Gistable: Evaluating the Executability of Python Code Snippets on GitHub

Eric Horton, Chris Parnin
NC State University
Raleigh, NC, USA
Email: {ewhorton, cjparnin}@ncsu.edu

***Abstract*—Software developers create and share code online to demonstrate programming language concepts and programming tasks. Code snippets can be a useful way to explain and demonstrate a programming concept, but may not always be directly executable. A code snippet can contain parse errors, or fail to execute if the environment contains unmet dependencies.**

This paper presents an empirical analysis of the executable status of Python code snippets shared through the GitHub gist system, and the ability of developers familiar with software configuration to correctly configure and run them. We find that 75.6% of gists require non-trivial configuration to overcome missing dependencies, configuration files, reliance on a specific operating system, or some other environment configuration. Our study also suggests the natural assumption developers make about resource names when resolving configuration errors is correct less than half the time.

We also present Gistable, a database and extensible framework built on GitHub's gist system, which provides executable code snippets to enable reproducible studies in software engineering. Gistable contains 10,259 code snippets, approximately 5,000 with a Dockerfile to configure and execute them without import error. Gistable is publicly available at https://github.com/gistable/gistable.

I. Introduction

Online programming communities such as Stack Overflow and GitHub facilitate social learning of programming and API concepts. One common learning mechanism is to share code snippets or examples, which contain explanations and demonstrate how to perform a programming task or use an API [1]. Code snippets are often reused and incorporated in open source projects [2]. Currently, GitHub provides the ability to create and share code snippets (called *gists*), with over 300k Python gists, and over 4.5 million gists in multiple programming languages.

This work focuses on evaluating the excitability of publicly available Python scripts hosted on GitHub's gist system in the context of software configuration management (the process of configuring system environments to properly execute a software program). We seek to categorize the common reasons for why gists cannot be executed in a default environment and motivate further research on automated software configuration management by identifying what difficulties exist in properly configuring an environment to enable gist execution, specifically with regard to installing application dependencies. Our work also provides a dataset of gists known to not be executable and a baseline analysis against which to compare future configuration methods.

We start by highlighting a method for performing automated gist collection and analysis. The process involves scraping gist URLs from the GitHub gist UI. This technique led to an initial dataset of 10,259 gists containing over 1,700 unique third-party library packages. We then cloned each gist and executed it inside of a Docker container based on the official Python image for Docker, categorizing the gist by its exit status. To evaluate gist configuration, we attempted to infer a correct environment specification using a naive algorithm that approximates the first steps human developers often take by attempting to install third-party packages by the name of the resource imported within the gist. After running the naive inference algorithm, we then reevaluated the executable status of the gist.

Our findings show that correct dependency resolution and environment configuration are often required even for small programs. Less than 25% of gists were executable by default, with over half failing due to `ImportError` in Python 2. Of the gists which initially failed with `ImportError`, our naive inference algorithm could successfully infer an environment specification less than 50% of the time.

To gain a better understanding of the why the naive inference algorithm fails, we asked 24 developers familiar with system configuration practices to create a Dockerfile for 10 unique gists, assigned to them at random, for which inference failed to resolve import errors. We then used the produced Dockerfiles and feedback from the developers to categorize gists, focusing on the first cause of failure if any gist would have failed inference due to more than one reason. The most common cause is that the names of resources used in a gist do not necessarily match the names of the packages they belong to. Gists also frequently fail due to missing transitive dependencies, missing system dependencies, configuration files, and deprecated or non-standard packages.

Finally, we present Gistable, an extensible database and framework used to perform our mining and analysis. Gistable also contains the ˜5k gists with environment specifications which allow them to be run without `ImportError`. We believe that several areas of software engineering research can benefit from a database of executable code snippets, such as: automatic code summarization, testing, and API usage analysis.

In summary, this paper makes the following contributions:

- An empirical analysis on the executable status of Python

gists on GitHub.
- A qualitative analysis of reasons why code may not be executable.
- An extensible mining framework, Gistable, for obtaining gists and environment containers from our gist database.

