Uranium Documentation

Release 0.1

Yusuke Tsutsumi

Jul 26, 2018

Contents

CHAPTER 1

What is Uranium?

Uranium is an assembly framework for Python, designed to help assist with the assembling Python services. Uranium provides tools for dependency management, reuse of assembly scripts, configuration, and other common requirements for an assembly system.

Uranium provides package isolation and management via virtualenv and pip, and is a good solution to problems that arise in large-scale assembly systems:

- setting a version pin across multiple projects.
- reusing common assembly tasks, such as downloading configuration
- providing a simple configuration system that can be consumed by multiple projects.

An example configuration looks like this:

```
import subprocess
# this is a uranium.py file
# it requires at the minimum a function
# main that accepts a parameter build.
def main(build):
    # you can change the index urls as desired.
    build.packages.index_urls = ["http://www.mycompany.com/index",
                                 "http://pypi.python.org"]
    # packages are installed using the packages.install method.
    build.packages.install("py.test")
    # once an egg is installed, you can run arbitrary scripts installed
    # into the sandbox:
    return subprocess.call(["py.test", "mytests"] + build.options.args)
```

Uranium is compatible with python2, python3, and pypy

Contents:

1.1 Installation

There are two ways one can run uranium:

- install it globally
- use the uranium script

1.1.1 Installing it globally

You can install uranium globally with any Python package manager:

```
pip install uranium
```

You would then enter a directory with a ubuild.py, and execute the uranium entry point:

```
uranium
```

Note: This approach is not ideal, as it enforces strict version restrictions of Uranium's dependencies onto your globally installed packages.

for most situations, it's suggested to use the uranium script.

1.1.2 Use the Uranium Script

The Uranium script can handle the installation and execution of uranium for you. There are two versions of the script:

- ./scripts/uranium_standalone, which downloads a local copy of uranium and executes it.
- ./scripts/uranium , a thin wrapper that downloads and executes the standalone version.

You would then execute the local uranium script instead:

```
./uranium
```

1.1.3 Which method should I use?

Utilizing the uranium_standalone or uranium script is recommended. The standalone scripts allow consumers to assemble your code without the need for any modification of their machine globally. It also allows for each individual project to choose their version of uranium, if that becomes necessary.

The uranium script requires trust in the uranium project and the developers, as it is downloads the standalone script from the git repository and executes that script.

The uranium script also provides a blueprint on how to provide your own bootstrapping script. This is recommended when setting up a structure for an organization, as a common standalone script ensures that changes can be applied globally with ease, in contrast to updating each uranium script in every location individually.

1.2 Tutorial

This tutorial is an introduction to the basic concepts around Uranium.

Let's start with a simple example: setting up a virtualenv and install an egg.

unix-based commands are used for the tutorial, but this documentation attempts to describe these steps so they can be easily replicated on other operating systems.

For the purpose of the tutorial, let's create a root directory:

```
$ mkdir -p /tmp/uranium-tut/ && cd /tmp/uranium-tut/
```

Start by downloading the uranium script. The uranium script is a python wrapper around the uranium library that handles the following:

- downloading and setting up a virtualenv
- installing the uranium script into the virtualenv
- running uranium for the first time.

You can get the uranium script here:

https://raw.githubusercontent.com/toumorokoshi/uranium/master/uranium/scripts/uranium

You should download a copy and add the script into the root directory:

```
$ curl -s https://raw.githubusercontent.com/toumorokoshi/uranium/master/uranium/
↪scripts/uranium > uranium
$ chmod +x uranium  # the script should be executable.
```

Now you need a ubuild.py file. Let's make one now:

```
$ touch ubuild.py
```

And we'll need to fill it in with at the very least, a main function:

```
def main(build):
    print("uranium works!")
```

Now, you can run uranium. Try it now:

```
$ ./uranium
installing virtualenv...
setting up uranium...
done!
[HH:MM:SS] ================
[HH:MM:SS] STARTING URANIUM
[HH:MM:SS] ================
[HH:MM:SS] uranium works!
[HH:MM:SS] ================
[HH:MM:SS] URANIUM FINISHED
[HH:MM:SS] ================
```

And congrats, you've had your first Uranium run! Uranium read the ubuild.py, found the main function, and executed it. However, the only result is having a set up virtualenv. The next step is working or real functionality.

