yatr Documentation

Release 0.0.11b

Matt Bodenhamer

Jun 20, 2018

Contents

1 Contents: **3**
- 1.1 Installation . . . 3
- 1.2 Yatrfile Structure and Features . . . 3
- 1.3 Command-Line Interface . . . 15
- 1.4 Future Features . . . 20
- 1.5 yatr package . . . 20
- 1.6 Changelog . . . 28

2 Indices and tables **31**

Python Module Index **33**

Yet Another Task Runner. Or alternatively, YAml Task Runner. Yatr is a YAML-based task runner designed for 21st-century software-development contexts. The project is in the preliminary stages of development, but is nonetheless functional for a number of applications.

To install, run:

```
$ pip install yatr
```

More information can be found in the package documentation.

CHAPTER 1

Contents:

1.1 Installation

To install, simply run:

```
$ pip install yatr
```

Dynamic tab completions for bash are supported. To install, run:

```
$ yatr --install-bash-completions
```

As with invoking `pip`, you will need to have root access or use `sudo`.

Depending on your system configuration, configurable bash tab completions may not be enabled by default. If it is infeasible or undesirable to enable such functionality globally, then the file `/etc/bash_completion.d/yatr` will need to be sourced in `~/.bashrc`. This can be accomplished by adding the following line:

```
source /etc/bash_completion.d/yatr
```

1.2 Yatrfile Structure and Features

Suppose you have the following `yatrfile.yml` in your current working directory:

```
include:
  - "{{urlbase}}/test/test2.yml"

capture:
  baz: "ls {{glob}}"

macros:
  urlbase: https://raw.githubusercontent.com/mbodenhamer/yatrfiles/master/yatrfiles
  b: bar
```

(continues on next page)

(continued from previous page)

```
  c: "{{b}} baz"
  canard: "false"
  glob: "*.yml"

default: foo

tasks:
  cwd: pwd

  bar:
    - foo
    - "echo {{c}} {{_1|default('xyz')}}"

  verily: "true"

  cond1:
    command: 'echo "{{baz}}"'
    if: "true"

  cond2:
    command: foo
    if: "false"

  cond3:
    command: foo
    ifnot: verily

  cond4:
    command: foo
    ifnot: "{{canard}}"
```

As this example demonstrates, the primary functionality of a yatrfile is found in five top-level sections: `include`, `capture`, `macros`, `tasks`, and `default`. Four other sections, `files`, `settings`, `import`, and `declare`, are also supported (see *files*, *settings*, *import*, and *declare*, respectively).

1.2.1 macros

The `macros` section must be a mapping of macro names to macro definitions. Macro definitions may either be plain strings or Jinja2 templates. Macros that include Jinja2 templates will be rendered according to the values of the macros in terms of which they are defined. For example, in the above `macros` section, two macros `b` and `c` are defined thusly:

```
b: bar
c: "{{b}} baz"
```

As such, `b` resolves to `bar` and `c` resolves to `bar baz`. As the `macros` section is a mapping, and not a list, there is no inherent order to macro definition. yatr takes care of resolving macros and their dependencies in the right order, provided that there are no cyclic macro definitions (e.g. a macro `a` defined in terms of `b`, which is defined in terms of `a`). If any such cycles exist, the program will exit with an error.

1.2.2 include

The `include` section must be a list of strings, each of which must be either a filesystem path or a URL specifying the location of another yatrfile. When a yatrfile is "included" in this manner, its macros and tasks are added to the macros

and tasks defined by the main yatrfile. Nested includes are supported, following the rule that conflicts in macro or task names are resolved by favoring the definition closest to the main yatrfile.

For example, suppose yatr is invoked on a yatrfile named `C.yml`, which includes `B.yml`, which includes `A.yml`, as follows:

`A.yml`:

```
macros:
  a: foo
  b: def
  c: xyz
```

`B.yml`:

```
include:
  - A.yml

macros:
  a: bar
  b: ghi
```

`C.yml`:

```
include:
  - B.yml

macros:
  a: baz
```

In this case, the macro values would resolve as follows:

```
$ yatr -f C.yml --dump
a = baz
b = ghi
c = xyz
```

Name conflicts of tasks from includes are resolved the same way as for macros.

Include paths or URLs may use macros, as the main yatrfile above demonstrates, having an include defined in terms of the `urlbase` macro. However, any such macros must be defined in the yatrfile itself, and cannot be defined in an included yatrfile or depend on the macros defined in an included yatrfile for their proper resolution.

If an include path is a URL, yatr will attempt to download the file and save it in a cache directory. By default, the cache directory is set to `~/.yatr/`, but this may be changed through the `--cache-dir` option. If the URL file already exists in the cache directory, yatr will load the cached file without downloading. To force yatr to re-download all URL includes specified by the yatrfile, run `yatr --pull` at the command line.

1.2.3 `tasks`

Tasks are defined in the `tasks` section of the yatrfile. Tasks may be defined as a single command string. In this example, the task `cwd` is simply defined as the system command `pwd`. If your current working directory happens to be `/foo/baz`, then:

```
$ yatr cwd
/foo/baz
```

Macros are not fully resolved until task runtime. The example yatrfile specifies the inclusion of a file named test2.yml, which defines a task named `foo`. However, `foo` is defined in terms of a macro named `b`, which is not defined in `test2.yml`. The macro `b` is defined in the main yatrfile, however, which induces the following behavior:

```
$ yatr foo
bar
```

Tasks may also be defined as a list of command strings, to be executed one after the other, as illustrated by `bar`:

```
$ yatr bar
bar
bar baz xyz
```

If the command string is the name of a defined task, then yatr will simply execute that task instead of trying to execute that string as a system command. The `bar` task will first execute the `foo` task defined in test2.yml, and then run the `echo` command.

The `bar` task also illustrates another feature of yatr: command-line arguments may be passed to tasks for execution. For example:

```
$ yatr bar foo
bar
bar baz foo
```

Unless, explicitly re-defined, the macro `_1` denotes the first task command-line argument, `_2` denotes the second task command-line argument, and so on. Default values may be specified using the Jinja2 `default` filter, as is illustrated in the definition of `bar`.