II. Motivation

Code snippets are not always directly usable [3]: They can contain parse errors or require system dependencies unmet in a programming environment. As a result, the following challenge emerges: *Given a code snippet, successfully infer the environment configuration necessary for execution.* Frequently, developers must perform this inference step manually, or rely on the creation of configuration scripts, which in itself is a time consuming task [4]. Unfortunately, it is not always clear what dependencies or environment configurations are required to execute code. Consider the following Python code snippet.

```
# Import modules from networkx and matplotlib
from networkx.drawing.nx_agraph import
  ↪ graphviz_layout
import matplotlib.pyplot as plot
import networkx as nx

# Generate the complete graph on five vertices
k5 = nx.complete_graph(5)

# Draw using layout generated by graphviz
plot.figure()
nx.draw(k5, graphviz_layout(k5, prog="neato"))
plot.savefig('/output/graph.png')
plot.close()
```

To successfully run this code fragment, several requirements must be met. First, the environment requires graphviz, which is a tool for visualizing graphs. Second, the environment needs to install the Python bindings for graphviz. Third, the environment needs the Python package for matplotlib and networkx. Fourth, an environment variable, `MPLBACKEND`, may be needed to specify a rendering engine that is compatible with a headless VM, which does not have a graphics display. Finally, the environment needs to ensure that an `/output` directory exists.

These requirements can also be encapsulated by a working environment configuration. One system that can be used for specifying environment configuration is the containerization system Docker. Docker configuration is centered around the Dockerfile, a configuration script which tells the Docker engine how to properly build an image that can be distributed and run by others. We present a Dockerfile for the snippet below.

```
FROM python:2.7.13
VOLUME /output
ENV MPLBACKEND Agg
RUN apt-get update
RUN apt-get install -y graphviz
RUN pip install pygraphviz
RUN pip install matplotlib
RUN pip install networkx
ADD snippet.py /snippets/
CMD python /scripts/snippet.py
```

III. Gistable Dataset and Tool

Gistable is a framework for collecting, evaluating and executing self-contained programming code snippets, called gists. The name is derived from a portmanteau of the words *gists* and *runnable*. Gistable is designed to support empirical research for a variety of software engineering tasks. Gistable can mine code snippets and automatically generate a Dockerfile which can be used to run the code snippet. Gistable provides a command line interface for performing tasks with the mined gists, such as checking out snippets into a working directory, and executing the code snippet inside a docker container.

A. Research context.

Our initial evaluation of Gistable focuses on Python gists. Python is a popular programming language and ranks among the fastest growing languages today. It follows only Ruby and Javascript in proportion of files in public gists [5]. Python is frequently used for teaching introductory programming classes as well as used by non-professional programmers, such as scientists.

Previous research by Yang et al. [3] examined Python snippets on Stack Overflow and found that only 25% were runnable (but did not investigate why). In this work, we focus on examining gists shared on GitHub instead of Stack Overflow. As observed by Sillito et al. [1] and Yang et al. [3], code snippets on Stack Overflow are often mixed with exposition and code, making it difficult to understand which segments of code are meant to be executed in an automated analysis. Therefore, there is strong motivation to investigate the underlying reasons why Python code may not be executable and understand the effort involved in configuring environments capable of running it. These barriers can cause problems for learners and non-professionals programmers lacking system configuration skills.

B. Mining Gists

We consider two strategies for mining gists from GitHub. GitHub provides a REST API for public gists, however, there are several limitations. Currently, the API provides no support for filtering queries based on language type. Furthermore, the API limits requests to 3000 gists when using pagination. To overcome these limitations, it is possible to filter gists based on creation date, meaning that all gists could be slowly enumerated by strategically modifying the creation date as a filter.

Another strategy is to scrape gists from the GitHub gist search UI. The search UI allows several filters, such as star rating, language, and keywords contained in the gist. The UI returns at most 100 pages of 10 random gists matching a search, allowing 1000 gists to be returned per search. By strategically modifying search terms, it is possible to quickly discover gists that meet the desired criteria.

For our initial population of the Gistable database, we focused on the scraping approach, which allowed us to focus on a particular language and to better control the quality of gists while using less computational resources.

```python
import requests
import json

urlbase = 'http://maps.googleapis.com/maps/api/geocode/json?sensor=false&address='
urlend = 'Zurich,Switzerland'

r = requests.get(urlbase+urlend) # request to google maps api

r=r.json()
if r.get('results'):
    for results in r.get('results'):
        latlong = results.get('geometry','').get('location','')
        latitude = latlong.get('lat','')
        longitude = latlong.get('lng','')
        break
    print latitude, longitude

else:
    print 'No results'
```

(a) Gist 10017416

```
FROM python:2.7.13
ADD snippet.py snippet.py
RUN ["pip", "install", "requests"]
CMD ["python", "snippet.py"]
```

(b) Dockerfile

Fig. 1: (a) Code snippet for using the Google Maps geocode API. (b) Dockerfile containing environment specification required to run code snippet.