1.2.1 Developing and Installing Eggs

We started with an empty main method. To add eggs and develop-eggs, you can use the packages attribute of build:

```
def main(build):
build.packages.install("nose", version="==1.3.4")
```

And let's run uranium again:

```
$ ./uranium
setting up uranium...
done!
[HH:MM:SS] ================
[HH:MM:SS] STARTING URANIUM
[HH:MM:SS] ================
[HH:MM:SS] installing eggs...
[HH:MM:SS] Adding requirement nose==1.3.4...
[HH:MM:SS] ================
[HH:MM:SS] URANIUM FINISHED
[HH:MM:SS] ================
```

If you want to install an egg for development purposes, you can use:

```
def main(build):
    build.packages.install(".", develop=True)
```

1.2.2 Executing Different Tasks

the ubuild.py can define other methods, and they can be executed as well. Any method that accepts a single parameter build can be a task that's executed:

```
import subprocess

# $ uranium
def main(build):
    print("this is the main method!")
    return 0

# $ uranium test
def test(build):
    build.packages.install("nose")

    # the return code is the integer returned
    # back.
    build.executables.run(["nose"])
```

1.3 The Build Object

The core functionality in Uranium is contained inside the build object. The build is an interface the environment that uranium is building: You can use the various attributes to manipulate it.

Examples include:

- build.packages to modify packages

- build.envvars to modify environment variables

And so on. For tasks, the build object is always passed in as the only argument:

```
def main(build):
    print(build.root)
```

1.3.1 uranium.current_build

There are situations where one needs to bootstrap a ubuild.py before executing a task, such as installing hooks or setting configuration.

In that situation, uranium.current_build works well: It is a proxy object that returns back whatever build object is currently executing:

```
from uranium import current_build
current_build.config.set_defaults({"debug": False})

def main(build):
    if build.config["debug"]:
        print("debug message")
```

1.3.2 Full API Reference

class uranium.build.**Build**(*root*, *config=None*, *with_sandbox=True*, *cache_requests=True*)

the build class is the object passed to the main method of the uranium script.

it's designed to serve as the public API to controlling the build process.

Build is designed to be executed within the sandbox itself. Attempting to execute this outside of the sandbox could lead to corruption of the python environment.

config

Returns a uranium.config.Config object

this is a generic dict to store / retrieve config data that tasks may find valuable

envvars

Returns a uranium.environment_variables.EnvironmentVariables object

this is an interface to the environment variables of the sandbox. variables modified here will be preserved when executing entry points in the sandbox.

executables

Returns uranium.executables.Executables

an interface to execute scripts

history

Returns uranium.history.History

a dictionary that can contain basic data structures, that is preserved across executions.

ideal for storing state, such as if a file was already downloaded.

hooks

Returns uranium.hooks.Hooks

provides hooks to attach functions to be executed during various phases of Uranium (like initializiation and finalization)

include (*script_path*, *cache=False*)
executes the script at the specified path.

options

Returns uranium.options.Options

an interface to arguments passed into the uranium command line.

packages

Returns uranium.packages.Packages

an interface to the python packages currently installed.

root

Returns str

returns the root of the uranium build.

task (*f*)
a decorator that adds the given function as a task.

e.g.

@build.task def main(build):

build.packages.install("httpretty")

this is useful in the case where tasks are being sourced from a different file, besides ubuild.py

tasks

Returns uranium.tasks.Tasks

an interface to the tasks that uranium has registered, or has discovered in the ubuild.py

1.4 More Examples

Uranium uses itself to build:

https://github.com/toumorokoshi/uranium/blob/master/ubuild.py

1.5 Cookbook

1.5.1 Best Practices

Using cache=True on include()

There are many cases where you would like to inherit a script from a remote source, but would also like to support offline execution of uranium (once dependencies are installed).

If cache=True is set for build.include, uranium will cache the script, thus allowing offline execution.

```
build.include("http://my-remote-base", cache=True)
```

1.5.2 Reusing Build Code

Uranium attempts to be as flexible as possible, so there are multiple patterns for reusing code in ubuild.py scripts. Choose the one that works for you.

build.includes

Uranium provides an includes function to download and execute a remote script. For example, let's say you want to share a common test function, as well as ensure builds are using a private repository. You can host a file uranium_base.py that looks like:

```
# http://internalgit.mycompany.com/shared-python/uranium_base.py
from uranium import current_build
import subprocess

build.packages.index_urls = [
    "https://pypi.python.org/",
    "https://internalpypi.mycompany.com/"
]

@current_build.task
def main(build):
    build.packages.install(".", develop=True)

@current_build.task
@uranium.requires("main")
def test(build):
    build.packages.install("pytest")
    build.packages.install("pytest-cov")
    subprocess.call(
        ["py.test", os.path.join(build.root, "tests")] + build.options.args
    )
```

And your consumer script will look like:

```
# ubuild.py in the project.
build.include("https://internalgit.mycompany.com/shared-python/uranium_base.py")
```

And you're done! One can modify the uranium_base.py, and apply those changes immediately.

