Gauge Security Development Automation

Michael Mattie

29th March, 2024

Contents

1 Introduction

Business assets go beyond code to include API's, integrations, teams, and releases. The process to get from code in a editor to a release is paved by tools.

Using those tools effectively can move a project efficiently from code to production. Not having effective devops can hobble and kill projects with broken deliveries.

In many cases developers huddled around their limited knowledge of tools, like a swimmer clinging to a life vest. Instead of enabling progress with tools, the tools were the issue, day in and day out.

Starting in 2020 I tried to sort out this mess with automation. It started as a Makefile, and then a bash script called manage.sh, and finally PythonSh. The idea was to build correct tool use into a single command, with intelligence to tailor it to a context and scenario.

You are going to see the result of 20+ years of experience, and 4 years of development baked into a mature, effective, and innovative tool.

2 Gauge Security (PythonSh) - Automation is Innovation.

I have extensive experience in professional and personal projects with Python. From that experience clearly one of the most poorly thought out and vexing aspects of coding is just getting the code to run.

Getting it to run on one developer's machine is just the beginning. Making it portable to other developers is another hurdle. Finally building a good release and delivering to the end users is yet another hurdle.

Throughout the process version control: especially with git there is often a massive time-sink in trying to fix broken repos and branches. Programmers are made timid by their lack of understanding and in their frustration they never move beyond the most basic commands.

Not knowing how to integrate packages effectively leads to monolithic repos, where the inexperience with git, and a fear of merging leads to clumsy collisions, and losses in time, code, and lengthy git debugging sessions.

Pythonsh is a path forward for python development: combining effective use of git, python, packages, and multiple repositories.

3 Development Problems

3.1 The plethora of tools

A developer typically uses language tools, IDE tools, version control tools, artifact repositories, and dependency managers. All these tools are used without any integration into a cohesive toolchain.

- Running code is fragmented process due to the use of either plain system interpreters, tox for multiple python versions, or pyenv for isolated virtual environment environments.

  Developers are often unaware of the different strengths and weaknesses of the tools and just copy the first thing they find on the internet.

- Python has many different versions, and each milestone introduces new features that code can become dependent on. Installation can be by a OS package, Open Source package managers like homebrew, or virtual environment interpreters like pyenv.

  This fragmentation leads to chaos with developers often not having a firm grasp on what python their code is actually running on.

- Without good version control practices, or artifact repositories - the sharing of code can be as bad as cut and paste or sending files over slack.
- Backwards compatibility is an assumption with no explicit guarantee. Practices for maintaining release metadata that allow a release to be rebuilt from a given source version, interpreter, and module environment are not even considered much less implemented.
- The python build system is frankly a disaster. It is both layered and fragmented with the latest standards encouraging this dumpster fire instead of putting the flames out. It's origins are setuptools with an executable config and an outdated egg format.
- Basic best practices such as the storage and use of secrets is anarchy. Developers still fequently put credentials in the code. (my utilities repo: wallet handles this)

3.2 The Tool Tax

Modern code is built with so many tools touted as giving this or that advantage to the developer, that the developer is taxed in time, learning, and management by every tool.

I call this the "Tool Tax". What usually falls out from the vast array of tools is that the developer skims 10% of the tools capability, but does not know how to use, not use, or fix any of it.

Mostly the developer tries to survive off blogs and stack overflow searching by their task for solutions, but often just digging the hole deeper, and deeper.

3.3 Automation

What then is the answer to these dilemna ? The answer is automation. Automation has three key benefits:

- speed: the pace of development is greatly accelerated by automation.
- utilization: with automation ten commands becomes one. This means that the full capabilities of the tools are realized.
- correctness: the right way, the way to avoid mistakes, is baked into the automation. Development becomes reliable and routine in the tool aspect.

Pythonsh commands are mostly single commands that accomplish the entire task by automation.

For example the ”ahead” command shows what changes have been made that are not in the trunk. Even if a developer has known how to invoke git to do this, he may fumble or look it up. With PythonSh a single command detects the structure of the repo and constructs the git command for the developer without any arguments.

