

---

# PyBuilder Documentation

*Release 0.10*



**PyBuilder Team**



**May 25, 2018**



# CONTENTS



<b>1</b>	<b>Installation</b>	<b>1</b>
1.1	Virtual Environment . . . . .	1
1.2	Installing completions . . . . .	1
<b>2</b>	<b>Concepts</b>	<b>3</b>
2.1	Introduction . . . . .	3
2.2	Advantages for python projects . . . . .	3
2.3	Why Another Build Tool . . . . .	3
2.4	Design . . . . .	4
<b>3</b>	<b>Complete walkthrough for a new PyBuilder project</b>	<b>5</b>
3.1	Installing PyBuilder . . . . .	5
3.2	Scaffolding . . . . .	5
3.3	Our new <code>build.py</code> . . . . .	6
3.4	Our first test . . . . .	7
3.5	Adding a script . . . . .	9
<b>4</b>	<b>Walkthrough working on an existing PyBuilder project</b>	<b>13</b>
4.1	Getting the project . . . . .	13
4.2	Ensuring your environment is ready . . . . .	13
4.3	Building the project . . . . .	13
<b>5</b>	<b>The <code>build.py</code> project descriptor</b>	<b>17</b>
5.1	The <code>build.py</code> anatomy . . . . .	17
5.2	Initializers . . . . .	18
5.3	Tasks . . . . .	20
<b>6</b>	<b>Packaging your project</b>	<b>23</b>
6.1	Dealing with project dependencies . . . . .	23
6.2	Packaging with <code>setuptools</code> . . . . .	23
6.3	The <code>copy_resources</code> plugin . . . . .	23
6.4	Installing non-python files . . . . .	23
6.5	Writing and shipping a <code>setup.cfg</code> . . . . .	23
6.6	Replacing placeholders before packaging - the <code>filter_resources</code> plugin . . . . .	23
<b>7</b>	<b>Plugins for testing</b>	<b>25</b>
7.1	Running python unit tests . . . . .	25
7.2	Running python integration tests . . . . .	25
7.3	The <code>cram</code> commandline test plugin . . . . .	25
7.4	Monitoring test status with the <code>pytdmon</code> plugin . . . . .	25
7.5	Running tests with arbitrary shell commands ( <code>pytest, ...</code> ) . . . . .	25

<b>8</b>	<b>Plugins for code quality</b>	<b>27</b>
8.1	Measuring coverage . . . . .	27
8.2	Integrating with SonarQube . . . . .	27
8.3	Linting python sources . . . . .	27
<b>9</b>	<b>IDE integration</b>	<b>29</b>
9.1	IntelliJ IDEA / PyCharm . . . . .	29
9.2	Eclipse PyDev . . . . .	29
9.3	Sublime Text 3 . . . . .	29
<b>10</b>	<b>Extending PyBuilder</b>	<b>31</b>



## INSTALLATION

PyBuilder is available on PyPI, so you can install it with

```
$ pip install pybuilder
```

### 1.1 Virtual Environment

We recommend installing PyBuilder into a [virtual environment](#) using `pip`:

```
$ virtualenv venv
```

---

**Note:** At first it might seem tempting to install PyBuilder system-wide with `sudo pip install pybuilder`, but if you work with virtualenvs then PyBuilder will see your system python (due to being installed there) instead of the virtualenv python.

---

### 1.2 Installing completions

If you are a `zsh` or `fish` shell user, we recommend installing the [pybuilder-completions](#). These will provide tab-based completions for PyBuilder options and tasks on a per-project basis.

```
sudo pip install pybuilder-completions
```

---

**Note:** The completions can be installed system-wide since they are just files for the relevant shells.

---



## CONCEPTS

### 2.1 Introduction



PyBuilder is a multi-purpose software build tool. Most commonly it targets the building and management of software with a strong focus on Python.

### 2.2 Advantages for python projects



**Some of the capabilities provided by PyBuilder out-of-the box are:**

- Automatic execution of unit and integration tests on every build
- Automatic analysis of the code coverage
- Automatic execution and result interpretation of analysis tools, such as flake8
- Automatic generation of distutils script `setup.py`

The general idea is that everything you do in your continuous integration chain, you also do locally before checking in your work.