Caveats

- Potentially insecure. https is recommended, as it verifies the authenticity of the page you're actually accessing.
- No builtin system for pinning yourself to older versions. You'll need to have every version of your uranium_base.py available publicly. This can be provided using a version control server that exposes files through an api.
- not easily testable.

using eggs and packages

The build.includes pattern works well, but it has some caveats, as explained above. As with distributing any code, it's better to utilize existing best practices.

Python's packaging infrastructure is already a great framework for reuse. Supply an package in your index repository that contains all the tasks, and download it in your ubuild.py.

```
# in a module mycompany_build
import subprocess
import uranium

def setup(build):
    build.packages.index_urls = [
        "http://pypi.python.org/",
        "http://internalpypi.mycompany.com/"
    ]

    @build.task
    def main(build):
        build.packages.install(".", develop=True)

    @build.task
    @uranium.requires("main")
    def test(build):
        main(build)
        build.packages.install("pytest")
        build.packages.install("pytest-cov")
        subprocess.call(
            ["py.test", os.path.join(build.root, "tests")] + build.options.args
        )
```

And your consumer script will look like:

```
# ubuild.py in the project.
from uranium import get_remote_script

# this is required, to consume internal packages.
build.packages.index_urls = [
    "http://pypi.python.org/",
    "http://internalpypi.mycompany.com/"
]
build.packages.install("mycompany-build")
import mycompany_build
mycompany_build.setup(build)
```

1.6 Configuration

Uranium provides infrastructure to pass in configuration variables.

Configuration variables are useful in a variety of situations, including:

- choose whether to run in development mode
- select the environment to run against

```
# config.set_defaults can be used to set some default values.
build.config.set_defaults({
    "development": "false"
})
```

(continues on next page)

(continued from previous page)

```
def test(build):
    # one can set development mode by adding a -c development=true before the task:
    # ./uranium -c development=true test
    if build.config["development"].startswith("t"):
        build.packages.install(".", develop=True)
```

1.6.1 Full API Reference

class uranium.config.**Config**

Config is a dictionary representing the configuration values passed into the Uranium build.

config acts as a dictionary, and should be accessed as such.

The current configuration is serialized to and from yaml, during the start and stop of uranium, respectively. As such, only primitive types such as arrays, dictionaries, strings, float, int, bool are supported.

The command line of uranium supports a dotted notation to modify nested values of the config object. To ensure that there is no ambiguity, it's best to keep all key names without any periods.

set_defaults(*default_dict*)

as a convenience for setting multiple defaults, set_defaults will set keys that are not yet set to the values from default_dict.

```
build.config.set_defaults({
    "environment": "develop"
})
```

1.7 Declaring Task Dependencies

1.7.1 Prepending Tasks

Uranium provides a declarative task dependency system through task_requires:

```
from uranium import task_requires

def main(build):
    print("main was")

# this ensures main is run first, during
# an execution.
@task_requires("main")
def test(build):
    print("test was run")

# a list can be passed in. In that case,
# each dependency is executed in the order
# it appears in the list.
#
# notice that a string with the task name,
# or the task itself can be passed in.
@task_requires(["main", test])
def build_docs(build):
    print("main was")
```

This relationship can be created after the fact by add_requires:

```
# test requires main
build.tasks.prepend("test", "main")
```

1.7.2 Executing Tasks After an Existing Task

```
# ensures test executes after main
build.tasks.append("main", "test")
```

1.8 Environment Variables

An environment variable set within uranium is active for not only the lifetime of the build, but for any entry points or scripts generated as well.

environment variables can be modified as a regular dictionary:

```
import os

def main(build):
    build.envvars["EDITOR"] = "emacs"
    build.envvars["LD_LIBRARY_PATH"] = os.path.join(build.root, "lib")
```

1.8.1 Full API Reference

class uranium.environment_variables.**EnvironmentVariables**

an interface exposed which allows the setting of environment variables.

it acts identical to a dictionary.