1.2.4 default

The `default` section, if specified, must contain the name of a task to be run if no task names are provided at the command line. In this example, the default task is set to `foo`:

```
default: foo
```

As such, running `yatr` at the command line is equivalent to running `yatr foo`:

```
$ yatr
bar
```

If no default task is defined, and if yatr is invoked without any arguments, then yatr will exit after printing usage information.

1.2.5 capture

The `capture` section defines a special type of macro, specifying a mapping from a macro name to a system command whose captured output is to be the value of the macro. Values of `capture` mappings cannot contain task references, though they may contain references to other macros. In the main example above, the yatrfile defines a capture macro named `baz`, whose definition is `ls {{glob}}`. In the macro section, `glob` is defined as `*.yml`. Thus, if yatr is invoked in the example working directory, the value of `baz` will resolve to `A.yml B.yml C.yml D.yml yatrfile.yml`.

1.2.6 files

The files section defines another special type of macro, associating names with filesystem paths. Each associated path must either be a filesystem path or a URL specifying the location of a file. As with the *include* and *import* sections, if the path is a URL, the file will be downloaded to the cache directory and the associated name will contain the path of the cached file. If the file already exists in the cache directory, no download will be performed, unless yatr --pull is run. The files section has the same limited support for macros as the *include* and *import* sections.

For example, consider the following yatrfile:

```
files:
  test1: "{{urlbase}}/test/test1.txt"

macros:
  urlbase: https://raw.githubusercontent.com/mbodenhamer/yatrfiles/master/yatrfiles
  tmpdir: "{{_1}}"

tasks:
  foo: 'cp "{{test1}}" "{{tmpdir}}/test1.txt"'
  bar: 'cat "{{tmpdir}}/test1.txt"'
```

Invoking yatr will cause test1.txt to be downloaded to the cache directory. Running the tasks defined in this yatrfile produces the following behavior:

```
$ yatr foo /tmp
$ yatr bar /tmp
foo
```

The first invocation of yatr downloads test1.txt to the cache directory, and copies the file to /tmp. The second invocation dumps the contents of the copied file to stdout.

1.2.7 settings

The top-level section settings allows the global execution behavior of yatr to be modified in various ways. For example, the silent setting, if set to true, will suppress all system command output at the console. Such behavior is disabled by default.

An example of setting setting values in a yatrfile can be found in D.yml, which includes the example yatrfile discussed in *Yatrfile Structure and Features*:

```
include:
  - yatrfile.yml

settings:
  silent: true
```

In the *example* above, running yatr foo led to the output bar being printed to the console. However, invoking the same task through D.yml will result in no output being printed:

```
$ yatr -f D.yml foo
```

However, any setting can be set or overridden at the command line by supplying the -s option:

```
$ yatr -f D.yml -s silent=false foo
bar
```

For Boolean-type settings, such as `silent`, any of the following strings may be used to denote True, regardless of capitalization: `yes`, `true`, `1`. Likewise, any of the following strings may be used to denote False, regardless of capitalization: `no`, `false`, `0`.

The following table lists the available settings:

Name	Description
`exit_on_error`	If true, halt execution if task command exits with non-zero status; true by default (Boolean string)
`loop_count_macro`	The name of the macro that contains the current loop iteration number (string)
`preview_conditionals`	If true, specially demarcate `if` and `ifnot` commands when `-p` is supplied; true by default (Boolean string)
`silent`	If true, suppress task output; false by default (Boolean string)

1.2.8 `import`

The `import` feature enables the functionality of yatr to be extended when necessary, while preserving the simplicity of the default YAML specification for the majority of use cases in which yatr's default capabilities are sufficient.

The `import` section must be a list of strings, each of which must be either a filesystem path or a URL specifying the location of a Python module. Strings containing Python module names (such as would be found in a Python `import` statement) are also supported. Each module so imported must contain a top-level variable named `env`, which must be an instance of the `Env` class (see yatr.env module). Modules to be imported in this manner are called "extension modules" (not to be confused with Python extension modules written in C/C++).

The following is an example of a yatr extension module:

```
from yatr import Env
env = Env()

@env.jinja_filter('foo')
def foo(value, **kwargs):
    return '{}_foo'.format(value)

@env.jinja_function('bar')
def bar(value, **kwargs):
    return '{}_bar'.format(value)

@env.task('barfoo', display=('path', 'baz1'))
def bar_foo(env, *args, **kwargs):
    import os
    print(os.path.join(kwargs['path'], kwargs['baz1']))
```

This particular extension module defines a custom Jinja2 function (see *Custom Jinja2 Functions*) and a custom Jinja2 filter (see *Custom Jinja2 Filters*). A task is also defined in terms of a Python function. Macros can theoretically be defined in an extension module through use of the yatr API, but the straightforward manner of macro declaration facilitated by the standard yatrfile YAML syntax makes the use of `include` directives a much more efficient and user-friendly alternative.

In this example extension module, a Jinja2 function `bar` is defined that appends "_bar" to its first argument. Likewise, a Jinja2 filter `foo` is defined that appends "_foo" to its first argument. Because yatr supplies the current execution environment to custom Jinja 2 filters and functions by way of a keyword argument named `env`, all such filters and functions defined in extension modules should accept `**kwargs` as the final argument, even if the `kwargs` variable is not used within the body of the filter or function itself.