### 2.3 Why Another Build Tool

When working on large scale software projects based on Java and Groovy I delved into the build process using tools such as Apache Ant, Apache Maven or Gradle. Although none of these tools is perfect they all provide a powerful and extensible way for building and testing software.

When focusing on Python I looked for a similar tool and got frustrated by the large number of tools that all match some aspect of the build and test process. Unfortunately, many of those tools were not suitable for composition and there was no central point of entry.

I suddenly found myself writing “build scripts” in Python over and over again using the tools I found out to be useful.


**PyBuilder was born on the attempt to create a reusable tool that should:**


- Make simple things simple
- Make hard things as simple as possible
- Let me use whatever tool I want to integrate
- Integrate these tools into a common view
- Let me use Python (which is really great) to write my build files

## 2.4 Design

PyBuilder executes build logic that is organized into tasks and actions.

Tasks are the main building blocks of the build logic. A task is an enclosed piece of build logic to be executed as a single unit. Each task can name a set of other tasks that it depends on. PyBuilder ensures that a task gets executed only after all of its dependencies have been executed.

Actions are smaller pieces of build logic than tasks. They are bound to  execution of task. Each action states that it needs to be executed before or after a named task. PyBuilder will execute the action if and only if the named task is executed, either directly or through another tasks' dependencies.

Actions as well as tasks are decorated plain  on functions. Thus, you can structure your code the way you like if you provide a single point of entry to a build step.

Both task and action functions can request parameters known to PyBuilder through dependency injection by parameter name.



## COMPLETE WALKTHROUGH FOR A NEW PYBUILDER PROJECT

### 3.1 Installing PyBuilder

We'll start by creating a folder for our new project:

```
mkdir myproject
cd myproject
```

Then, onto creating a [virtualenv](#) and install PyBuilder inside it:

```
virtualenv venv
source venv/bin/activate
pip install pybuilder
```

### 3.2 Scaffolding

Now we can use PyBuilder's own scaffolding capabilities:

```
(venv) mriehl@isdeblnnl084 myproject $ pyb --start-project
Project name (default: 'myproject') :
Source directory (default: 'src/main/python') :
Docs directory (default: 'docs') :
Unittest directory (default: 'src/unittest/python') :
Scripts directory (default: 'src/main/scripts') :
Use plugin python.flake8 (Y/n)? (default: 'y') :
Use plugin python.coverage (Y/n)? (default: 'y') :
Use plugin python.distutils (Y/n)? (default: 'y') :
```

As you can see, this created the content roots automatically:

```
(venv) mriehl@isdeblnnl084 myproject $ ll --
inode Permissions Size Blocks User  Group Date Modified Name
1488 drwxr-xr-x    -      - mriehl admins 28 Jul 17:46 .
1521 .rw-r--r--   324      8 mriehl admins 28 Jul 17:46 | build.py
2844 drwxr-xr-x    -      - mriehl admins 28 Jul 17:46 | docs
2143 drwxr-xr-x    -      - mriehl admins 28 Jul 17:46 | src
2789 drwxr-xr-x    -      - mriehl admins 28 Jul 17:46 |   main
2803 drwxr-xr-x    -      - mriehl admins 28 Jul 17:46 |     | python
2864 drwxr-xr-x    -      - mriehl admins 28 Jul 17:46 |     | scripts
2827 drwxr-xr-x    -      - mriehl admins 28 Jul 17:46 |     | unittest
2829 drwxr-xr-x    -      - mriehl admins 28 Jul 17:46 |     | python
```

### 3.3 Our new build.py

Let us now take a look at the build.py which is the centralized project description for our new project. ~~The annotated contents are:~~

```
from pybuilder.core import use_plugin, init

# These are the plugins we want to use in our project.
# Projects provide tasks which are blocks of logic executed by PyBuilder.

use_plugin("python.core")
# the python unittest plugin allows running python's standard library unittests
use_plugin("python.unittest")
# this plugin allows installing project dependencies with pip
use_plugin("python.install_dependencies")
# a linter plugin that runs flake8 (pyflakes + pep8) on our project sources
use_plugin("python.flake8")
# a plugin that measures unit test statement coverage
use_plugin("python.coverage")
# for packaging purposes since we'll build a tarball
use_plugin("python.distutils")

# The project name
name = "myproject"
# What PyBuilder should run when no tasks are given.
# Calling "pyb" amounts to calling "pyb publish" here.
# We could run several tasks by assigning a list to `default_task`.
default_task = "publish"

# This is an initializer, a block of logic that runs before the project is built.
@init
def set_properties(project):
    # Nothing happens here yet, but notice the `project` argument which is
    # automatically injected.
    pass
```

