Bazel + Python Design Notes

Status: Draft
Visibility: **PUBLIC (shared outside Google to bazel-sig-python)**
Authors: dgreiman@google.com
Last Updated: 2018-02-07

Objective

Enable Bazel to be a useful part of a modern Python build, test, and deploy workflow.

Background

The "modern Python workflow" is the workflow described by the Python Packaging Authority circa 2018, along with relevant PEPs, and the documentation on www.python.org. Bazel is described on www.bazel.build. The Google-internal Python workflow will be referred to as "Google3", and is not fully documented publicly. Blaze is the Google-internal counterpart to the Bazel tool, and is not documented publicly.

Glossaries:

- Python Packaging Glossary. In particular, the word "distribution" used here has nothing to do with Linux distributions.
- Bazel Concepts and Terminology

Ambiguous Term: "Package"

- "Bazel Package": A directory tree containing a BUILD file, excluding any subdirectories which themselves contain a BUILD file. It has a name like "//google/cloud/logging".
- Python "Distribution Package": A versioned archive file that contains Python import packages, modules, and other resource files that are used to distribute a Release. It has a name like "google-cloud-logging".
- Python "Import Package": A Python module which can be imported at runtime, and contains other modules and subpackages. It has a name like "google.cloud.logging". Some Import Packages are "Namespace Packages", where the contents of multiple separate directory trees are unioned at runtime into a single hierarchy.

Noteworthy Terms:

- "Requirements File": (requirements.txt, Pipfile, Pipfile.lock): A file containing a list of versioned dependencies. Versions may be strict or loose. Declarative.

- "Project specification file": (setup.py, pyproject.toml): Specification file for creating Distribution Packages. May contain a list of versioned dependencies. May contain arbitrary imperative Python code.
- "PyPI": The global Python Package Index. Developers may also want to run local mirrors and/or private package indexes.

Problems

Output Artifacts

In a language like C++, the inputs and outputs of the build process are fairly well understood. The inputs are source files (.cc, .h) and the outputs are executables and libraries (.exe, .so, .o, .a, .dll, .lib). Likewise, for Java, the inputs are source files (.java) and various metadata inputs, and the outputs are the equivalent of executables and libraries (.jar, .war).

By contrast, Python has little agreement on what the equivalent of an "executable" or "library" is. The dynamic nature of the language and its import mechanism allow many possible structures. So, as a result of the build process, a Python developer might want any or all of the following output artifacts:

1. Source files (.py): The source files might be copied or linked from the source tree to some output location.
2. Generated source files (.py): Tools like the Protocol Buffer compiler generate Python source code from other inputs.
3. Bytecode files (.pyc): Python source code can be converted to Python bytecode by the interpreter to save time during execution. This usually requires that the Python interpreter used during the build process be the same as the Python interpreter used during program deployment.
4. C Extension Modules (.so, .dll, .lib): Compiled native code that can be imported by the Python interpreter at runtime. Despite the name, they can be compiled from C, C++, FORTRAN, Cython, or nearly anything else imaginable. They must be compiled for a particular ABI, which is a combination of processor (x86/x64/ARM...) ABI, operating system ABI (ELF/PE/Mach-O...), C ABI (libc symbol versioning), C++ ABI (C++03/11/14/17, helper libraries, exception handling, stack and structure layout, templates and inlining, linkage, etc), Python ABI (unicode width, debugging support, 32 bit vs 64 bit, Python Extension API version, etc) and more.
5. Native Libraries (.so, .a, .dll, .lib): Compiled native code that is not itself an extension module, but is a dependency of one or more extension modules. For example, the `_tkinter` extension module depends on the `libtcl8.6.so` shared library. May be in the form of a dynamic (shared) library (.so, .dll), or a static library (.a, .lib). If two extension modules depend on the same native library, the result is the "Diamond

Dependency" problem. If a single process includes multiple versions of the same native library, the result is a “One Definition Rule” (ODR) violation.

6. Generated C/C++ source files: Tools such as Cython (CLIF, SWIG, Protocol Buffers, ...) may generate C/C++ source code to be compiled into extension modules and/or native libraries.