There are dozens of commands with this kind of intelligence built-in. The commands have also been debugged over years of use in a wide range of repositories.

Fast,reliable, and powerful are the three prongs of automation.

3.4 Multiple repositories

Often times developers get crowded into a single monolothic repository because the construction of a new repo and it’s devops toolchain is magical, and laborious.

They collide frequently leading developers to ”section off” their files and avoid merging. When the inevitable implict merge occurs at release build time there is a frantic late-night integration sessions and working Saturdays to get the build done by Monday.

A better approach is to break the project down into components. Components into repositories, and integrate back together with packages.

Each repository or package is released on it’s own cadence and the result is stability and smooth integration.

Due to the magical nature of devops however, teams still crowd into whatever repository has a working tool-chain.

PythonSh can spin up a repository and tool-chain in under five minutes allowing code to be replicated, distributed, and built built in a modular fashion.

3.5 Mistakes

The big mistake is to think that these tool and repository hygene tasks are chrome on a semi-truck, a litte flashy but not essential to get down the road.

Its only when the project goes to build and things go sideways that the importance of the toolchain and the development practices starts to cut deep.

PythonSh deals with the Tool Tax upfront, and delivers speed, efficiency, and reliable progress from start to finish.

4 Workflows

Workflow comes in two parts

- Version Control Workflow
- Build Workflow

4.1 Multiple Repositories

Strategically it is vital to move developers and teams into their own repositories for these workflows to be effective. Each component should develop and release as fast as they can solidify.

To do this right you need a Version Control Workflow, and a Build Workflow.

4.2 Version Control Workflow

Version Control is the ultimate workflow for the developer. There is a near universal convergance on git as the version control system.

Git was developed as a implementation strata, and interfaces called porcelins. Despite improving user interfaces developers still lack understanding of core concepts like merging, and often times you are back to the command line, like it or not.

There is also no workflow defined by git, so it's the wild west unless a process is defined. Let's look closer:

There are two basic variations on workflows.

- develop on trunk, branch for release
- develop on branch, merge to trunk which is stable.

4.2.1 Develop on Trunk

Many projects do development on the trunk and branch for release. each developer, often at best is on a branch. This makes for painful merging, usually late in the game, and developers do not have an opportunity to merge incrementally.

4.2.2 Develop on a branch

The alternative is to develop on a branch, but without a development trunk each developer ends up on "forever" branches, merging with great difficulty.

4.2.3 Git Flow

Git Flow is a workflow supported by reliable and intuitive tooling. [1]

In git flow all of the shared development is on "develop" and developers work in isolated branches.

The developers all merge their work into development, and can constantly and incrementally pull from develop, merging on-the-go, so a huge mess to untangle isn't created.

That is develop, and feature branches. Releases are on main/master, merging the work from develop once develop has stabalized. When There are releases they are tagged and every release can be easily re-constructed.

4.3 Build Workflow

The correct way to integrate is by packages. Developers should not try and sync and push and pull each other into integrations that are half-baked.

Each repository releases when it is fully baked, and integration is by well defined API's instead of fragile ad-hoc source level interfaces.

It's vital to have a shared artifact repository, a simple way to build, and well tested packages to integrate.

PythonSh accomplishes this by setting up virtual environments for dependency isolation, building, testing, and releasing.

5 Introducting PythonSh

pythonsh has a script pysh-install.sh that when called with:

- private = my personal ssh checkout for development
- public = https checkout for client consumption

This script will install a git submodule pythonsh and create a symlink to py.sh for the CLI interface.

from there all commands are in the form:

```
./py.sh tools-unix
```

Most commands are a single command, a few take arguments. They accomplish a task with as much intelligence as possible so arguments don't have to be a stumbling block, and it speeds things up.