Let's run PyBuilder and see what happens:

```
(venv) mriehl@isdeblnn1084 myproject $ pyb
PyBuilder version 0.10.63
Build started at 2015-07-28 17:55:53

-----
[INFO] Building myproject version 1.0.dev0
[INFO] Executing build in /tmp/myproject
[INFO] Going to execute task publish
[INFO] Running unit tests
[INFO] Executing unit tests from Python modules in /tmp/myproject/src/unittest/python
[WARN] No unit tests executed.
[INFO] All unit tests passed.
[INFO] Building distribution in /tmp/myproject/target/dist/myproject-1.0.dev0
[INFO] Copying scripts to /tmp/myproject/target/dist/myproject-1.0.dev0/scripts
[INFO] Writing setup.py as /tmp/myproject/target/dist/myproject-1.0.dev0/setup.py
[INFO] Collecting coverage information
[INFO] Running unit tests
[INFO] Executing unit tests from Python modules in /tmp/myproject/src/unittest/python
```

(continues on next page)

(continued from previous page)

```
[WARN] No unit tests executed.
[INFO] All unit tests passed.
[WARN] Overall coverage is below 70%: 0%
Coverage.py warning: No data was collected.

-----
BUILD FAILED - Test coverage for at least one module is below 70%
-----

Build finished at 2015-07-28 17:55:54
Build took 0 seconds (515 ms)
```

We don't have any tests so our coverage is zero percent, all right! We have two ways to go about this - coverage breaks the build by default, so we can (if we want to) choose to not break the build based on the coverage metrics. This logic belongs to the project build, so we would have to add it to our `build.py` in the initializer. You can think of the initializer as a function that sets some configuration values before PyBuilder moves on to the actual work:

```
# This is an initializer, a block of logic that runs before the project is built.
@init
def set_properties(project):
    project.set_property("coverage_break_build", False) # default is True
```

With the above modification, the coverage plugin still complains but it does not break the build. Since we're clean coders, we're going to add some production code with a test though!

## 3.4 Our first test

We'll write an application that outputs "Hello world". Let's start with a test at `src/unittest/python/myproject_tests.py`:

```
from unittest import TestCase

from mock import Mock

from myproject import greet

class Test(TestCase):

    def test_should_write_hello_world(self):
        mock_stdout = Mock()

        greet(mock_stdout)

        mock_stdout.write.assert_called_with("Hello world!\n")
```


**Note:** As a default, the unittest plugin finds tests if their filename ends with `_tests.py`. We could change this with a well-placed `project.set_property` of course.

### 3.4.1 Our first dependency

Since we're using mock, we'll have to install it by telling our initializer in `build.py` about it:

```
# This is an initializer, a block of logic that runs before the project is built.
@init
def set_properties(project):
    project.set_property("coverage_break_build", False) # default is True
    project.build_depends_on("mock")
```

We could require a specific version and so on but let's keep it simple. Also note that we declared mock as a build dependency - this means it's only required for building and if we upload our project to PyPI then installing it from there will not require installing mock.

We can install our dependency by running PyBuilder  the corresponding task:

```
(venv) mriehl@isdeblnnl084 myproject $ pyb install_dependencies
PyBuilder version 0.10.63
Build started at 2015-07-28 19:35:37
-----
[INFO] Building myproject version 1.0.dev0
[INFO] Executing build in /tmp/myproject
[INFO] Going to execute task install_dependencies
[INFO] Installing all dependencies
[INFO] Installing build dependencies
[INFO] Installing dependency 'coverage'
[INFO] Installing dependency 'flake8'
[INFO] Installing dependency 'mock'
[INFO] Installing runtime dependencies
-----
BUILD SUCCESSFUL
-----
Build Summary
    Project: myproject
    Version: 1.0.dev0
    Base directory: /tmp/myproject
    Environments:
        Tasks: install_dependencies [1480 ms]
Build finished at 2015-07-28 19:35:39
Build took 1 seconds (1486 ms)
pyb install_dependencies 1.44s user 0.10s system 98% cpu 1.570 total
```