These artifacts may be packaged for test, deployment, and execution as any of the following:

7. "site-packages": A semi-structured directory tree containing Python source files (.py), bytecode (.pyc), compiled C extensions (.so, .dll/.pyd), path files (.pth), and arbitrary other data files and executables. Normally encountered as part of Python interpreter installations, or inside a virtualenv. This is sometimes called "dist-packages" (for example, on Debian systems). The site-packages directory is usually found under the Python standard library directory, but is conceptually separate from it.
8. "runfiles": A directory tree similar to "site-packages" with a different directory arrangement. Created by Bazel. Does not (currently) use Python .pth files, instead the Python import path is setup by a helper script ("stub") before the main Python program starts execution.
9. "virtualenv": A semi-structured directory tree created by the "virtualenv" program. Contains a "site-packages" directory, along with a symbolic link and/or copy of a Python interpreter binary and its associated helper files. There are multiple tools and virtualenv formats in use.
10. "zip" (.zip, .egg, .par, .pex, .pyz): A single file in zip-archive format, designed to be directly imported by Python. Contains a structured directory tree, similar to the structure of a "site-packages" directory, a "runfiles" directory, or something else. Relies on an external Python interpreter. There are many flavors, with widely varied functionality and limitations.
11. "hermetic launcher": A statically linked executable containing a Python interpreter and native libraries. Normally combined with a zip archive to form a single "hermetic par file", providing a completely self-contained Python interpreter, Python source files, C extension modules, and dependent native libraries.
12. "executable": Similar to "hermetic par file", but may use a variety of other strategies to produce an executable file. For example, binaries produced by the py2exe tool.
13. "embedded executable": The Python interpreter may be embedded inside another executable, for example the uWSGI webserver or iPython REPL. Source files, bytecode, extension modules, and native libraries may also be linked into that executable, or loaded dynamically at runtime.
14. "sdist": (.tar.gz) A Source Distribution suitable for uploading to, and downloading from, PyPI.
15. "wheel": (.whl) A Built Distribution suitable for uploading to, and downloading from, PyPI.
16. Something else

Difficult Scenarios

Build tools generally, and Bazel specifically, easily handle the scenario where pure-Python source code is written and executed in a single machine environment. Therefore, this document concerns itself with more complex scenarios.

We assume a project will span multiple programs, multiple source files, multiple packages, multiple repositories, and/or multiple sets of dependencies (both Python and native code).

We assume first-party and/or third-party C Extension Modules will be used.

One source of complexity occurs when the code is built in one machine environment, and run in a different machine environment. This is very common. We will generically call this "cross-compilation", even though we're using it more expansively than usual (suggestions for better terminology welcome). For example:

1. The output artifact is built in one directory ($HOME/foo/bazel-bin/bar/baz), then copied to another directory to be be run (/usr/local/bin). This may cause problems with code that initializes the Python import path ($PYTHONPATH/sys.path). This may cause problems with extension modules that use RPATH or equivalent to find their shared library dependencies. This may also cause problems when the output artifacts are stored as symlinks (as in "runfiles" tree on non-Windows platforms).

2. The Python interpreter is in one directory during the build process, but a different directory at runtime. For example, the build process may be run inside virtualenv(s), where the Python interpreter path is "<env>/bin/python", while the run process is not, or vice versa.

3. The Python interpreter is one version during the build process, but a different version at runtime. For example, a developer laptop may have Python 2.7.14 installed, while production servers may have Python 2.7.13 or 3.5.4 installed.

4. Multiple simultaneous Python versions. The developer may have some code that has been ported to Python 3, and other code which requires Python 2. Or, the developer might be developing a Python library and want to test that the same code works under Python 2.7, 3.5 and 3.6. Or, the developer might have Python code which is invoked during the build process to perform some build function.

5. Shared library, tool, or other differences. The build process may occur on a machine that has libtcl8.6 installed as a system library, but run on a machine that has that library in a different system directory, in a non-system directory, or does not have that library at

all.

6. Operating system differences. The developer may be developing on a Mac laptop, but building code for a Linux Docker container.