5.1 Project Creation

Pythonsh needs to install virtualenv for the user. To solve the chicken-and-egg problem PythonSh is cloned first:

Tools installation installs the tools and the zsh configuration.

```
apt install git

cd <code>

curl <pythonsh> pysh-install.sh

./py.sh tools-unix
./py.sh tools-zshrc
./py.sh tools-custom # optional, but it should be a guide for manual setup
./py.sh tools-prompt # optional prompt setup
```

New repository configuration:

```
git init
git flow init

<CODE>/pythonsh/pythonsh/pysh-install.sh public
git commit -m "first commit of a new repository"
```

Here we install git, Then we clone the pythonsh repository. The tools command install git-flow, pyenv, and pyenv-virtual

Inside pythonsh we run ”tools-unix” which installs pyenv from git source into $HOME/tools/pyenv

```
./py.sh tools-zshrc
./py.sh tools-custom
./py.sh tools-prompt
```

These tools commands setup the developer's shell for git and python. It is a toolbox with numerous functions, including virtualenv switching.

The ”tools-zshrc” command is required, along with use of the zsh shell. It is maybe possible to use bash, but you would have to rename the files like: .zshrc -> .basrc etc...

”tools-custom” sets up an environment autodetecting many key things such as pyenv, paths, ssh-agent, and the EDITOR.

On systems like MacOS where you can't hook the login, it will work fine in shells. On linux systems you can copy it into .xsessionrc and have the setup be global.

At this point pythonsh has completed global setup. Here is what creating a repository looks like:

```
mkdir project
cd project
```

```
git init
git flow init

pysh-install.sh public
```

For cloning existing project it looks like this:

```
git clone <project> <project dir>
cd <project>

git flow init

pysh-install.sh finish
```

This is all that is needed to setup a project. PythonSh is ready to use for best practices. This is what a prompt looks like:

```
<work> [system] pythonsh:develop(*+) ->
```

- work is the system name so you dont get confused when remoting into other systems
- system is the virtualenv which is not activated in this case.
- pythonsh is the repository you are at.
- develop is the current branch
- () encloses * and + , where * = dirty, and + = staged changes

5.2 Project Configuration

The idea of the python.sh file is that it contains all the information needed to drive the tool-chain. It contains version information and the names for things.

```
,# pythonsh configuration file
VERSION=0.15.0

PACKAGES=pyutils
SOURCE=.

BUILD_NAME=pythonsh

DOCKER_VERSION="0.1.0"

VIRTUAL_PREFIX='pythonsh'
PYTHON_VERSION='3.12'
```

We will circle back later on this file, but the important thing to know is that this is the ”Source of Truth” for the toolchain and as much as possible all other files needed for python are generated from this configuation.

5.3 Source Configuration

Setting up the source requires one key thing from python.sh: the directory containing the source.

```
,# pythonsh configuration file
SOURCE=src
VIRTUAL_PREFIX='pythonsh'
PYTHON_VERSION='3.12'
```

5.4 Python configuration

```
./py.sh project-virtual
```

This is where the intelligence starts. This command does:

- deactivates any pre-exising source environment
- finds the latest ”dot” release of the specified python version.
- compiles a new python interpreter if needed.
- installs the ”dev” and ”test” virtual environments.

At this point the developer would type:

```
switch_dev
```

which would activate the virtual environment for python. A dependency isolated environment for development, without any extraneous packages the developer might have on his system.

The next step is to bootstrap.

5.5 PythonSh bootstrap

A virtual environment doesnt have packages that pythonsh itself needs, nor does it have pipenv for package management, or development tools.

Bootstrap initializes the virtual environment and does so in three stages. All of boostrap is fully automatic.

- ugprade pip
- install pipenv
- install dependencies for pythonsh
- search the source tree for Pipfile fragments, merge into a root Pipfile
- install the dependencies of the repository.
- search installed packages as well as sources for Pipfile fragments
- make a second merged Pipfile with source and package Pipfile fragments
- install combined dependencies.

The merging process sorts through all the dependencies in source and packages managed by PythonSh. It takes the higher version of every version comparison and generates a root Pipfile.

Some would say this is imperfect: that only PythonSh packages can be merged - which is true. But by synchronizing versions at teir-1 and tier-2 the problems with package version issues are massively reduced, and far more tractable to solve. Often pendantic hand wringing is a obstacle to making practical solutions.