### 3.4.2 Running our test

We can run our test now:

```
(venv) mriehl@isdeblnnl084 myproject $ pyb verify
PyBuilder version 0.10.63
Build started at 2015-07-28 19:36:41
-----
[INFO] Building myproject version 1.0.dev0
[INFO] Executing build in /tmp/myproject
[INFO] Going to execute task verify
[INFO] Running unit tests
[INFO] Executing unit tests from Python modules in /tmp/myproject/src/unittest/python
[ERROR] Import error in test file /tmp/myproject/src/unittest/python/myproject_tests.
→py, due to statement 'from myproject import greet' on line 5
[ERROR] Error importing unittest: No module named myproject
-----
BUILD FAILED - Unable to execute unit tests.
```

(continues on next page)

(continued from previous page)

```
-----
Build finished at 2015-07-28 19:36:41
Build took 0 seconds (249 ms)
```

It's still failing because we haven't implemented anything yet. Let's do that right now in `src/main/python/myproject/__init__.py`:

```
def greet(filelike):
    filelike.write("Hello world!\n")
```

Any finally rerun the test:

```
(venv) mriehl@isdeblnnl084 myproject $ pyb verify
PyBuilder version 0.13
Build started at 2015-07-28 19:39:15
-----
[INFO] Building myproject version 1.0.dev0
[INFO] Executing build in /tmp/myproject
[INFO] Going to execute task verify
[INFO] Running unit tests
[INFO] Executing unit tests from Python modules in /tmp/myproject/src/unittest/python
[INFO] Executed 1 unit tests
[INFO] All unit tests passed.
[INFO] Building distribution in /tmp/myproject/target/dist/myproject-1.0.dev0
[INFO] Copying scripts to /tmp/myproject/target/dist/myproject-1.0.dev0/scripts
[INFO] Writing setup.py as /tmp/myproject/target/dist/myproject-1.0.dev0/setup.py
[INFO] Collecting coverage information
[INFO] Running unit tests
[INFO] Executing unit tests from Python modules in /tmp/myproject/src/unittest/python
[INFO] Executed 1 unit tests
[INFO] All unit tests passed.
[INFO] Overall coverage is 100%
-----
BUILD SUCCESSFUL
-----
Build Summary
    Project: myproject
    Version: 1.0.dev0
    Base directory: /tmp/myproject
    Environments:
        Tasks: prepare [231 ms] compile_sources [0 ms] run_unit_tests [10 ms]
        ↳ package [1 ms] run_integration_tests [0 ms] verify [255 ms]
Build finished at 2015-07-28 19:39:15
Build took 0 seconds (504 ms)
```

### 3.5 Adding a script

Since our library is ready, we can now add a script.

We'll just need to create `src/main/scripts/greeter`:

```
#!/usr/bin/env python
import sys
from myproject import greet
```

(continues on next page)

(continued from previous page)

```
greet(sys.stdout)
```

Note that there is nothing else to do. Dropping the file in `src/main/scripts` is all we need to do for PyBuilder to pick it up, because this is the convention.

Let's look at what happens when we package it up:

```
(venv) mriehl@isdeblnn1084 myproject $ pyb publish
PyBuilder version 0.10.63
Build started at 2015-07-28 19:44:34
-----
[INFO] Building myproject version 1.0.dev0
[INFO] Executing build in /tmp/myproject
[INFO] Going to execute task publish
[INFO] Running unit tests
[INFO] Executing unit tests from Python modules in /tmp/myproject/src/unittest/python
[INFO] Executed 1 unit tests
[INFO] All unit tests passed.
[INFO] Building distribution in /tmp/myproject/target/dist/myproject-1.0.dev0
[INFO] Copying scripts to /tmp/myproject/target/dist/myproject-1.0.dev0/scripts
[INFO] Writing setup.py as /tmp/myproject/target/dist/myproject-1.0.dev0/setup.py
[INFO] Collecting coverage information
[INFO] Running unit tests
[INFO] Executing unit tests from Python modules in /tmp/myproject/src/unittest/python
[INFO] Executed 1 unit tests
[INFO] All unit tests passed.
[INFO] Overall coverage is 100%
[INFO] Building binary distribution in /tmp/myproject/target/dist/myproject-1.0.dev0
-----
BUILD SUCCESSFUL
-----
Build Summary
    Project: myproject
    Version: 1.0.dev0
    Base directory: /tmp/myproject
    Environments:
        Tasks: prepare [227 ms] compile_sources [0 ms] run_unit_tests [9 ms]
↳ package [2 ms] run_integration_tests [0 ms] verify [252 ms] publish [241 ms]
Build finished at 2015-07-28 19:44:35
Build took 0 seconds (739 ms)
```