7. Processor differences. The build process might occur on an x64 processor, but target an ARM processor for runtime.

Other complexities:

8. Shared and/or “diamond” dependencies, both Python and native code.

9. The Python interpreter might be embedded in another executable.

10. Non-CPython interpreters. For example, PyPy, IronPython (.NET), or Jython (JVM).

11. Non-language-specific packaging tools. The developer might want to create a containerized image, Windows Installer, MacOS dmg, or similar.

Technical Points

Extension Modules - Cross Compilation

The build system may need create an extension module that it can't actually run, for any of the reasons listed above. In fact, the build process may have to build multiple different instances of an extension module. For example, if an extension module is needed at build time and at runtime, it may need to be built once in the host configuration(s), and again in the target configuration(s). Bazel Dynamic Configurations.

The standard Python mechanism for building extension modules is for each package to have a source file called "setup.py", which invokes the "distutils" module in the Python standard library, usually in concert with the "setuptools" package. “distutils” has basic modules for compiling and linking extension modules, in the form of the “CCompiler” and “Extension” classes, and setuptools extends those classes with extra functionality.

Each distribution package's setup.py is unique, and can include arbitrary Python code to do things like: Manually inspect the filesystem for specific files (header files, compiler executables, native libraries, or anything else) at particular hard-coded paths, invoke arbitrary executables, and invoke arbitrary operating system services. Some packages have a very simple setup.py that invokes "distutils" with a few arguments. However, many important packages have very complex setup.py files. For example, the "numpy" package has several thousand lines of quite complex code to build itself.

All of this code is executed at build time in the host configuration. “distutils” obtains compiler executable paths and command line flags by reading the Makefile used to compile the host Python interpreter, parsing out various lines of interest, and applying a variety of heuristics. There is some rudimentary support for cross-compilation in "distutils" and "setuptools", but it only works for the simplest cases. And each package's unique setup.py code might or might not support cross-compilation (almost always not) depending on how it was written.

Note that "distutils" and "setuptools" have a complicated development history. In the interest of feasibility, we will require use of "distutils" version 2.7.x or 3.x.x and above, and "setuptools" version 38.x.x and above, and not support the ".egg" format. This effectively drops support for Python 2.6.