5.6 Integrating Source into the virtualenv

There are a couple of ways to insert the source into the python virtualenv. The first is with an editable package, the common way. A second way is to put a .pth file into site-packages.

I prefer the uncommon .pth file approach since it is more flexible, and I will usually prefer flexiblity over dogma.

```
./py.sh  add-src
```

This installs a .pth file from python.paths, a file in the repository. Both absolute and relative paths are accepted.

```
show-paths  =  list  .pth  source  paths
add-paths    =  install  .pth  source  paths  into  the  python  environment
rm-paths      =  remove  .pth  source  paths
site              =  print  out  the  path  to  site-packages
```

- show-paths: shows all the paths in the virtual environment
- add-paths: installs a pth file generated from python.paths in the repo root
- rm-paths: removes the .pth file
- site: prints out the virtualenv site-packages directory location

The next step is to get the source code into the virtualenv. There is a way to make it possible by using ”editable” packages, however I prefer a second approach. It is possible to put ”.pth” packages into site-packages in the virtual environment.

5.7 PythonSh Starter Kit

Here is a template for starting a PythonSh repository. The code is in scripts/starter-kit.sh

```
REPO=$1
CLONE=$2
BRANCH=$3

git  clone  $REPO  $CLONE
cd  $CLONE

,#  setup  git  flow
git  flow  init

,#  install  pythonsh
test  -d  pythonsh  ||  $HOME/code/pysh-install.sh  public

,#  edit  python.sh
$EDITOR  python.sh

,#  create  the  virtual  environments
  ./py.sh  project-virtual

,#  install  source  shims
$EDITOR  python.paths
./py.sh  add-paths

,#  boostrap  virtualenv
./py.sh  bootstrap

,#  start  the  feature  branch
git  flow  feature  start  $BRANCH
```

This is a complete developer environment and devops toolchain in less than five minutes.

6 Version Control Workflow

The version control workflow is the most difficult part for developers to master due to the frequent need to understand complex history graphs, and arcane commands.

Usually there are only loose practices around commits, and it makes it impossible to ”look over the shoulder” and understand what is going on in the repository.

6.1 Version Control Conventions

It is vital for the tools, and for the developers to adhere to conventions in commits, tags, and releases. How can another developer understand your history if your commit messages are: ”fixed some bugs”?

6.2 Conventional Commits and Reports

Conventional commits [2] is a standard for semantics and formatting of commits. I use it as a starting point, and add a couple such as refactor, and sync.

```
(feat) add a new dialog for listing reports
```

These conventions are crucial since it makes it clear to developers what a commit consists of, and allows tools to process the history in powerful ways.

To really understand the power of conventional commits you have to consider the tooling. What if it was possible to generate release notes entirely from commits ? Pythonsh does!

This is what a history looks like, a jungle of different types of commits:

```
(sync) [2024-03-16T08:16:03-07:00] syncd: pythonsh
04fb73e Merge branch 'release/0.10.0' into develop
8f13b6b (sync) [2024-03-15T22:35:36-07:00] syncd: pythonsh
776971e (release) release 0.10.0 many fixes to dwim, support for org mode etc..
dbd926b (fix) insert the commit type as well as the message and report
1ad304d (fix) report no longer takes a message argument so insert the message ourselves
c6f8a67 (sync) [2024-03-15T22:14:20-07:00] syncd: pythonsh
0fbf0df (feat) create a insert-syncd command that generates a sync commit message
fd93322 (sync) 3-15-2024 sync pythonsh
b8b069a (fix) make m keybinding be menu and remove a
420f003 (feat) functionalize the helm frame configuration
7bea9b9 (sync) 3-15-2024 sync latest citeproc,helm, and helm-bibtext
ec46225 (fix) add ignore=dirty to helm-frame
```

This is what a report looks like. It groups the commits by type and is injectable into a commit. This allows for editing the report into release notes in the commit.