Here is an example yatrfile that uses the extension module defined above:

```
import:
  - test8.py

macros:
  shebang: "#!/bin/bash"
  bar: "{{env('YATR_BAR', 'baz')}}"
  path: "{{env('PATH')}}"
  baz1: "{{'baz'|foo}}"
  baz2: "{{bar('foo')}}"

tasks:
  foo:
    - echo foo
    - echo bar
    - echo baz

  bar:
    - "echo {{bar}}"

  path:
    - "echo {{path}}"

  baz: "echo {{baz1}} {{baz2}}"
```

In addition to `bar` and `foo`, this yatrfile also makes use of the built-in custom Jinja2 function `env` (see *env()*). The task `baz` is defined in terms of macros that make use of `bar` and `foo`. Invoking the task produces the following output:

```
$ yatr baz
baz_foo foo_bar
```

In addition to custom Jinja2 functions and filters, extension modules can also be used to define tasks that execute as Python callables. In this example, the extension module defines a function named `bar_foo` that will be defined in the yatr execution environment as a task named `barfoo`. Extension tasks have access to all defined macro values through the first parameter, `env` (see yatr.env module). Moreover, any extension tasks defined using the `@env.task` decorator will also receive all defined macros through the `*args` and `**kwargs` arguments: `*args` will be populated with any positional argument macros that are defined (i.e., `_1`, `_2`, `_3`, etc.), and `**kwargs` will be populated with all defined macro values that are not positional argument macros. While these values can also be accessed via `env`, the `*args` and `**kwargs` parameters are notable in that they represent the current execution environment. In cases where macros in different sections are defined with the same name, using `**kwargs` enables the programmer to access the actual execution value for that name without having to replicate yatr's macro precedence logic in the extension function.

In this example, suppose the value of the environment variable `PATH` is set to `/foo/bar`. In such case, executing the extension task `barfoo` produces the following output:

```
$ yatr barfoo
/foo/bar/baz_foo
```

When executing extension tasks defined via the `@env.task` decorator, yatr will treat any function that does not raise an exception as exiting with return code 0. Likewise, a function that raises an exception is treated as exiting with return code 1. If extension functions are defined without using the `@env.task` decorator, the programmer should ensure that the function returns either 0 or 1, as appropriate.

The preview and verbose options (`-p` and `-v`) also work with extension functions. By default, yatr will print the function name, along with the full contents of `*args` and `**kwargs`. As many more macros may be defined than are used in the extension function, the optional `display` keyword argument may be provided in the `@env.task`

decorator, allowing the programmer to specify only those keyword arguments to be displayed. In the above example, executing the `barfoo` extension task with verbose output would produce the following behavior:

```
$ yatr -v barfoo
bar_foo(baz1=baz_foo, path=/foo/bar)
/foo/bar/baz_foo
```

1.2.9 declare

The `declare` section must be a list of strings, each of which must be the name of a macro. The function of this section is best explained by example. Suppose a macro named `foo` is included as part of the definition of either a macro or a task. If `foo` is not defined in the yatrfile (or any included yatrfiles), the yatrfile will fail validation (see *Commands*, particularly *–validate*). If `foo` is included in the `declare` section, however, yatr will effectively ignore the macro and allow the yatrfile to validate.

The `declare` section is necessary for validating yatrfiles that make use of certain macros that are only defined at runtime. Yatr will automatically handle cases of builtin runtime-defined macros (such as `_1`), and these do not need to be included in the `declare` section. However, any runtime-defined macros that are not builtin to yatr will need to be included in the `declare` section in order for the yatrfile to validate successfully. An example of using the `declare` section is included in *List Macros and For Loops*.

1.2.10 Custom Jinja2 Functions

The following functions are defined by default for use in Jinja2 templates.

commands()

The `commands` function takes a single argument and prints the commands corresponding to the execution of the task whose name is the argument. For example, suppose one is using the example yatrfile of the *import* section above in order to run a *–render* command on the following template file (`template.j2`):

```
{{shebang}}
{{commands('foo')}}
```

One could then render the template like so:

```
yatr -i template.j2 -o template.bash --render
```

The resulting output file (`template.bash`) would look like:

```
#!/bin/bash
echo foo
echo bar
echo baz
```

env()

The `env` function takes either one or two arguments. In either case, the first argument must be the name of an environment variable. The `env` function will return the value of this environment variable if it is defined. If the environment variable is undefined and only one argument is supplied to `env`, the function will raise an exception and halt execution of the task. On the other hand, if a second argument is supplied to `env`, it will be returned in the case that the environment variable in question is undefined.

For example, consider the example yatrfile of the *import* section above. The `home` macro is defined in terms of the environment variable `PATH`. In the practically-inconceivable case that `PATH` is not defined, yatr will exit with an exception when loading this yatrfile. On the other hand, in an environment in which `YATR_BAR` is not defined, the program will behave as follows:

```
$ yatr bar
baz
$ YATR_BAR=foo yatr bar
foo
```

1.2.11 Custom Jinja2 Filters

There are currently no custom Jinja2 filters defined by default for use in Jinja2 templates, but some will probably be added in future releases.

1.2.12 Conditional Task Execution

Tasks may be defined to execute conditionally upon the successful execution of a command, using the keys `if` and `ifnot`. If these or other command options are used, the command itself must be explicitly identified by use of the `command` key. These principles are illustrated in the `cond1`, `cond2`, `cond3`, and `cond4` tasks:

```
$ yatr cond1
A.yml  B.yml  C.yml  D.yml  yatrfile.yml
$ yatr cond2
$ yatr cond3
$ yatr cond4
bar
```

The values supplied to `if` and `ifnot` may be anything that would otherwise constitute a valid task definition. If a value is supplied for `if`, the command will be executed only if the return code of the test command is zero. Likewise, if a value is supplied for `ifnot`, the command will be executed only if the return code of the test command is non-zero.

1.2.13 List Macros and For Loops

In most use cases, macros will either be plain strings or Jinja2 templates. However, there are some cases in which it is useful to have a list of strings or macros defined itself as a macro. To define such a "list macro", simply use YAML list syntax in the macro definition. For example, consider the following yatrfile:

```
declare:
  - count

macros:
  a: x
  b:
    - "{{a}}"
    - "y"
  c:
    - w
    - z

tasks:
  foo:
    command: "echo {{a}} {{u}} {{v}} {{_n}}"
```

(continues on next page)

(continued from previous page)

```
    for:
      var:
        - u
        - v
      in:
        - b
        - c

  bar:
    command: "echo {{x}} {{count}}"
    for:
      var: x
      in: [1, 2, 3, 4]
```

The macro `a` is a plain string, but both `b` and `c` are list macros. List macros can be used for iteration via for loops, as is illustrated by the definitions of the tasks named `foo` and `bar`.