We can now simply `pip install` the tarball:

```
(venv) mriehl@isdeblnn1084 myproject $ pip install target/dist/myproject-1.0.dev0/
↳ dist/myproject-1.0.dev0.tar.gz
Processing ./target/dist/myproject-1.0.dev0/dist/myproject-1.0.dev0.tar.gz
Building wheels for collected packages: myproject
  Running setup.py bdist_wheel for myproject
    Stored in directory: /data/home/mriehl/.cache/pip/wheels/89/05/9e/
↳ 4b035292abf39e5d6ddcf442cc7c96c2e56f5cc49c5c673d3a
Successfully built myproject
Installing collected packages: myproject
Successfully installed myproject-1.0.dev0
(venv) mriehl@isdeblnn1084 myproject $ gr
Hello world!
```

Of course since there is a `setup.py` in the distribution folder, we can use it to do whatever we want easily, for

example uploading to PyPI:

```
(venv) mriehl@isdeblnnl084 myproject $ cd target/dist/myproject-1.0.dev0/  
(venv) mriehl@isdeblnnl084 myproject-1.0.dev0 $ python setup.py upload
```





## WALKTHROUGH WORKING ON AN EXISTING PYBUILDER PROJECT

### 4.1 Getting the project

We'll use `yadtshell` as an example to checkout and build a PyBuilder project from scratch.

Begin by cloning the git project:

```
git clone https://github.com/yadt/yadtshell
```

and then move into the new directory:

```
cd yadtshell
```

### 4.2 Ensuring your environment is ready

Please make sure you have `virtualenv` installed. You will need it to isolate the `yadtshell` dependencies from your system python.

Now go ahead and create a new `virtualenv` (we'll name it `venv`):

```
virtualenv venv
```

You can now make the `virtualenv` active by sourcing its activate script:

```
source venv/bin/activate
```

Now install `PyBuilder` in your new `virtualenv`:

```
pip install pybuilder
```

### 4.3 Building the project

We're finally ready to build our project. You can start by asking `PyBuilder` what tasks it knows about:

```
(venv) mriehl@isdeblnnl084 yadtshell (git)-[master] $ pyb -t
Tasks found for project "yadtshell":
    analyze - Execute analysis plugins.
              depends on tasks: run_unit_tests prepare prepare
    clean - Cleans the generated output.
    compile_sources - Compiles source files that need compilation.
                   depends on tasks: prepare
    generate_manpage_with_pandoc - <no description available>
    install_build_dependencies - Installs all build dependencies specified in the
    build_descriptor
```

(continues on next page)

(continued from previous page)

```

        install_dependencies - Installs all (both runtime and build) dependencies
        ↳ specified in the build descriptor
        install_runtime_dependencies - Installs all runtime dependencies specified in the
        ↳ build descriptor
        list_dependencies - Displays all dependencies the project requires
        package - Packages the application. Package a python
        ↳ application.

        depends on tasks: run_unit_tests
        prepare - Prepares the project for building.
        publish - Publishes the project.
        depends on tasks: verify
        run_integration_tests - Runs integration tests on the packaged application.
        ↳ Runs integration tests based on Python's unittest module
        depends on tasks: package
        run_sonar_analysis - Launches sonar-runner for analysis.
        depends on tasks: analyze
        run_unit_tests - Runs all unit tests. Runs unit tests based on
        ↳ Python's unittest module
        depends on tasks: compile_sources
        verify - Verifies the project and possibly integration
        ↳ tests.
        depends on tasks: run_integration_tests

```

You can call any of these tasks (PyBuilder will ensure dependencies are satisfied):

```
pyb clean analyze
```

The most obvious task to start with is installing the project dependencies - PyBuilder makes a distinction between run-time (a hipster async library for example) and build-time dependencies (a test framework). You can simply install all dependencies with:

```
pyb install_dependencies
```

There are also other tasks (`install_build_dependencies` and `install_runtime_dependencies`) for more fine-grained control.