Without release notes it's only tribal knowledge what is in a release or not. Professionals do not leave history up to word of mouth. Take a look at a report and see how easy it is to edit into release notes.

```
devil> [system] grail:develop(*) -> ./py.sh status-report

* features

(feat) through questions determine what type of report to insert
(feat) create a insert-syncd command that generates a sync commit message
(feat) show the branch on the modeline
(feat) add a bunch of fonts and a install script for macos
(feat) add a tramp command that opens dired on the host home directory
(feat) revamp the scripts to build emacs from brew and deal with byte copmilation

* fixes

(fix) insert the commit type as well as the message and report
(fix) report no longer takes a message argument so insert the message ourselves
(fix) make m keybinding be menu and remove a
(fix) add ignore=dirty to helm-frame
(fix) fix the battery with a closing >
(fix) remove initial bib file which is obsoleted by compsci repo now
(fix) disable helm-frame for now

* syncs

(sync) [2024-03-16T08:16:03-07:00] syncd: pythonsh
(sync) [2024-03-15T22:35:36-07:00] syncd: pythonsh
(sync) [2024-03-15T22:14:20-07:00] syncd: pythonsh
(sync) 3-15-2024 sync latest citeproc,helm, and helm-bibtext
(sync) ff pythonsh to 3-15-2024
(sync) sync helm core 3-15-2024

* refactor

(refactor) clean up formatting so it's easier to read
(refactor) cleanup formatting
(refactor) pull all the scripts and files for emacs from pythonsh and combine into compile-emacs.sh
(refactor) refactor out mattie references and fix scheme repl auto start
(refactor) minor whitespace changes
(refactor) clean up lex-cache and check dwim-complete
```

```
<devil> [system] grail:develop(*) ->
```

Here are my prefixes which extend the conventional commit standard:

- (feat) new features. A oneliner is either sufficient or some prose is added below the main commit line.
- (fix) this is primarily for development. They belong on feature branches. fixes are corrections to code that has not been released yet.
- (bug) bugs are defects in code that has been released. They need to be included in the release notes
- (issue) issues are bugs that have been reported by users and have a ticket assigned.
- (sync) a fast-forward. This is done only on the trunks: develop and main where they are histories that are stable, and consist entirely of merges.

  The other use case is for third party submodules. since .gitmodules and git internals remember the commit sync'd its not a good idea to introduce local commits. That will get ugly.

  For (sync) is is critical that the date be in the one-liner as dependencies are being updated and this has a large impact on the release.
- (pull) for working on feature branches, pull is for pulling changes into the feature branches
- (merge) merge is for merging developer work into the shared development trunk.
- (release) releases are alpha and beta releases. The actual release process with git flow release start is more complex and is documented below
- (alpha) both tag and possibly a commit this indicates it's a beta candidate and the developer wants to tag/commit to establish a baseline
- (beta) This is on the develop trunk and indicates that this is a point from which beta_fix and beta_<feature branch> should be branched off this point.
- (refactor) a change to make development or maintenance easier that has no impact on functionality
- (doc) documentation updates.

This systematic annotating of the history makes it possible to understand the changes far beyond cryptic and poor commit messsages.

This also allows for tools that help insert commit messages, and generate entire release notes into merge commits and the like.

```
./py.sh status-report
./py.sh release-report
```

Status report shows all the developers changes grouped by type that are outstanding from the development trunk.

The release-report shows all the work in the development trunk outstanding since the last release. With tag, branching, and commit conventions this is fully automated.

6.3 Git Flow - the nitty gritty

git flow establishes a structure that is time-proven and boosts productivity and "incremental" merges instead of putting off merging until the final moments on a Friday.

git flow init and the developer work by this process:

- git-flow: creates main/master as the release branch
- git-flow: creates develop as the development trunk
- git-flow/developer: creates ”feature” branches for a specific task.
- developer: works in ”feature” and merges develop changes from the team with ”pulls”
- developer: when work is done, ”squashes” the ”feature” and merges into ”develop”
- developer: when ”develop” is ready for testing, they make a ”alpha” tag
- developer: when integration begins ”beta” tags are created.
- developer: when integration is complete a ”release” merge into ”main” is done.

All of the developer tasks are not done manually, instead they are done with either PythonSh commands or git flow commands.