Extension Modules - Native Code Dependencies

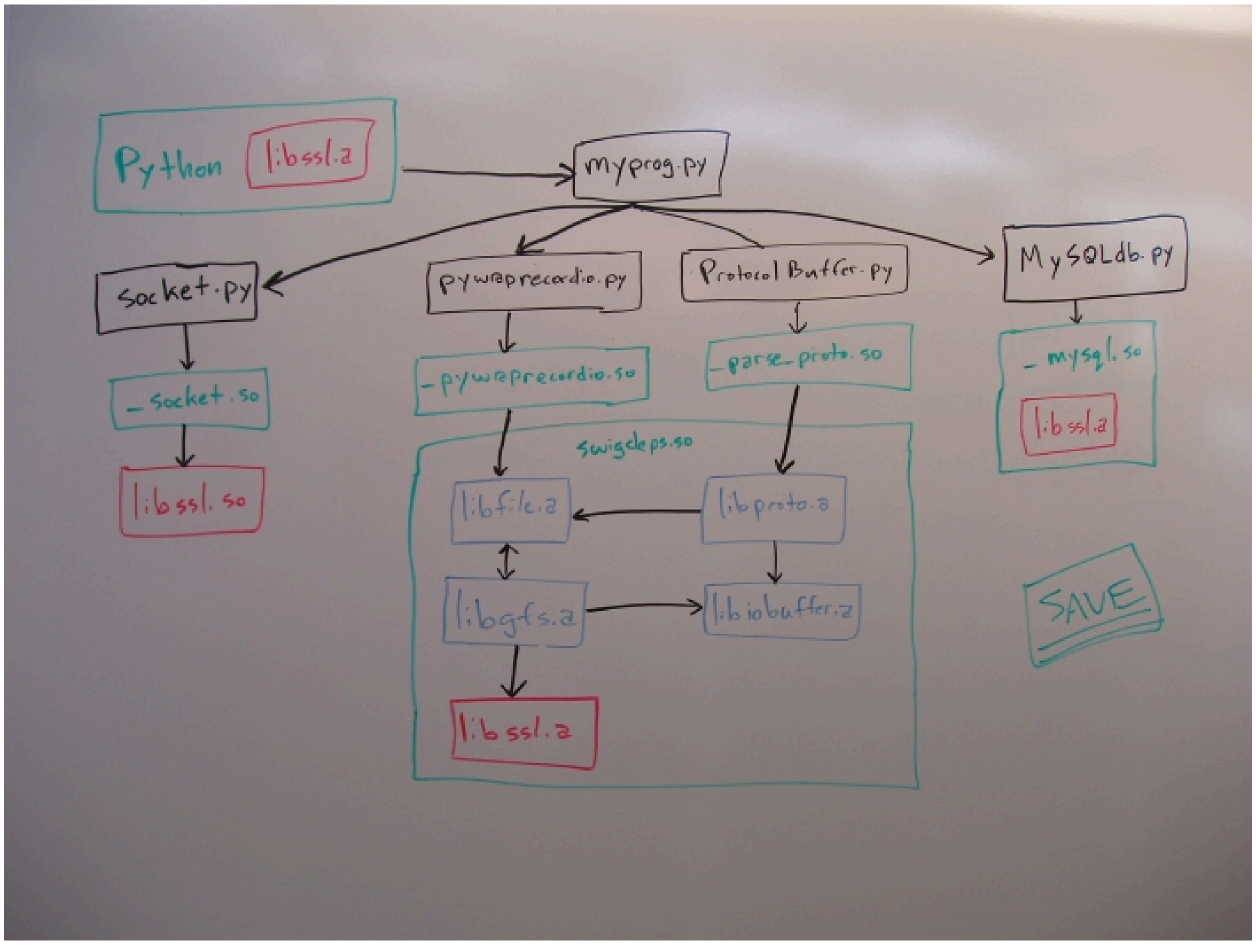
Native code dependencies are a hard problem. In general, C programs can only have one version of a particular library. Google handles this internally by very strictly controlling all aspects of the environment, and enforcing a single workflow across all developers. This isn't feasible for Bazel's users, and isn't entirely desirable for Googlers, so we will come up with an alternate scheme.

For the pedantic: With extreme care and the proper compilation and linking scheme, it's _possible_ to have multiple different versions of a single C/C++ library in the same process, but this will inevitably end in tears. And by tears, I mean incredibly hard-to-debug memory corruption, bugs, and crashes. Also, tears. 😢

Hypothetical Example Program

This program, called `myprog.py`, uses the Python standard library `socket` module, various first-party Python/C++ libraries, and the third-party extension module `MySQLdb`. The library "libssl" is a dependency of multiple other extension modules, other native libraries, and perhaps the Python interpreter binary itself.

Black Python source code
    myprog.py, socket.py, pywraprecordio.py, ProtocolBuffer.py, MySQLdb.py
Green C Extension Module
    _socket.so, _pywraprecordio.so, _parse_proto.so, swigdeps.so, _mysql.so
Blue Compiled native library
    libfile.a, libproto.a, libgfs.a, libiobuffer.a
Red Third-party native library with version conflict
    libssl.so, libssl.a



Linking against system libraries

Every piece of code in the example program above must make the same decision about where to get the dependency "libssl" from. If, for example _mysql.so statically links the system-installed /usr/lib/libssl.a into itself, but _socket.so decides to dynamically link against a copy of libssl.so built by Bazel, the mostly likely result is disaster, even if the two are identical versions. If they are different versions, or the same version compiled differently, the problem is even worse.

Internal Google Approaches

1. Hermetic launcher

   All C/C++ code, including the Python interpreter, all extension modules in the Python standard library, all extension modules and native libraries in first-party code, and all third-party extension modules and native libraries, are built from source. All native code except for extension modules is statically linked into a single executable. Extension modules are loaded from either the filesystem, or a zip file embedded within the hermetic launcher, by a special-purpose loader.