The `for` key requires two sub-keys, `var` and `in`. The `var` sub-key defines the iteration variable(s), while the `in` sub-key specifies the lists or list macros over which to iterate. In the case that `var` is a string value, `for` specifies a simple and intuitive for loop over the values specified by `in`. The value of `in` may either be the name of a list macro, as in the task named `foo`, or a list literal, as in the task named `bar`. In the case that `var` is a list, `for` specifies a loop over the Cartesian product of the lists specified by `in`. The task named `foo` illustrates a 2x2 Cartesian product, while the task named `bar` illustrates a simple for loop.

It should be noted that the local variables defined by `var` only exist in the context of the execution of the loop. It should also be noted that the for loop defines a special local variable named `_n`, which contains the current iteration number. Note that the task named `foo` is defined in terms of `_n`. As such:

```
$ yatr foo
x x w 0
x x z 1
x y w 2
x y z 3
```

The name of `_n` may be changed if desired via the `loop_count_macro` setting. For example:

```
$ yatr -s loop_count_macro=count bar
1 0
2 1
3 2
4 3
```

Note that the macro `count` is included in the `declare` section (see *declare*) to allow the yatrfile to validate successfully.

1.2.14 Dictionary Macros

In addition to list macros, yatr also supports the use of dictionary macros. To define such a "dictionary macro", simply use YAML dictionary syntax in the macro definition. For example, consider the following yatrfile:

```
macros:
  a: abc
  b:
    a: def
    b: "{{a}}"
```

(continues on next page)

(continued from previous page)

```
  c: "{{b.a}} {{b.b}}"

  d:
    a:
      b: 1
    b:
      b: 2
    c:
      b: 3

tasks:
  foo: "echo {{c}}"
  bar: "echo {{d[_1].b}}"
```

The macro `b` is a dictionary macro that is defined in terms of `a`. A second string macro, `c`, is defined in terms of the two items in `b`. As such:

```
$ yatr foo
def abc
```

1.2.15 Calling Tasks with Arguments

Tasks can be called from other tasks by providing the name of a task as the value to the `command` key. When a task is called in this manner, its macros can also be overridden using the `args` and `kwargs` keys. Values in `args` will override `_1`, `_2`, and so on, while values in `kwargs` will override named macros. These macro overrides only take effect for that specific task call, and do not change macro values globally.

Consider the following example yatrfile:

```
macros:
  x: 5
  y: 10

tasks:
  x: "echo {{x}} {{_1|default(3)}} {{ARGS[0]|default(10)}}"

  y:
    command: x
    args:
      - 1
      - 2
    kwargs:
      x: 7

  z:
    command: x
    args:
      - 6
    for:
      var: x
      in: [1, 2, 3]

  w:
    - x
    - "y"
```

(continues on next page)

(continued from previous page)

```
    - x

  u:
    - echo foo
    - task:
        command: x
        args:
          - "{{y * 2}}"
        kwargs:
          x: "{{y * 3}}"
```

The task `y` shows how macro values may be overridden in a task definition:

```
$ yatr x
5 3 10

$ yatr x 4
5 4 4

$ yatr y 4
7 1 4
```

In calling `x`, `y` overrides `_1` and `_2` (which is not used by `x`), but does not affect the builtin macro `ARGS` (see *Builtin Macros*). Note that `ARGS[0]` is equivalent to `_1`, unless `_1` is overridden locally through a task call.

The task `z` shows that calling tasks in this manner is compatible with for loop functionality:

```
$ yatr z 4
1 6 4
2 6 4
3 6 4
```

The task `w` shows that calling tasks in this manner does not change global macro values:

```
$ yatr w 4
5 4 4
7 1 4
5 4 4
```

Tasks can also be defined anonymously within task list definitions using the `task` keyword, as illustrated by the task `u`:

```
$ yatr u
foo
30 20 10
```

Note also that the `args` and `kwargs` of the task `u` are defined in terms of the macro `y`.

1.2.16 Builtin Macros

The following macros are defined by default:

Name	Description
ARGS	Command-line task arguments (list of strings)
CURDIR	Yatrfile directory path (string)
YATR	Invocation of yatr on current yatrfile (string)
YATRFILE	Yatrfile path (string)

The use of these macros is illustrated in the following yatrfile (test11.yml):

```
tasks:
  err: yatr --render -i /foo/bar/baz -o /foo/bar/bazz

  foo: "echo {{_1}} >> {{_2}}"
  bar:
    - foo
    - "yatr -f {{YATRFILE}} foo a {{_2}}"
    - "{{YATR}} foo c {{_2}}"
    - "yatr -f {{CURDIR}}/test11.yml foo d {{ARGS[1]}}"
    - foo

  baz:
    - err
    - foo
```

For example, suppose bar is invoked in the following manner on an empty file /tmp/foo:

```
$ yatr bar b /tmp/foo
```

The file /tmp/foo will now contain the following:

```
b
a
c
d
b
```

The other tasks illustrate the use of the exit_on_error setting (see *settings*). Supposing that neither /foo/bar/baz or /foo/bar/bazz exist on the filesystem, attempting to run baz with default settings will result in an error and foo will not be run. On the other hand, foo will run if baz is invoked like so:

```
$ yatr -s exit_on_error=false baz a /tmp/baz
```

If /tmp/baz was an empty file, it will now contain:

```
a
```

1.3 Command-Line Interface

1.3.1 Usage

```
usage: yatr [-h] [-f <yatrfile>] [-i <file>] [-o <file>] [-m <macro>=<value>]
            [-s <setting>=<value>] [--cache-dir <DIR>] [-v] [-p] [--cache]
            [--dump] [--dump-path] [--pull] [--render] [--version]
```

(continues on next page)