6.4 PythonSh Version Control Feature Summary

- track = set upstream tracking
- tag-alpha = create alpha tag
- tag-beta = create beta tag
- info = show branches, tracking, and status
- verify = verify commit cryptographic signatures
- status = show status of repository and all sub-modules
- fetch = fetch main, develop, and current branch
- pull = pull current branch no ff
- staged = show staged changes
- merges = show merges only
- releases = show releases (tags)
- history = show commit history
- summary = show diffstat between feature and develop or last release and develop
- delta = show diff between feature and develop or last release and develop
- ahead = show log of commits in branch but not in parent
- behind = show log of commit in parent but not branch
- release-report = generate a report of changes since last release
- status-report = generate a report of changes ahead of the trunk
- graph = show history between feature and develop or last release and develop
- upstream = fetch upstream and show changes not yet merged
- sync = merge from the root branch commits not in this branch no ff

6.5 PythonSh Version Control in-depth

Let’s look at the version control capabilities in-depth and see what developers could do if they intensively studied git and git-flow.

6.5.1 status

This is a example of using status:

```
<devil> [pastepipe_dev] pastepipe:develop(*) -> status
On branch develop
Your branch is up to date with 'origin/develop'.

Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)
    (use "git restore <file>..." to discard changes in working directory)
    modified:     Pipfile.lock
    modified:     pyproject.toml

Untracked files:
    (use "git add <file>..." to include in what will be committed)
    dist/
    src/pastepipe.egg-info/

no changes added to commit (use "git add" and/or "git commit -a")
<devil> [pastepipe_dev] pastepipe:develop(*) ->
```

6.5.2 info

```
<mobile> [system] pythonsh:develop(*) -> ./py.sh info
* develop 879b2a4 [origin/develop] (feat): put code directory [zshrc.custom]
    main      7f47e7c [origin/main: behind 134] Merge branch 'release/0.15.1'
[staged]
  0 files changed
[changes]
  pythonsh.org |   130 +++++++++++++++++++++++++++++++++++-------------------------------
  1 file changed, 75 insertions(+), 55 deletions(-)
[untracked]
<mobile> [system] pythonsh:develop(*) ->
```

py.sh info shows the stauts of the branches and the repo. this is a very handy command.

6.5.3 track

- track <1> <2> = set upstream tracking 1=remote 2=branch

sometimes you need to set the upstream for a branch. track makes this easy.

6.5.4 fetch & pull

- fetch = fetch main, develop, and current branch
- pull = pull to current branch no ff

fetch retrieves the commits from upstream but does not merge them. pull is basically fetch + merge.

6.5.5 staged

- staged = show staged changes

show staged changes. Note that git diff showing the unstaged changes is a shell alias.

6.5.6 Advanced View

- merges = show merges only
- history = show commit history
- summary = show diffstat of branch to trunk or trunk to release.
- delta = show diff of branch to trunk or trunk to release

- log = show log of branch to trunk or trunk to release
- graph = show history graph of branch to trunk or trunk to release
- upstream = show upstream changes that havent been merged yet

The most powerful feature is ”agains the parent”. What this means is that pythonsh detects if it’s on a feature branch, the develop trunk, or the main trunk.

- if on a feature branch it’s a diff from develop -> feature
- if on the develop branch it’s a diff from main -> develop
- if on main it’s a diff from the last tag -> main

This intelligence means a single command can be used in three different contexts with no additional arguments.

- sync = merge from the root branch commits not in this branch no ff

sync is a tool to pull changes from the parent into the current branch. This is used for when development work on the develop trunk needs to be merged into the feature branch.

6.5.7 tagging

- tag-beta = <feat> <msg> : create a beta tag on the trunk

tagging is important for making a file set for alpha or beta releases. by drawing a line across the repository the entire state of the repo can be checked out.

7 Packaging and Integration

Python packaging can be very difficult because there are many different systems fragmenting the tool-chain into camps that don’t get along.

The Python developers tried to impose some order on the build process. The PEP 517 standard with pyproject.toml is their attempt to homogenize the build landscape.