Pros:

1. Avoids all version skew, ODR violations, and diamond dependency problems.
2. Easy to deploy - just copy a single file and run.
3. Updates to an installed Python interpreter cannot break a deployed application.

Cons:

1. Must have Bazel rules to build the Python interpreter and all third-party dependencies. These rules are quite laborious to write and maintain.
2. The current implementation is Linux-only and depends on custom libc patch so that .so files can be loaded from a single-file Python binary without extracting them. It is possible to re-implement this in a way that does not need a custom libc. It is also possible to create implementations for other operating systems.
3. Adds a ~20MB fixed overhead for every Python program.
4. Building a single Python program in a clean workspace requires building the Python interpreter, standard library, and all dependencies from source.
5. Requires a one-version policy: The entire repository must agree on a single canonical version of all libraries and dependencies.

2. Native Code Dependencies Dynamic Shared Object (aka "swigdeps.so")

   All first-party and third-party code is built from source. All extension modules are built as dynamic shared libraries (.so). All native code dependencies are built as static libraries (.a), and then linked into a single dynamic shared library per Python program, called "_name_of_program_swigdeps.so". Note that the name is a historical artifact and has nothing specifically to do with the tool SWIG.

   Pros:

   1. Avoids most version skew, ODR violations, and diamond dependency problems

   Cons:

   2. Deployment requires copying an entire directory tree.
   3. Must have Bazel rules to build all third-party dependencies.
   4. Might not be able to use the system Python interpreter. The Python interpreter used must carefully match the linking and compilation options used by Bazel, but does not need to be built by Bazel.
   5. The Python interpreter must be separately deployed, and version skew is a serious problem. Upgrading interpreters in a production environment can be quite difficult.

Since neither of these approaches works well for Bazel, alternatives are discussed below.

Package and Dependency Granularity

Inside Google, there is a strict relationship between Blaze packages, Python Import Packages, and repository layout in the filesystem. There is a single Blaze workspace, rooted at a directory called “google3”. The directory “google3/foo/bar” == the Blaze package “//foo/bar” == the Python Import Package “google3.foo.bar”. Python Distribution Packages are not used. The directory tree “google3/third_party” is handled specially, so that the directory “google3/third_party/py/foo” == the Blaze package “//third_party/py/foo” == the Python Import Package “foo”, where “foo” is the normal name the package would have outside of Google.

The most important unit of organization is the Blaze target (py_binary, py_library, etc) rather than the Blaze package. Dependencies are specified on a per-Blaze-target granularity. Therefore, a single Blaze package “//foo/bar” could have two programs “baz” and “quxx” with completely different sets of (possibly incompatible) dependencies.

Historical note: The Google3 repository was originally intended to have a package-centric organization, where each package would contain a single non-test library or binary, with a single set of dependencies (This is why “//foo/bar” target notation is shorthand for “//foo/bar:bar”). However, it didn’t really work out like that. Also, in hindsight, it would have been nicer to have a single aggregated test target per package, instead of requiring developers to specify one per source file. So, test targets in the same package can also have completely different sets of dependencies.

Bazel’s rules assume a similar sort of structure, except that the main workspace name is used instead of “google3”, and the main repository can refer to other repositories via “external workspaces”. The syntax “@external workspace//target” is used to refer to them in BUILD rules, and they are given top-level directories in runfiles tree. This runfiles structure is not always well suitable for external use cases.

By comparison, the “modern Python workflow” considers Python Distribution Packages to be the main unit of organization. A Distribution Package contains multiple programs (“scripts”) and Import Packages, but has only a single set of dependencies. Distribution Packages may also contain “extras”, which can declare their own sets of dependencies, but all such dependencies are eventually merged together into a common tree, so they cannot correctly have incompatible dependencies. Specifically, given “extra1” depends on “dep==1.0”, and “extra2” depends on “dep==2.0”, “pip install [extra1,extra2]” gives you some version picked arbitrarily.