(continued from previous page)

```
            [--validate] [--install-bash-completions]
            [<task>] [ARGS [ARGS ...]]

Yet Another Task Runner.

positional arguments:
  <task>                The task to run
  ARGS                  Additional arguments for the task

optional arguments:
  -h, --help            show this help message and exit
  -f <yatrfile>, --yatrfile <yatrfile>
                        The yatrfile to load
  -i <file>             Input file
  -o <file>             Output file
  -m <macro>=<value>, --macro <macro>=<value>
                        Set/override macro with specified value
  -s <setting>=<value>, --setting <setting>=<value>
                        Set/override setting with specified value
  --cache-dir <DIR>     Path of cache directory
  -v, --verbose         Print commands to be run
  -p, --preview         Preview commands to be run without running them
                        (implies -v)
  --cache               Cache local input file (-i) as if it were a URL (-o)
  --dump                Dump macro values and exit
  --dump-path           Print yatrfile path and exit
  --pull                Download all URL includes and imports, then exit
  --render              Use macros to render a Jinja2 template file (requires
                        -i and -o)
  --version             Print version info and exit
  --validate            Validate the yatrfile and exit
  --install-bash-completions
                        Install bash tab completion script globally, then exit
```

1.3.2 Options

`--cachedir`

Certain yatr features make use of a cache directory to increase the efficiency of repeated yatr invocations. The cache directory is currently used for processing yatrfiles included via URL (see *include*), extension modules included via URL (see *import*), as well as for populating bash tab completion values. By default, the cache directory is set to `~/.yatr/`, but this may be changed like so:

```
$ yatr --cache-dir /path/to/cache/dir <some task or command>
```

`-f (--yatrfile)`

Specify the yatrfile to load. If this option is not supplied, yatr will try to load a file whose filename matches the regular expression `^[Yy]atrfile(.yml)?$`. If such a file is not present in the current working directory, yatr will search rootward up the filesystem tree looking for a file that matches the expression. This is intended as a feature of convenience, so that tasks can be easily executed when working in a project sub-directory. If it is unclear which yatrfile has been loaded, `yatr --dump-path` may be run to disambiguate.

-i and -o

The -i and -o options specify input and output files, respectively. These options have no effect when invoking yatr to run a task, and are only used by specific yatr commands that require them, such as *–render*.

-m (--macro)

Macro values may also be set or overridden at the command line by supplying the -m (or --macro) option. For example:

```
$ yatr -f C.yml -m a=zab --macro d=jkl --dump
a = zab
b = ghi
c = xyz
d = jkl
```

(See *C.yml*)

-s (--setting)

Any setting value may be set or overridden at the command line by supplying the -s (or --setting) option. For example:

```
$ yatr -f D.yml -s silent=false foo
bar
```

(See *D.yml*)

-v and -p

If the -v option is supplied at the command line, yatr will print the commands to be run before running them:

```
$ yatr -v bar foo
echo bar
bar
echo bar baz foo
bar baz foo
```

If the -p option is supplied, yatr will simply print the commands without running them:

```
$ yatr -p bar foo
echo bar
echo bar baz foo
```

(See *main example*)

1.3.3 Commands

As its name implies, yatr is primarily a task runner. As such, its default execution behavior is to run tasks defined in a yatrfile. However, when using a task runner in real-world applications, there are often situations where other execution behaviors become desirable. For example, if it becomes necessary to debug a particular yatrfile, dumping the values of the macros (via yatr --dump) might prove helpful. As such, yatr supports a number of special execution behaviors, called "commands", which do not run tasks. To avoid unnecessarily restricting the set of potential task names, all yatr

commands are prefixed by `--`. However, unlike normal command-line options, at most one command should be specified at the command line for any yatr invocation.

The following table lists the available commands:

Name	Description
`--cache`	Cache local input file (`-i`) as if it were a URL (`-o`)
`--dump`	Dump macro values to `stdout`
`--dump-path`	Print yatrfile path to `stdout`
`--install-bash-completions`	Install bash tab completion script in `/etc/bash_completions.d/`
`--pull`	Download all URL includes and imports defined in yatrfile
`--render`	Use macros to render a Jinja2 template file (requires `-i` and `-o`)
`--validate`	Validate the yatrfile
`--version`	Print version information to `stdout`

A discussion of each command is provided below.

`--cache`

Saves a local file specified by `-i` to the cache directory, as if it had been downloaded from a URL specified by `-o`. For example:

```
$ yatr --cache -i test.txt -o http://foo.com/bar.txt
```

In this example, the file `test.txt` will be copied into the cache directory with a filename corresponding to the URL `http://foo.com/bar.txt`. This command can be useful in yatrfile development, allowing one to test included functionality without having to upload the included yatrfile(s) every time a change is made.

`--dump`

Prints macro values (including `capture` values) to `stdout`. For example, with *C.yml*, running `--dump` produces the following:

```
$ yatr -f C.yml --dump
a = baz
b = ghi
c = xyz
```

`--dump-path`

Prints the absolute path of the loaded yatrfile. For example:

```
$ yatr -f /path/to/yatrfile.yml --dump-path
/path/to/yatrfile.yml
```

`--install-bash-completions`

Installs script for dynamic bash tab completion support. See *Installation*.

--pull

Downloads all URL includes and imports defined in the loaded yatrfile. If included yatrfiles define URL imports or includes, these will also be downloaded.

--render

The `--render` command renders a Jinja2 template file using the macros defined by a yatrfile. For example, suppose one has the following template for a Dockerfile named `Dockerfile.j2`:

```
# -*- dockerfile -*-
FROM python:2-alpine

RUN pip install -U --no-cache \
    syn>={{version}}

CMD ["python2"]
```

Suppose one also has the following `yatrfile.yml` in the same directory:

```
macros:
  version: 0.0.14
  image: foo

tasks:
  render: yatr --render -i Dockerfile.j2 -o Dockerfile
  build:
    - render
    - "docker build -t {{image}}:latest ."
```

One could then run:

```
$ yatr build
```

to generate the desired Dockerfile and then build the desired Docker image. The generated Dockerfile would look like:

```
# -*- dockerfile -*-
FROM python:2-alpine

RUN pip install -U --no-cache \
    syn>=0.0.14

CMD ["python2"]
```

--validate

Validates the loaded yatrfile. A number of validation tasks are performed during the course of loading a yatrfile (such as validating proper YAML syntax) even if the `--validate` command is not given. However, the `--validate` command validates further aspects of the loaded task environment, such as ensuring that no task definitions contain undefined macros. If an error is found, an exception will be raised and the program will terminate with a non-zero exit status.

--version

Prints the program name and current version to `stdout`.