However instead of making things uniform it fanned the flames by specifying backends as plugins, and duplicated the dependancy information also in Pipfile.

Now the developer has to keep in sync both Pipfile and pyproject.toml. This is arguably almost worse than before.

PythonSh uses the PEP517 build, but instead of maintaining the files by hand, Pythonsh puts Pipfile fragments in the source modules and generates both the Pipfile, and pyproject.toml from these fragments.

This means that the files will always be in sync since they are generated by the same tool, and from the same sources.

7.1 Virtual Environments

Python package management takes place in virtual environments. These are directories that have a python built from source and a set of installed packages.

When you ”activate” a virtual environment and your shell is correctly set you can execute programs, including python, in that environment.

7.1.1 Virtual Environment Stucture

A project has four virtual environments

- dev: for development
- test: for pre-release testing
- build: for building a release
- release: for testing release packages

The dev environment is for the development work. The test environment is for testing for release ready.

The build environment is created and destroyed automatically. The release environment is created as needed. The release environment is for dev:prod parity and testing the built package or packages without the development packages present.

the "dev" and "test" environments are the commonly used ones. With the shell setup by py.sh typing

- "switch_dev" = switch to development environment
- "switch_test" = switch to test environment

The most important thing is to focus on with virtual environments is that dev, test, and release is that they are kept in sync mirroring places like cloud environments, or on-site environments.

The process for code to bake is: dev -> test -> release

- First dev is a sandbox for developmental code.
- Test is an environment for integration testing.
- release is a environment for checking that the build works in "prod".

Code is first built in dev. From dev it's promoted to testing to integrate with other developers. From test it's release tested. If there is feedback from release or test it goes back to test.

7.2 Building

Building should be done in an isolated environtment. tox allows for tests and such to execute in different environments but this will dissapear as older python versions are phased out. With virtualenv you can take your python with you so multiple versions of python isn't a target anymore.

```
./py.sh  build
```

This is all it takes to build a package with PythonSh.

8 Release

The release process is often the worst of the practices in the environment. It is common convention to never do a release late in the week, because there is always a huge hurdle of after-release activity to hammer the release into shape.

This is absolutely unacceptable. Good programmer's dont forget stuff in the release or have to patch the release numerous times due to integration issues, and missing files or code. Good programmers take extra effort to hit the mark with releases.

Second of all it should be an absolute rule that enough is recorded to make it possible to rebuild a release. If your repository doesn't have enough information to rebuild, and the situation arises where you need to, it's like an airbag: you don't need it usually, but when you do, it's a life saver.

8.1 PythonSh Release Process

PythonSh walk the developer through a automated process to perform the release.

8.2 Building & Testing

The build for a release can come in two flavors with python.sh:

- Singular packages created by the python build module.
- The second type is a buildset package which is an abbreviation for built set. it's a zip named like a wheel, except it's a all the runtime dependencies gathered from the test virtual environment.

buildset packages are used when there are private packages in the mix and we need to be able to install all the dependencies in one shot.

to start the release proccess a release environment is created.

```
./py.sh  mkrelease

switch_release

pipenv  install  <package>
```

This use of a release virtualenv allows the package to be tested in an environment that mirrors ”prod”

Code takes time bake, and so rushing into a release is not a good idea. after some time has passed and a few final fixes are made it's time for the full source release process to start.

8.3 Source Release Procedure

- checks are ran to make sure the repository and virtual environment are ready for a release
- the release is created with git flow
- Pipfile is generated and locked
- The Pipfile, Pipfile.lock, and python.sh are copied into a release/ directory with the version of the release appended
- all files added or generated are added to git
- an automatic git commit is performed

8.3.1 Source Release commands

- check = check virtualenv, fetch upstream, show repo state
- start = <VERSION> = update VERSION in python.sh, reload config, snapshot Pipfile if present, and start a git flow release with VERSION
- release = finalize the release with a git-flow finish command.
- upload = push main and develop branches, and push and tags

Then the release drops down to a shell so the developer can inspect the release. On this release branch the developer can fix up any missing or incorrect bits.