Detailed Design

Bazel Python Configurations: py_toolchain()

Bazel uses files called CROSSTOOL to specify details of C compilation and linking. Inside Google, Blaze has a CROSSTOOL file pointing to version-controlled compilers and tools. Outside of Google, Bazel has CROSSTOOL files based on autodetection of system-installed compilers.

Bazel needs something analogous for Python configurations. Call it “py_toolchain”. A py_toolchain answers at least one of the following questions:

1. How do I run Python code?
    a. Filesystem path to an installed Python interpreter
    b. Shebang line to prepend to standalone scripts
    c. Required environment variables, command line flags, and/or library search paths
2. Can Bazel run this Python interpreter during the build process?
    a. Answer might be “No” for cross-compilation cases
3. How do I compile Python extension modules?
    a. References an existing CROSSTOOL/cc_toolchain for basic compilation functionality
    b. Filesystem path to required header files (“Python.h” etc)
    c. Filesystem path to required library files
    d. Compiler flags and other logic
4. How do I compile the Python interpreter itself?
    a. Lots of Bazel rules
    b. Checked in pyconfig.h files for relevant platforms as we don't run configure.

It’s expected that most py_toolchains won’t be able to answer all of these questions. A py_toolchain for a cross-compilation (e.g. x64 to ARM) can’t do #2. A Python interpreter installed without the required header files can’t do #3 (e.g. Debian with “python” installed but not “python-dev”). Any py_toolchain without laboriously handcrafted Bazel rules can’t do #4.

Points 1 and 2 seem reasonably straightforward. Point 3 could be done in several ways. We could go platform by platform (Windows, Mac OS, Linux, ...) and reverse-engineer “distutils” and the system-installed Python interpreter Makefile. This requires a fair bit of work, limits the platforms available, and only works for system-installed interpreters. Or we could invoke “distutils” at build time, and scrape the output. This seems likely to be fragile. We’ll probably have to do a bit of both.

Point 4 is what we do inside Google, and will continue to do. It’s quite labor intensive and high maintenance. We could write and maintain equivalent open-source rules for Python interpreters of interest.

Default Python Configurations

At or before the start of the build process, Bazel will create a set of default Python configurations for the system-installed Python interpreters on the build host. Bazel will use appropriate operating system facilities, such as querying the Windows registry, looking at $PATH, or directly probing the filesystem. Often, multiple interpreters are installed side by side, for example Debian 8 usually has Python 2.7.x and Python 3.4.x installed as /usr/bin/python2 and /usr/bin/python3.

Bazel may be invoked from inside a Python virtualenv. This can be detected by examining environment variables. If so, Bazel will create a Python configuration corresponding to this virtualenv.

Users will be able to define their own py_toolchain() targets in the WORKSPACE file to handle cross-compilation, non-standard interpreters, and any other desired cases. Syntax TBD, but reasonably complicated. Open question, do users need the ability to specify this per-user, or in a .bazelrc? Also, it would be nice to provide some facility to share these targets between workspaces.

The services provided by py_toolchain()s look a lot like the services provided by virtualenvs. Therefore, Bazel will create and maintain a virtualenv for each defined py_toolchain(). This is fairly easy (more symlinks) except for Windows. The virtualenvs will be stored in bazel-bin/<toolchain name>_virtualenv/.

System Installed Python Packages

Each Python interpreter has an unpredictable set of third-party Python packages installed alongside its standard library. These packages are implicitly available to all Python code run by that interpreter. While this might seem like a feature at first, it raises serious problems of correctness and reproducibility.

Having these packages implicitly available requires no work, it's just what happens by default. To use a Python interpreter without using third-party Python packages installed in it, one must do some work. Luckily, we decided to create a virtualenv for each py_toolchain, and virtualenvs already contain the functionality to separate a Python interpreter from unwanted third-party packages.