PyBuilder comes with a “default goal” so that building is easy - per convention this means that when not given any tasks, PyBuilder will perform all the tasks deemed useful by the project developers. So in general, building your project is as simple as:

```

(venv) mriehl@isdeblnn1084 yadtshell (git)-[master] $ pyb
PyBuilder version 0.10.63
Build started at 2015-07-28 09:50:00
-----
[INFO] Building yadtshell version 1.9.2
[INFO] Executing build in /data/home/mriehl/workspace/yadtshell
[INFO] Going to execute tasks: clean, analyze, publish
[INFO] Removing target directory /data/home/mriehl/workspace/yadtshell/target
[INFO] Removing yadtshell log directory: /tmp/logs/yadtshell/2015-07-28
[INFO] Removing yadtshell integration test stubs directory: /tmp/yadtshell-it
[INFO] Removing yadtshell state directory: /data/home/mriehl/.yadtshell
[INFO] Executing unittest Python modules in /data/home/mriehl/workspace/yadtshell/
↳ src/unittest/python
[INFO] Executed 289 unittests
[INFO] All unittests passed.
[INFO] Executing flake8 on project sources.

```

(continues on next page)

(continued from previous page)

```

[INFO] Executing frosted on project sources.
[INFO] Collecting coverage information
[INFO] Executing unittest Python modules in /data/home/mriehl/workspace/yadtshell/
↳src/unittest/python
[INFO] Executed 289 unittests
[INFO] All unittests passed.
[WARN] Test coverage below 50% for yadtshell.restart: 29%
[WARN] Test coverage below 50% for yadtshell.actionmanager: 40%
[WARN] Test coverage below 50% for yadtshell.dump: 17%
[WARN] Module not imported: yadtshell.broadcast. No coverage information available.
[WARN] Test coverage below 50% for yadtshell.TerminalController: 42%
[INFO] Overall coverage is 65%
[INFO] Building distribution in /data/home/mriehl/workspace/yadtshell/target/dist/
↳yadtshell-1.9.2
[INFO] Copying scripts to /data/home/mriehl/workspace/yadtshell/target/dist/
↳yadtshell-1.9.2/scripts
[INFO] Copying resources matching 'setup.cfg docs/man/yadtshell.1.man.gz' from /data/
↳home/mriehl/workspace/yadtshell to /data/home/mriehl/workspace/yadtshell/target/
↳dist/yadtshell-1.9.2
[INFO] Filter resources matching **/yadtshell/__init__.py **/scripts/yadtshell **/
↳setup.cfg in /data/home/mriehl/workspace/yadtshell/target
[WARN] Skipping impossible substitution for 'GREEN' - there is no matching project_
↳attribute or property.
[WARN] Skipping impossible substitution for 'BOLD' - there is no matching project_
↳attribute or property.
[WARN] Skipping impossible substitution for 'NORMAL' - there is no matching project_
↳attribute or property.
[WARN] Skipping impossible substitution for 'BG_YELLOW' - there is no matching_
↳project attribute or property.
[WARN] Skipping impossible substitution for 'BOLD' - there is no matching project_
↳attribute or property.
[WARN] Skipping impossible substitution for 'NORMAL' - there is no matching project_
↳attribute or property.
[WARN] Skipping impossible substitution for 'RED' - there is no matching project_
↳attribute or property.
[WARN] Skipping impossible substitution for 'BOLD' - there is no matching project_
↳attribute or property.
[WARN] Skipping impossible substitution for 'NORMAL' - there is no matching project_
↳attribute or property.
[INFO] Writing MANIFEST.in as /data/home/mriehl/workspace/yadtshell/target/dist/
↳yadtshell-1.9.2/MANIFEST.in
[INFO] Writing setup.py as /data/home/mriehl/workspace/yadtshell/target/dist/
↳yadtshell-1.9.2/setup.py
[INFO] Running integration tests in parallel
[-----]
[INFO] Executed 68 integration tests.
[INFO] Building binary distribution in /data/home/mriehl/workspace/yadtshell/target/
↳dist/yadtshell-1.9.2
-----
BUILD SUCCESSFUL
-----
Build Summary
    Project: yadtshell
    Version: 1.9.2
    Base directory: /data/home/mriehl/workspace/yadtshell
    Environments:
        Tasks: clean [30 ms] prepare [282 ms] compile_sources [0 ms] run_unit_
↳tests [951 ms] analyze [2679 ms] package [26 ms] run_integration_tests [2695 ms]
↳verify [0 ms] publish [523 ms]