1.4 Future Features

As an inspection of the source code might reveal, two additional top-level keys are also allowed in a yatrfile: `secrets`, and `contexts`. The `secrets` section defines a special type of macro, specifying a list of names corresponding to secrets that should not be stored as plaintext. In future releases, yatr will attempt to find these values in the user keyring, and then prompt the user to enter their values via stdin if not present. There will also be an option to store values so entered in the user keyring to avoid having to re-enter them on future task invocations. No support for secrets is implemented at present, however.

The `contexts` section allows the specification of custom execution contexts in which tasks are invoked. For example, one might define a custom shell execution context that specifies the values of various environment variables to avoid cluttering up a task definition with extra macros or statements. This feature is not currently supported, and its future is uncertain.

1.5 yatr package

1.5.1 Subpackages

yatr.tests package

Submodules

yatr.tests.mod1 module

yatr.tests.mod2 module

class yatr.tests.mod2.**Foo**(***kwargs*)

Bases: *yatr.context.Context*

Keyword-Only Arguments:

_skip_validation (*default* = False): *bool* envvars: *dict* (any => *basestring*) inside (*default* =): *basestring* opts: *dict*

Class Options:

- args: ()
- autodoc: True
- coerce_args: False
- id_equality: False
- init_validate: True
- make_hashable: False
- make_type_object: True
- optional_none: False
- register_subclasses: False
- repr_template:
- coerce_hooks: ()

- create_hooks: ()
- init_hooks: ()
- init_order: ()
- metaclass_lookup: ('coerce_hooks', 'init_hooks', 'create_hooks', 'setstate_hooks')
- setstate_hooks: ()

Groups:

- _all: _skip_validation, envvars, inside, opts
- _internal: _skip_validation

```
context_name = 'foo'
```

yatr.tests.test_base module

yatr.tests.test_context module

yatr.tests.test_env module

yatr.tests.test_env_decorators module

yatr.tests.test_parse module

yatr.tests.test_task module

Module contents

1.5.2 Submodules

1.5.3 yatr.base module

exception yatr.base.**ValidationError**
Bases: exceptions.Exception

1.5.4 yatr.context module

class yatr.context.**Context**(**kwargs*)
Bases: syn.base.b.base.Base

Keyword-Only Arguments:

_skip_validation (*default* = False): *bool* envvars: *dict* (any => *basestring*) inside (*default* =): *basestring* opts: *dict*

Class Options:

- args: ()
- autodoc: True
- coerce_args: False

- id_equality: False
- init_validate: True
- make_hashable: False
- make_type_object: True
- optional_none: False
- register_subclasses: False
- repr_template:
- coerce_hooks: ()
- create_hooks: ()
- init_hooks: ()
- init_order: ()
- metaclass_lookup: ('coerce_hooks', 'init_hooks', 'create_hooks', 'setstate_hooks')
- setstate_hooks: ()

Groups:

- _all: _skip_validation, envvars, inside, opts
- _internal: _skip_validation

context_name = None

classmethod from_yaml(*name*, *dct*)

required_opts = ()

resolve_macros(*env*, ***kwargs*)

run = None

run_command(*command*, *env*, ***kwargs*)

validate()
Raise an exception if the object is missing required attributes, or if the attributes are of an invalid type.

verbose(*command*, *env*, ***kwargs*)

1.5.5 yatr.env module

class yatr.env.**Env**(***kwargs*)
Bases: syn.base.b.base.Base, yatr.env.Copyable, yatr.env.Updateable

Keyword-Only Arguments:

captures: *dict* (any => *basestring*) Commands to captures output of

commandline_macros: *dict* (any => *basestring*) Macros defined with -m

contexts: *dict* (any => *Context*) Execution context definitions

declares: *set* Declared runtime-defined macros

default_context: *Context* Execution context to use if none is specified in task definition

default_task (*default* =): *basestring* Task to run if no task is specified at the command line

env: ***dict*** **(any =>** ***basestring*** **|** ***int*** **|** ***float*** **|** ***list*** **(*basestring* | *int* | *float* | *list* | *dict*) | *dict* (any => *basestring* | *int* | *float* | *list* | *dict*))**
Current name resolution environment

files: ***dict*** **(any =>** ***basestring*****)** File name macros

function_aliases: ***dict*** **(any =>** ***basestring*****)** Jinja function aliases

jenv: ***Environment*** Jinja2 environment

jinja_filters: ***dict*** **(any => <callable>)** Custom Jinja2 filters

jinja_functions: ***dict*** **(any => <callable>)** Custom Jinja2 functions

macros: ***dict*** **(any =>** ***basestring*** **|** ***int*** **|** ***float*** **|** ***list*** **(*basestring* | *int* | *float* | *list* | *dict*) | *dict* (any => *basestring* | *int* | *float* | *list* | *dic*
Macro definitions

secret_values: ***dict*** **(any =>** ***basestring*****)** Secret value store

settings: ***dict*** **(any => any)** Global settings of various sorts

tasks: ***dict*** **(any =>** ***Task*****)** Task definitions

Class Options:

- args: ()
- autodoc: True
- coerce_args: False
- id_equality: False
- init_validate: True
- make_hashable: False
- make_type_object: True
- optional_none: False
- register_subclasses: False
- repr_template:
- coerce_hooks: ()
- create_hooks: ()
- init_hooks: ()
- init_order: ()
- metaclass_lookup: ('coerce_hooks', 'init_hooks', 'create_hooks', 'setstate_hooks')
- setstate_hooks: ()

Groups:

- _all: captures, commandline_macros, contexts, declares, default_context, default_task, env, files, function_aliases, jenv, jinja_filters, jinja_functions, macros, secret_values, settings, tasks
- copy_copy: captures, commandline_macros, contexts, declares, env, files, function_aliases, jinja_filters, jinja_functions, macros, secret_values, settings, tasks
- assign_update: default_context, default_task
- _internal: function_aliases
- dict_update: captures, commandline_macros, contexts, env, files, function_aliases, jinja_filters, jinja_functions, macros, secret_values, settings, tasks

- eq_exclude: jenv
- set_update: declares

capture_value(*cmd*, ***kwargs*)

jinja_filter(*name*, **args*, ***kwargs*)

jinja_function(*name*, **args*, ***kwargs*)

macro_env(***kwargs*)

resolve(*template*, ***kwargs*)

resolve_arg_macros(*arg_macros*, ***kwargs*)

resolve_macros(***kwargs*)

task(*name*, **args*, ***kwargs*)

validate()
Raise an exception if the object is missing required attributes, or if the attributes are of an invalid type.

