes sure all the source archives are downloaded for the specified attendees.

If no *attendee* is specified, the source archives for all *attendees* are fetched.

By default, this command only fetches archives that weren't already downloaded. Use the `--force` option to force the download of all *attendees*.

```
$ teapot fetch --help
usage: teapot fetch [-h] [-f] [attendee [attendee ...]]

positional arguments:
  attendee     The attendees to fetch.

optional arguments:
  -h, --help   show this help message and exit
  -f, --force  Fetch archives even if they already exist in the cache.
```

The *unpack* command

Unpacks the fetched source archive to prepare for a build.

If no *attendee* is specified, all the attendees are unpacked.

```
$ teapot unpack --help
usage: teapot unpack [-h] [-f] [attendee [attendee ...]]

positional arguments:
  attendee     The attendees to unpack.

optional arguments:
  -h, --help   show this help message and exit
  -f, --force  Unpack archives even if they already exist in the build.
```

This step is usually not required as it performed automatically whenever needed. Use it when you don't want to build right away but want the next build to be as fast as possible.

Calling *unpack* automatically fetches the source archives if they are not present.

The *build* command

Builds the attendees.

If no *attendee* is specified, all the attendees are built. If a list of *attendees<attendee>* is specified, only those attendees and the ones they depend on will be built.

```
$ teapot build --help
usage: teapot build [-h] [-t tag] [-u] [-f] [-k] [attendee [attendee ...]]

positional arguments:
  attendee            The attendees to build.

optional arguments:
  -h, --help          show this help message and exit
  -f, --force         Build archives even if they were already built.
  -k, --keep-builds   Keep the build directories for inspection.
```

Only the builds that didn't succeeded the last time or the one that changed since the last build are run. To change that behavior, specify the `--force-build` option.

Temporary build directories are deleted automatically whenever a build terminates (either with a success or a failure), unless the `--keep-builds` option is specified. In that case, the build directory remains until the build gets restarted.

5.2 Glossary

party file The party file is a Python file, named *Party*, *Party.py*, *.party* or *.party.py* that can be located anywhere.

Within the party file, all paths are relative to the party file directory.

attendee An attendee is a fancy name for a third-party software to build.

A *party file* can contain as many attendees as you like, and different attendees can even represent the same third-party software if that makes sense in your situation.

source A source designates the location and the method where and how to fetch the source files for an *attendee*. While the most common case is downloading a file using HTTP, one can also copy a file locally, through a network share or from Github.

fetcher A fetcher is the entity that is responsible for handling a specific type of source.

Usually, fetchers are smart enough to recognize sources from their format and you should not have to care too much about them.

unpacker An unpacker is the entity that is responsible for turning a compressed archive (Zip file or tarball for instance) into a source tree.

build A build is an environment and a list of commands to execute in order to transform the attendee source into a compiled set of binaries (or whatever a build process can produce).

Builders rely a lot on *environments*.

environment An environment is a set of environment variables, shell value and inheritance parameters that wraps one or several builds.

Environments define the tools to use for a given build, and their options.

filter A filter is an entity whose role is to check if the current execution environment matches a series of criterias.

For instance, the *windows* filter checks that *teapot* has been run on Windows. Another example is the *mingw* filter whose role is to check that MinGW is currently available in the execution environment.

teapot *teapot* is the name of the command-line tool that implements all teapot logic.

shell A shell is a command line interpreter that will execute the different commands of a builder.

extension An extension is an entity the resides in builder commands and that gets replaced when the command is evaluated.

Extension are python function that can optionally take parameters.