Git flow finish does:

- The release is merged back into main
- The release is merged back into develop
- a tag is created starting with ”release-<VERSION>”

The developer would insert the release-report into the release trunk merge producing a easy to understand set of release notes describing all the ingredients baked into the release.

A Appendix

A.1 python.sh

python.sh is the master file for pythonsh. It contains all the variables needed to generate python files.

The idea is that there is one master file, and all the other files are generated from it so they are all synchronized.

Unfortunately python has numerous redundancies so syncing them up is key, and best done with a single master file.

Here is an example from pythonsh itself:

```
,#  pythonsh  configuration  file
VERSION=0.14.0

PACKAGES=pyutils
SOURCE=.

BUILD_NAME=pythonsh

DOCKER_VERSION="0.1.0"

VIRTUAL_PREFIX='pythonsh'
PYTHON_VERSION='3.12'
```

- VERSION = the version of the repository
- PACKAGES = packages that comprise the project
- SOURCE = the directory containing the package sources. it is typically: ”src/”
- BUILD_NAME = the name of the built packages
- VIRTUAL_PREFIX = the prefix for the virtualenvs. pythonsh = ”pythonsh_dev” etc...
- PYTHON_VERSION = what python version to install/use

From this the following packages are generated:

- pyproject.toml = PEP517 build template. contains build system directives and runtime dependencies
- Pipfile = Dependency management. sections for repositories, dependencies, and other variables.

A.2 Virtual Environment Creation

When the virtual environments are created the latest possible PYTHON matching the PYTHON_VERSION is installed. This is done automatically. If a interpreter has already been built for that version it is re-used.

Then the virtualenvs are created by

```
./py.sh  project-virtual
```

This creates VIRTUAL_PREFIX-{dev,test}

Then the environment is bootstrapped.

A.3 boostrap

```
./py.sh bootstrap
```

There are many steps to a bootstrap

- The pip command is upgraded, pipenv is installed
- ./py.sh minimal which installs only the packages needed by pythonsh itself
- then a search is made of the source directories for .pypi and Pipfile

This is unique to pythonsh. Normally all Pipfile instances are singluar and at the root of the tree. However pythonsh is built to find fragments of Pipfile in source directories, and installed packages.

The .pypi fils define repositories. Typically for open-source projects only the central pypi repository is used. However for commercial projects private artifact repositories are used as well.

- now all the .pypi repos and fragements are merged by the highest version

This merging process reduces tangled dependencies by syncing all the dependencies at the teir 1 packages.

- The root Pipfile is written with the packages are installed.
- A second pass then searches installed packages for fragments and merges those

This second pass allows us to gather Pipfile fragments from installed packages from the first pass.

- Now the final install takes place with all the teir-1 and teir-2 dependencies synced.
- at both stages vulnerability checks are performed.
- finally pyproject.toml is written for the PEP517 build ”build” module.

The pyproject.toml build file contains all of the information needed to build the package.

It is not currently possible to specifiy additional repositories with a setuptools backend in pyproject.toml. This means that if there are private repositories it’s not possible to specify the dependencies.

When all of the packages are on pypi a dependencies list will be written to pyproject.toml. If there are other repositories dependencies will be supressed but the rest of the file will be written.

This is the boostrap process. The end result is that the active virtualenv will contain a highly homogenous package set for the project.

Actually pyproject.toml is not generated until a package build is performed but the two files: Pipfile and pyproject.toml share a context.

A.4 Python commands

- test = run pytests
- python = execute python in pyenv
- repl = execute ptpython in pyenv
- run = run a command in pyenv

The python commands include all of the basic functionality for python development.

A.5 Package commands

- versions = display the versions of python and installed packages
- locked = update from lockfile
- all = update pip and pipenv install dependencies and dev, lock and check
- update = update installed packages, lock and check
- remove = uninstall the listed packages
- list = list installed packages

References

[1] Atlassian. *Git Flow*. 2024. URL: `https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow` (visited on 03/13/2024).

[2] Conventional Commits. *Conventional Commits*. 2024. URL: `https://www.conventionalcommits.org/en/v1.0.0/` (visited on 03/17/2024).