System Installed C Libraries

Each build machine has an unpredictable set of third-party C libraries installed system-wide. For reasons of practicality, most Bazel users will link their C/C++ code against these libraries instead of building absolutely everything from source. TODO A solution

Building extension modules via setup.py+distutils

For packages such as numpy that has an existing recipe for building extension modules via setup.py+distutil, Bazel has the following options:

1. Have Bazel run setup.py + distutils. Not hermetic, not reproducible, and not cross-compileable.

2. Create our own patched version of distutils that invokes Bazel instead of the system installed compilers. So we would have Bazel -> setup.py+distutils -> Bazel -> compiler. We'd do something clever to avoid actually invoking Bazel recursively. It's not clear how feasible this is, given the complexity of distutils, and the unlimited complexity of setup.py.

3. Have Bazel create a special sandbox that "looks like" the system-installed Python and system-installed compilers, but actually points to a particular py_toolchain+cc_toolchain. Then have Bazel run setup.py + distutils in this sandbox. It's not clear how feasible this is, but perhaps more so than #2.

4. Create hand-written Bazel rules to duplicate the setup.py functionality. We would need to create such rules for every Python package of interest (100s-1000s). Ideally these would be stored in a centralized place, and community curated and maintained. This would definitely work, since we do it inside Google, but is a large and ongoing maintenance and community support responsibility.

5. Don't handle this case: Download a pre-compiled binary wheel from PyPI, otherwise fail. Unfortunately, many packages don't have precompiled wheels, or don't have wheels for the platforms of interest. Also, there is a set of users (including Google) that aren't willing to download and run untrusted binaries from the Internet.

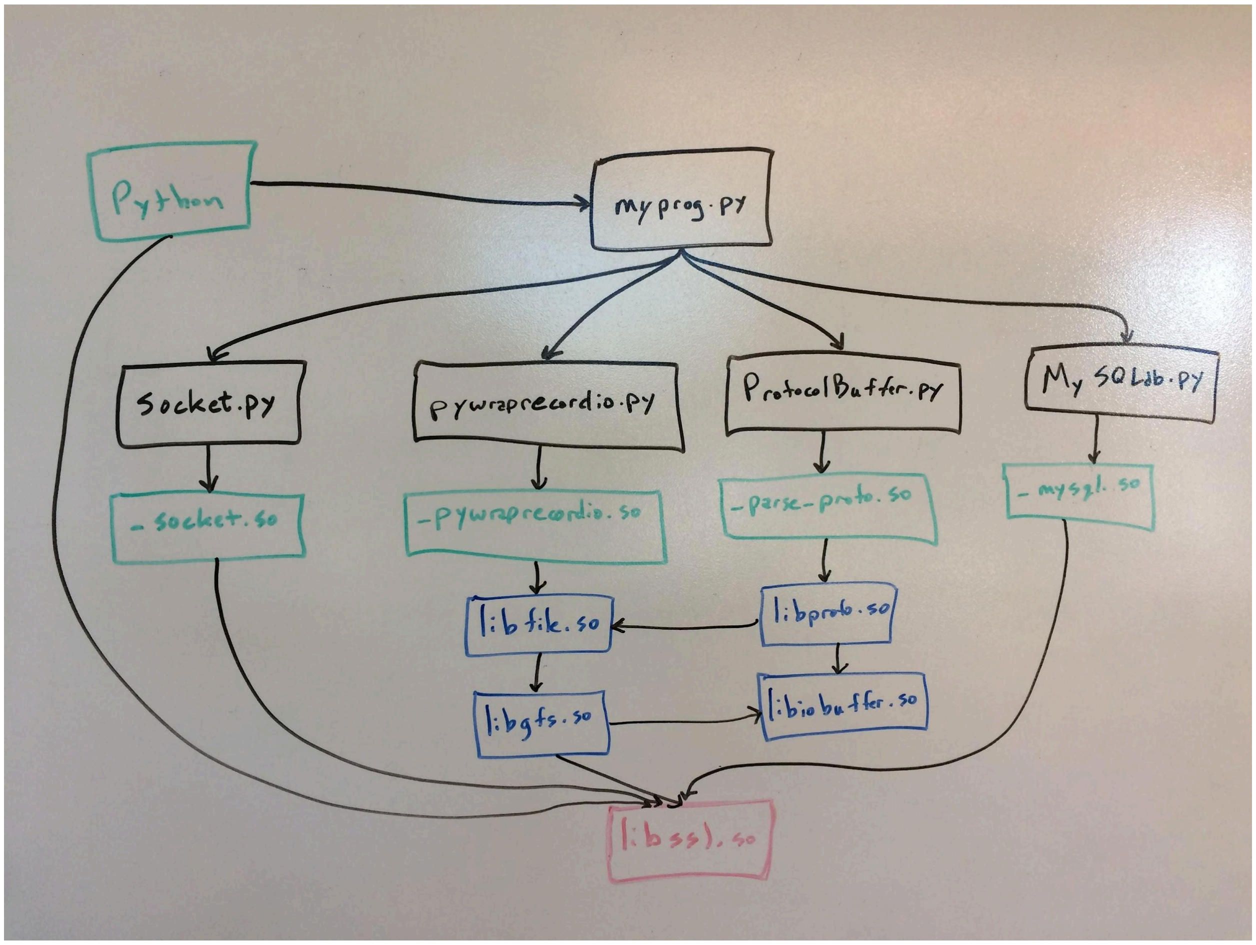
These are all hard choices.