```

(continues on next page)

(continued from previous page)

```
Build finished at 2015-07-28 09:50:31
Build took 30 seconds (30812 ms)
pyb 42.96s user 8.21s system 165% cpu 30.918 total
```

## THE BUILD.PY PROJECT DESCRIPTOR

### 5.1 The build.py anatomy

A `build.py` project descriptor consists of several parts:

#### 5.1.1 Imports

It's python code after all, so PyBuilder functions we use must be imported first.

```
import os
from pybuilder.core import task, init, use_plugin, Author
```

#### 5.1.2 Plugin imports

Through usage of the `use_plugin` function, a plugin is loaded (its initializers are registered for execution and any tasks are added to the available tasks).

```
use_plugin("python.core")
use_plugin("python.pycharm")
```

#### 5.1.3 Project fields

Assigning to variables in the top-level of the `build.py` will set the corresponding fields on the `project` object. Some of these fields are standardized (like `authors`, `name` and `version`).

```
authors = [Author('John Doe', 'john@doe.invalid'),
           Author('Jane Doe', 'jane@doe.invalid')]

description = "This is the best project ever!"
name = 'myproject'
license = 'GNU GPL v3'
version = '0.0.1'

default_task = ['clean', 'analyze', 'publish']
```

Note that the above is equivalent to setting all the fields on `project` in an initializer, though the first variant above is preferred for brevity's sake.

```
@init
def initialize(project):
    project.name = 'myproject'
    ...
```

## 5.2 Initializers

An initializer is a function decorated by the `@init` decorator. It is automatically collected by PyBuilder and executed before actual tasks are executed. The main use case of initializers is to mutate the `project` object in order to configure plugins.

### 5.2.1 Dependency injection

PyBuilder will automatically inject arguments by name into an initializer (provided the initializer accepts it). The arguments `project` and `logger` are available currently.

This means that all of these are fine and will work as expected:

```
@init
def initialize():
    pass

@init
def initialize2(logger):
    pass

@init
def initialize3(project, logger):
    pass

@init
def initialize3(logger, project):
    pass
```

### 5.2.2 Environments

It's possible to execute an initializer only when a specific command-line switch was passed. This is a bit akin to Maven's profiles:

```
@init(environments='myenv')
def initialize():
    pass
```

The above initializer will only get executed if we call `pyb` with the `-E myenv` switch.

### 5.2.3 The project object

The project object is used to describe the project and plugin settings. It also provides useful functions we can use to implement build logic.

## Setting properties

PyBuilder uses a key-value based configuration for plugins. In order to set configuration, `project.set_property(name, value)` is used. For example we can tell the flake8 plugin to also lint our test sources with:

```
project.set_property('flake8_include_test_sources', True)
```

In some cases we just want to mutate the properties (for example adding an element to a list), this can be achieved with `project.get_property(name)`. For example we can tell the filter\_resources plugin to apply on all files named `setup.cfg`:

```
project.get_property('filter_resources_glob').append('*/setup.cfg')
```

Note that `append` mutates the list.

## Project dependencies

The project object tracks our project's dependencies. There are several variants to add dependencies:

- `project.depends_on(name)` (runtime dependency)
- `project.build_depends_on(name)` (build-time dependency)
- `project.depends_on(name, version)` (where version is a pip version string like `'==1.1.0'` or `'>=1.0'`)
- `project.build_depends_on(name, version)` (where version is a pip version string like `'==1.1.0'`)

This will result on the `install_dependencies` plugin installing these dependencies when its task is called. Runtime dependencies will also be added as metadata when packaging the project, for example building a python `setuptools` tarball with a `setup.py` will fill the `install_requires` list.