1.5.6 yatr.env_decorators module

1.5.7 yatr.main module

yatr.main.**add_argument**(*parser*, **args*, ***kwargs*)

yatr.main.**cache_file**(*infile*, *outfile*, *cachedir*)

yatr.main.**compile_completion_data**(*path*, *cachedir*, *outpath*)

yatr.main.**data_path_from_yatrfile_path**(*path*, *cachedir*)

yatr.main.**default_data**()

yatr.main.**dump_bash_completions**(*args*, *idx*)

yatr.main.**expand_paths**(*opts*)

yatr.main.**find_bash_completions**(*args*, *idx*)

yatr.main.**find_yatrfile_path**(*path*)

yatr.main.**install_bash_completions**()

yatr.main.**load_completion_data**(*yatrfile*, *cachedir*)

yatr.main.**main**()

yatr.main.**matches**(*s*, *lst*)

yatr.main.**print_**(*s*, *flush=True*)

yatr.main.**render**(*doc*, *infile*, *outfile*)

yatr.main.**search_dir**(*path*)

yatr.main.**search_rootward**(*path*)

1.5.8 yatr.parse module

class yatr.parse.**Document**(**kwargs*)
Bases: syn.base.b.base.Base

Keyword-Only Arguments:

cachedir (*default* =): *basestring* Directory to store downloaded files

captures: *dict* (any => *basestring*) contexts: *dict* (any => *Context*) declares: *list* (*basestring*) default_task (*default* =): *basestring*

Task to run if no task is specified at the command line

dirname: *basestring* Relative path for includes

env: *Env* files: *dict* (any => *basestring*) imports: *list* (*basestring*) includes: *list* (*basestring*) macros: *dict* (any => *basestring* | *int* | *float* | *list* (*basestring* | *int* | *float* | *list* | *dict*) | *dict* (any => *basestring* | *int* | *float* | *list* | *dict*)) pull (*default* = False): *bool*

Force-pull URLs

secret_values: *dict* (any => *basestring*) secrets: *list* (*basestring*) settings: *dict* (any => any) tasks: *dict* (any => *Task*)

Class Options:

- args: ()
- autodoc: True
- coerce_args: False
- id_equality: False
- init_validate: True
- make_hashable: False
- make_type_object: True
- optional_none: False
- register_subclasses: False
- repr_template:
- coerce_hooks: ()
- create_hooks: ()
- init_hooks: ()
- init_order: ()
- metaclass_lookup: ('coerce_hooks', 'init_hooks', 'create_hooks', 'setstate_hooks')
- setstate_hooks: ()

Groups:

- _all: cachedir, captures, contexts, declares, default_task, dirname, env, files, imports, includes, macros, pull, secret_values, secrets, settings, tasks
- _internal: env

classmethod from_path(*path*, **kwargs*)

classmethod from_yaml(*dct*, *dirname*, ***kwargs*)

post_process(***kwargs*)

process(***kwargs*)

process_import(*path*, ***kwargs*)

process_include(*path*, ***kwargs*)

process_secret(*name*, ***kwargs*)

process_settings(***kwargs*)

run(*name*, ***kwargs*)

validate()
Raise an exception if the object is missing required attributes, or if the attributes are of an invalid type.

1.5.9 yatr.task module

class yatr.task.**For**(*var*, *in_*, ***kwargs*)
Bases: syn.base.b.base.Base

Positional Arguments:

var: ***basestring* | *list* (*basestring*)** The loop variable(s)

in_: *basestring* | *list* (*basestring* | *int* | *list*) Name(s) of list macro(s) to loop over

Class Options:

- args: ('var', 'in_')
- autodoc: True
- coerce_args: False
- id_equality: False
- init_validate: True
- make_hashable: False
- make_type_object: True
- optional_none: False
- register_subclasses: False
- repr_template:
- coerce_hooks: ()
- create_hooks: ()
- init_hooks: ()
- init_order: ()
- metaclass_lookup: ('coerce_hooks', 'init_hooks', 'create_hooks', 'setstate_hooks')
- setstate_hooks: ()

Groups:

- _all: in_, var

classmethod from_yaml(*dct*)

loop(*env*, ***kwargs*)

resolve_macros(*env*, ***kwargs*)

validate()
Raise an exception if the object is missing required attributes, or if the attributes are of an invalid type.

class yatr.task.**Command**(*command*, ***kwargs*)
Bases: syn.base.b.base.Base

Positional Arguments:

command: *basestring* | <callable>

Keyword-Only Arguments:

context (*default* =): *basestring*

Class Options:

- args: ('command',)
- autodoc: True
- coerce_args: False
- id_equality: False
- init_validate: True
- make_hashable: False
- make_type_object: True
- optional_none: False
- register_subclasses: False
- repr_template:
- coerce_hooks: ()
- create_hooks: ()
- init_hooks: ()
- init_order: ()
- metaclass_lookup: ('coerce_hooks', 'init_hooks', 'create_hooks', 'setstate_hooks')
- setstate_hooks: ()

Groups:

- _all: command, context

resolve_macros(*env*, ***kwargs*)

run(*env*, ***kwargs*)

class yatr.task.**Task**(***kwargs*)
Bases: syn.base.b.base.Base

Keyword-Only Arguments:

args: *list* (*basestring* | *int*) commands: *list* (*Command*) condition [**Optional**]: *Task* condition_type (*default* = True): *bool* kwargs: *dict* (any => *basestring* | *int*) loop [**Optional**]: *For*

Class Options:

- args: ()
- autodoc: True
- coerce_args: False
- id_equality: False
- init_validate: True
- make_hashable: False
- make_type_object: True
- optional_none: True
- register_subclasses: False
- repr_template:
- coerce_hooks: ()
- create_hooks: ()
- init_hooks: ()
- init_order: ()
- metaclass_lookup: ('coerce_hooks', 'init_hooks', 'create_hooks', 'setstate_hooks')
- setstate_hooks: ()

Groups:

- _all: args, commands, condition, condition_type, kwargs, loop

classmethod from_yaml(*name*, *dct*)

run(*env*, ***kwargs*)

run_preview(*env*, ***kwargs*)

1.5.10 Module contents

1.6 Changelog

1.6.1 0.0.11 (2018-xx-xx)

- Added [] read-only access to `env.Env`.
- Added validation support for dict macros indexed by runtime-defined macros.
- Added better error messages for validation failures.
- Added macro resolution for task `args` and `kwargs`.