Building extension modules directly in Bazel

A new BUILD rule will be added: py_extension(name, deps, srcs, ...). This rule will compile C/C++ source code to create an extension module in whatever format(s) is relevant. Bazel will use the relevant py_toolchain to figure out how to do this. Bazel may have to build an extension module more than once if multiple configurations are required. Extensions may depend on other cc_library() targets. Depending on or linking to other py_extension() targets is tricky, and will not be supported unless needed (alternatives such as the PyCapsule API exist).

The py_extension rule will provide some syntax for linking against a system-installed C library. For linking against a C library built by Bazel, the usual "deps" attribute will be used. When Bazel can detect link inconsistencies, it print a message and fail the build (i.e. linking against both libssl.a and libssl.so), but many cases can't be easily detected.

Dependencies will be linked as dynamic shared libraries, rather than statically linked into a single "swigdeps.so" as Google does. After building, it's largely up to the user to make sure that all the required shared libraries get to where they need to go for run time. Libraries such as openssl that are linked into the Python interpreter and/or standard library extension modules will continue to require attention and care from the user.



Packaging, PyPI, etc

BUILD Rule: py_distribution_package()

From a set of py_binary() and py_library() targets (default is all targets in the Bazel package), create a Distribution Package suitable for uploading to PyPI. Generates a suitable setup.py file. All py_binary() targets become 'scripts'. All py_library() targets other than the main one become 'extras'. All dependencies are translated from Bazel syntax to PyPI syntax.

TBD

WORKSPACE Rule: py_pypi_dependency()

Describe a Distribution Package on PyPI (or local mirror, or vendored directory tree). Such a package is then available as a "dep" for py_binary(), py_library(), and/or py_distribution_package() targets. Includes version resolution.

TBD

Runfiles 2.0

Reorganize the layout of runfiles directory tree to look more like a "site-packages" directory.

More ambitiously, reorganize the runfiles directory tree to look like a "virtualenv" (e.g. with bin/ and similar directories, and links to the Python interpeter).

Add per-target option to use Python .pth files to dynamically join multiple partial runfiles directories, at runtime, rather than creating millions of near-duplicate symlink trees.

TBD

Workspace Layout - Monorepo vs Manyrepo

Handle both cases in a reasonable way. Currently, the runfiles trees for the two cases have a very different structure. They shouldn't.

TBD

Security Considerations

Downloading source and/or compiled binaries from PyPI opens major new attack surfaces.

Privacy Considerations

Connecting to PyPI to download source and/or compiled binaries leaks information about the code being built, the machines building it, and the users and organization working on it.

Testing Plan

A local PyPI mirror/mock for system tests.

Work Estimates

???

Document History

You can view a description of this section here.

Google Docs has built-in document revision history. Select “File > See revision history” from the menu to view detailed document history.

Whenever you add a significant new revision to the document, add a new line to the table below.

Date	Author	Description	Reviewed by	Signed off by
2018-02-05	dgreiman	Initial Draft	twouters, gps, ...	