__getitem__(*key*)

retrieve an environment variable.

```
envvars["PYTHONPATH"]
```

__setitem__(*key*, *item*)

set an environment variable, both in the current environment and for future environments.

```
envvars["EDITOR"] = "emacs"
```

1.9 Executables

1.9.1 EXPERIMENTAL

This function is still being reviewed, and may be subject to changes to it's signature and naming before uranium 1.0.

Uranium provides a convenience wrapper to interact with executables. This can handle some common scenarios, like execute a script and patch in the stdin, stdout, and stderr streams of the main Uranium processes.

```
def main(build):
    build.packages.install("py.test")
    build.executables.run(["py.test", "tests"])
```

Full API Reference

class uranium.executables.**Executables**(*root*)

executables contains utility methods to interact with executables, in the context of the directory passed in.

run(*args*, *link_streams=True*, *fail_on_error=True*, *subprocess_args=None*)

execute an executable. by default, this method links the stdin, stdout, and stderr streams. in the case of an non-zero exit code, it will also raise a NonZeroExitCodeException.

for more customizability, subprocess.call() is a completely acceptable alternative. run() just has some defaults that are more suitable for builds.

returns a tuple of (exit_code, stdout, stderr)

args: a list of command line arguments

link_streams (default True): if set to true, stdin, stdout and stderr of the parent process will be used as the pipes for the child process.

fail_on_error: (default True): if set to true, raise an exception on a non-zero exit code.

subprocess_args: if set to a dictionary, these arguments will be passed into the popen statement.

example:

```
def main(build):
    build.executables.run(["echo", ""hello world""])
```

1.10 History

** Warning: This is an experimental api. It is not a final design, and could be modified in the future. **

Sometimes, you'll need to store a history of what happened previously, for caching or re-use purposes. In that case, there is a history dictionary available.

```
import requests

def main(build):
    if not build.history.get("script_downloaded", False):
        resp = requests.get("http://www.mypage.com/my_script", stream=True)

        with open(os.path.join(build.root, "my_script"), "wb") as fh:
            for block in response.iter_content(1024):
                fh.write(block)

        build.history["script_downloaded"] = True
```

The history can store any of the following primitives:

- strings
- integers
- floats

- boolean
- lists of any storable type
- a dictionary of string keys and any storable type

1.10.1 Full API Reference

class uranium.history.**History**(*path*)

1.11 Hooks

class uranium.hooks.**Hooks**

hooks are a way to add functions which run at specific phases of the build process.

the following phases are supported:

- initialize, which is executed before the build starts
- finalize, which is executed after the build stops

each function has the "build" object passed to it when executing.

```
def print_finished_message(build):
    print("finished!")

current_build.hooks["finalize"].append(print_finished_message)

def main(build):
    print("this will print finished right after I'm done!")
```

1.12 Rules

Warning: This is an experimental api. It is not a final design, and could be modified in the future.

Rules are a way to help prevent re-executing tasks unnecessarily. For example, not re-downloading a script if it has already been downloaded:

```
import os
import requests
from uranium import rule
from uranium.rules import WasChanged

@rule(WasChanged("./config.json"))
def main(build):
    with open(.path.join(build.root, "config.json"), "w+") as fh:
        resp = requests.get("http://myconfig.internalcompany.com")
        fh.write(resp.content)
```

1.12.1 Full API Reference

class uranium.rules.**WasChanged**(*path*)

WasChanged is a rule that activates if the task has never run, or if a path has a file modified since the task last ran.

```
import subprocess
from uranium import rule
from uranium.rules import WasChanged

# only run tests if the code changed.
@rule(WasChanged("./my_module"))
def test(build):
    build.packages.install("pytest")
    return subprocess.call(["py.test", build.root])
```

1.13 Managing Packages

Any configuration related to packages is done through the Packages object. Here is an example showing some common operations:

```
def main(build):
    # it's possible to set the index urls that packages will be installed from:
    build.packages.index_urls = ["http://www.mycompany.com/python_index"]

    # this method installs the package "py.test" with version 2.7.0. It's
    # available in the sandbox as soon as the package is installed.
    build.packages.install("py.test", version="==2.7.0")

    # if you want to a development / editable egg, you can use this function.
    build.packages.install(".", develop=True)

    # if you want to set a specific version of a package to download, you can do so␣
↪with versions
    build.packages.versions.update({
        "requests": "==2.6.0"
    })

    # this takes effect on all subsequent installations. For example, it will be␣
↪considered here:
    build.packages.install("requests")
```