## Installing files

Installing non-python files is easily done with `project.install_file(target, source)`. The target path may be absolute, or relative to the installation prefix (`/usr/` on most linux systems).

As an important sidenote, the path to `source` *must* be relative to the distribution directory. Since non-python files are not copied to the distribution directory by default, it is necessary to use the `copy_resources` plugin to include them.

Consider you want to install `src/main/resources/my-config.yaml` in `/etc/defaults`. It would be done like so:

First, we use `copy_resources` to copy the file into the distribution directory:

```
use_plugin("copy_resources")

@init
def initialize(project):
    project.get_property("copy_resources_glob").append("src/main/resources/my-config.
↪yaml")
    project.set_property("copy_resources_target", "$dir_dist")
```

Now, whenever `copy_resources` run, we will have the path `src/main/resources/my-config.yaml` copied into `target/dist/myproject-0.0.1/src/main/resources/my-config.yaml`. We're now able to do:

```
use_plugin("copy_resources")

@init
def initialize(project):
    project.get_property("copy_resources_glob").append("src/main/resources/my-config.
↪yaml")
    project.set_property("copy_resources_target", "$dir_dist")
    project.install_file("/etc/defaults", "src/main/resources/my-config.yaml")
```

---

**Note:** It's important to realize that the source path `src/main/resources/my-config.yaml` is **NOT** relative to the project root directory, but relative to the distribution directory instead. It just incidentally happens to be the same here.

---

## Including files

Simply use the `include_file` directive:

```
project.include_file(package_name, filename)
```

## 5.3 Tasks

### 5.3.1 Creating a task

To create a task, one can simply write a function in the `build.py` and annotate it with the `@task` decorator.

```
from pybuilder.core import task, init

@init
def initialize(project):
    pass

@task
def mytask(project, logger):
    logger.info("Hello from my task")
```

Like with initializer, PyBuilder will inject the arguments `project` and `logger` if the task function accepts them.

We'll now be able to call `pyb mytask`.

The project API can be used to get configuration properties (so that the task is configurable). It's also possible to compute paths by using `expand_path`:

```
from pybuilder.core import task

@task
def mytask(project, logger):
    logger.info("Will build the distribution in %s" % project.expand_path("$dir_dist
↪"))
```



### 5.3.2 Task dependencies

A task can declare dependencies on other tasks by using the `@depends` decorator:

```
from pybuilder.core import task, depends

@task
def task1(logger):
    logger.info("Hello from task1")

@task
@depends("task1")
def task2(logger):
    logger.info("Hello from task2")

@task
@depends("task2", "run_unit_tests")
def task3(logger):
    logger.info("Hello from task3")
```

Here, running task1 will just run task1. Running task2 will run task1 first, then task2. Running task3 will run task1 first (dependency of task2), then run task2, then run unit tests, and finally run task3.



## PACKAGING YOUR PROJECT

6.1 Dealing with project dependencies

6.2 Packaging with `setuptools`

6.3 The `copy_resources` plugin

6.4 Installing non-python files

6.5 Writing and shipping a `setup.cfg`

6.6 Replacing placeholders before packaging - the  
`filter_resources` plugin



## PLUGINS FOR TESTING

### 7.1 Running python unit tests

### 7.2 Running python integration tests

#### 7.2.1 Doing it in parallel

### 7.3 The cram commandline test plugin

### 7.4 Monitoring test status with the pytdmon plugin

### 7.5 Running tests with arbitrary shell commands (pytest, ...)



## PLUGINS FOR CODE QUALITY

### 8.1 Measuring coverage

### 8.2 Integrating with SonarQube

### 8.3 Linting python sources

#### 8.3.1 flake8

#### 8.3.2 pylint

#### 8.3.3 frosted

#### 8.3.4 pychecker

#### 8.3.5 pep8

#### 8.3.6 pymetrics





## IDE INTEGRATION

### 9.1 IntelliJ IDEA / PyCharm

### 9.2 Eclipse PyDev

### 9.3 Sublime Text 3



## EXTENDING PYBUILDER

...or try searching for what you want: search