1.6.2 0.0.10 (2018-03-28)

- Fixed issue where `-m`-defined macros didn't override `capture` macros.
- Fixed issue where ~ and environment variables are not expanded in some paths.
- Added `declare` section, and fixed issue of yatrfiles with runtime-defined macros not validating.

1.6.3 0.0.9 (2018-02-03)

- Added Env decorators.
- Added anonymous task definitions.
- Added support for importing extension modules with Python `import` statement-style names.

1.6.4 0.0.8 (2017-11-28)

- Added `--cache`.
- Added dictionary macros.

1.6.5 0.0.7 (2017-11-15)

- Added `files` section and functionality.
- Refactored `Command.run()`; removed `Task.run_commands()` and `Command.run_command()`.
- Added arguments for calling tasks.
- Added more efficient `yatr` calls within yatrfiles.
- Added some builtin macros.

1.6.6 0.0.6 (2017-11-13)

- Fixed default task specification not inheriting.
- Added for loop tasks and list macros.
- Added support for custom Jinja2 functions and filters.

1.6.7 0.0.5 (2017-11-02)

- Fixed issue of capture command execution directory.
- Added default task section.
- Added `--render`.

1.6.8 0.0.4 (2017-11-01)

- Added bash tab completions.

1.6.9 0.0.3 (2017-10-29)

- Fixed process management for running tasks.
- Added `-m` option.
- Added `--cache-dir` option.
- Added `-p` and `-v` options.

- Added support for macros and task references in `if` and `ifnot` keys.
- Added `capture` section and functionality.
- Added `settings` section.

1.6.10 0.0.2 (2017-10-26)

- Added conditional task execution.
- Added URL support for includes and imports.
- Added support for macros in includes and imports.
- Added exit with task error return code.
- Added task referencing in task definition.

1.6.11 0.0.1 (2017-10-18)

Initial release.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

y

yatr, 28
yatr.base, 21
yatr.context, 21
yatr.env, 22
yatr.env_decorators, 24
yatr.main, 24
yatr.parse, 25
yatr.task, 26
yatr.tests, 21
yatr.tests.mod1, 20
yatr.tests.mod2, 20

Index

A

add_argument() (in module yatr.main), 24

C

cache_file() (in module yatr.main), 24
capture_value() (yatr.env.Env method), 24
Command (class in yatr.task), 27
compile_completion_data() (in module yatr.main), 24
Context (class in yatr.context), 21
context_name (yatr.context.Context attribute), 22
context_name (yatr.tests.mod2.Foo attribute), 21

D

data_path_from_yatrfile_path() (in module yatr.main), 24
default_data() (in module yatr.main), 24
Document (class in yatr.parse), 25
dump_bash_completions() (in module yatr.main), 24

E

Env (class in yatr.env), 22
expand_paths() (in module yatr.main), 24

F

find_bash_completions() (in module yatr.main), 24
find_yatrfile_path() (in module yatr.main), 24
Foo (class in yatr.tests.mod2), 20
For (class in yatr.task), 26
from_path() (yatr.parse.Document class method), 25
from_yaml() (yatr.context.Context class method), 22
from_yaml() (yatr.parse.Document class method), 25
from_yaml() (yatr.task.For class method), 26
from_yaml() (yatr.task.Task class method), 28

I

install_bash_completions() (in module yatr.main), 24

J

jinja_filter() (yatr.env.Env method), 24
jinja_function() (yatr.env.Env method), 24

L

load_completion_data() (in module yatr.main), 24
loop() (yatr.task.For method), 27

M

macro_env() (yatr.env.Env method), 24
main() (in module yatr.main), 24
matches() (in module yatr.main), 24

P

post_process() (yatr.parse.Document method), 26
print_() (in module yatr.main), 24
process() (yatr.parse.Document method), 26
process_import() (yatr.parse.Document method), 26
process_include() (yatr.parse.Document method), 26
process_secret() (yatr.parse.Document method), 26
process_settings() (yatr.parse.Document method), 26

R

render() (in module yatr.main), 24
required_opts (yatr.context.Context attribute), 22
resolve() (yatr.env.Env method), 24
resolve_arg_macros() (yatr.env.Env method), 24
resolve_macros() (yatr.context.Context method), 22
resolve_macros() (yatr.env.Env method), 24
resolve_macros() (yatr.task.Command method), 27
resolve_macros() (yatr.task.For method), 27
run (yatr.context.Context attribute), 22
run() (yatr.parse.Document method), 26
run() (yatr.task.Command method), 27
run() (yatr.task.Task method), 28
run_command() (yatr.context.Context method), 22
run_preview() (yatr.task.Task method), 28

S

search_dir() (in module yatr.main), 24
search_rootward() (in module yatr.main), 24

T

Task (class in yatr.task), 27
task() (yatr.env.Env method), 24

V

validate() (yatr.context.Context method), 22
validate() (yatr.env.Env method), 24
validate() (yatr.parse.Document method), 26
validate() (yatr.task.For method), 27
ValidationError, 21
verbose() (yatr.context.Context method), 22

Y

yatr (module), 28
yatr.base (module), 21
yatr.context (module), 21
yatr.env (module), 22
yatr.env_decorators (module), 24
yatr.main (module), 24
yatr.parse (module), 25
yatr.task (module), 26
yatr.tests (module), 21
yatr.tests.mod1 (module), 20
yatr.tests.mod2 (module), 20