1.13.1 Full API Reference

class uranium.packages.**Packages**(*virtualenv_dir=None*)

this is the public API for downloading packages into an environment.

unless otherwise specified, all properties in this class are mutable: updating them will take immediate effect.

index_urls

index urls is a list of the urls that Packages queries when looking for packages.

install(*name*, *version=None*, *develop=False*, *upgrade=False*, *install_options=None*)

install is used when installing a python package into the environment.

if version is set, the specified version of the package will be installed. The specified version should be a full PEP 440 version specifier (i.e. "==1.2.0")

if develop is set to True, the package will be installed as editable: the source in the directory passed will be used when using that package.

if install_options is provided, it should be a list of options, like ["–prefix=/opt/srv", "–install-lib=/opt/srv/lib"]

uninstall(*package_name*)
uninstall is used when uninstalling a python package from a environment.

versions
versions is a dictionary object of <package_name, version_spec> pairs.

when a request is made to install a package, it will use the version specified in this dictionary.

- if the package installation specifies a version, it will override

the version specified here.

```
# this sets the version to be used in this dictionary to 0.2.3.
packages.install("uranium", version="==0.2.3")
```

TODO: this will also contain entries to packages installed without a specified version. the version installed will be updated here.

1.14 Options

With uranium, arguments that configure uranium itself should be passed in before the task name, and any argument passed in afterward should be specific for the function.

For example, consider the following scenario:

```
./uranium test -sx
```

When using uranium to execute tests, one should be able to parameterize that test execution. To facilitate this, Uranium provides the Options class:

```
def test(build):
    """ execute tests """
    main(build)
    _install_test_modules(build)
    build.executables.run([
        "py.test", os.path.join(build.root, "tests"),
    ] + build.options.args)
```

1.14.1 Full API Reference

class uranium.options.**BuildOptions**(*directive*, *args*, *build_file*, *override_func=None*)
build options are user-driven options available to the build.

the following arguments are exposed:

directive: a string with the directive name (e.g. "main") args: a list of arguments, passed in after the directive name.

(e.g. ["-sx"] in the case of ./uranium test -sx)

build_file: the path to the ubuild.py file being consumed, relative to the root.

1.15 Utilities

To help make common scenarios easier, Uranium provides a set of utility methods.

uranium.**get_remote_script**(*url*, *local_vars=None*, *cache_dir=None*, *refresh_cache=False*)

download a remote script, evaluate it, and return a dictionary containing all of the globals instantiated.

this can be VERY DANGEROUS! downloading and executing raw code from any remote source can be very insecure.

if a cache directory is provided, the script will

1.16 FAQ

1.16.1 Should I use a Uranium sandbox in production?

In some cases, Uranium can work well for production. It is possible to take a sandbox and move it to another host, and have that host execute any bin/ script in the sandbox, if the following is met:

- the deploy os system matches the build os
- the deploy python version matches the build python version

A mismatch in OS will almost certainly fail: the deploy version of Python will fail in some esoteric cases (such as when the SSL verification behaviour changed from Python 2.7.3 to 2.7.9)

In general, virtualenv (and by extension uranium) works best when running directly on the host that will run the code.

As of December 2016, pex provides a much better experience when attempting to deploy to a host that doesn't completely match the build machine. pex does not bundle the python binary or any of the linked libraries, ensuring better portability.

Ultimately, portability of built Python packages is never guaranteed, due to the common usage of compiled c modules. Matching OS and python versions across machines is the best route, regardless of the packaging system.

1.16.2 build.packages.install vs setup.py's install_requires

Uranium provides the build.packages attribute to install packages into the sandbox. When working with a python package of any sort, a setup.py is provided, including an install_requires which also ensures packages will be installed.

Which one should be used? install_requires should only be used when the package in question is required by the package referenced by the setup.py. For everything else, use build.packages.

Is your service a flask application? It should have flask in it's setup.py.

Need nose to run your unit tests? add it via build.packages.install.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

Symbols

B

C

E

G

H

I

O

P

R

S

T

U

V

W