C. Environment Inference Algorithm

To perform environment inference, we use an approach which builds an Abstract Syntax Tree (AST) of the gist source code and extracts all declared imports. Extracted imports are then filtered to remove all packages which are part of the Python standard library. Imports are assumed to be part of the standard library if they are present in a Docker image containing a clean install of the Python runtime.

We use the assumption that each import represents a single package that needs to be installed, and that the import name matches the name used to install the package. This is not always the case. For example, the Python package `beautifulsoup4` is imported as `bs4`. However, developer practices from Section IV-C3 suggest that this is a useful approximation because it is the natural first step a developer takes when attempting to configure a computing environment. Errors from packages which could not be found are ignored. Such packages are simply not included in the final environment configuration. This allows us to recover from potential errors in our inference algorithm.

D. Execution Harness

To deal with the large number of gists analyzed as part of the Gistable database, we built an execution harness on a distributed cluster using the HashiCorp Nomad job scheduler, which natively supports docker containers. The harness is responsible for running all gists through the validation process to first determine if environment inference is needed and categorize the result of gist execution.

To isolate effects of dependencies and other system wide configurations, we perform analysis inside independent Docker containers. The container filesystem also guarantees consistent starting environments.

E. Using Gistable

Gistable provides a command line tool for interacting with gists from the Gistable database. Gists can be cloned into a specified directory using the command `gistable clone <id> [location]`. Behavior is similar to that of `git clone`, and gists are checked out to the working directory if no location is specified.

If Docker is installed and running on the system, the CLI can also be used to directly execute a gist and display all execution results. Just call `gistable run <id>`.

IV. Methodology

A. Research Questions

In this study, we investigate the following research questions and offer the motivation for each:

RQ1 – Can gists be executed? Can the average Python gist on GitHub be run to completion, or will it raise an exception? If gists can be run to completion, then they already form a database of snippets that can be used in research. However, if, like the Python snippets from [2], gists cannot be executed by default due to syntax errors or other runtime exceptions, then additional investigation is needed.

RQ2 – Can a naive algorithm enable exectuable gists? Can we apply a simple approach for resolving unmet Python dependencies to address most runtime exceptions? If a majority of errors can be addressed by a simple resolution strategy, then there are a limited number of cases where automated environment configuration is needed. However, if a simple approach cannot be used, then more research is needed for developing a more comprehensive automatic environment configuration technique.

RQ3 – Why might gists not be executable? If gists cannot be executed even after resolving package dependencies, the

natural question is why. Are they missing configuration for environment variables, services, or other kinds of dependencies? Categorizing gist execution failure and finding common root causes may lead to insight into how to improve future automatic environment configuration techniques.

B. Data Collection

To address our research questions, we first focused on building a large dataset of Python gists. We used the mining procedure outlined in Section III-B to mine 10,259 Python gists. We limited our search criteria to gists with at least one star [6]. Currently, GitHub contains 32,233 Python gists with at least star—meaning our sample represents nearly 31% of all public starred Python gists.

Figure 1 illustrates an example of a gist in our experimental dataset and its accompanying automatically created Dockerfile. The gist uses the Google Maps geocode api to retrieve the latitude and longtitude coordinates of Zurich, Switzerland. The Dockerfile bases the image off of a Python environment, adds the gist code file, installs `requests`, and configures the default command to run the gist. Note that the package, `json`, does not need to be installed as it is a default system package.

C. Analysis

To answer our research questions, we used the following procedures to analyze our data. The inference harness described in Section III-D was used to clone gists from GitHub and perform analysis. Using two `ubuntu-16-04-x64` worker nodes sized at `2gb` and running in Digital-Ocean, inference took approximately eight hours to schedule and run all jobs.

1) RQ1: To answer RQ1, we start by performing a baseline analysis of gists by attempting to execute them in isolated Docker containers based on the `python:2.7.13` and `python:3.6.5` images. Any gist which executed without error is considered to have exited with the code `Success`. Any non successful gist is coded by the name of the error which was raised. I.E., `SyntaxError`, `ImportError`, `NameError`, etc.

2) RQ2: Research from Becker et al. [7] indicates that the practical approach when there are multiple failures is to focus on the first error until it is resolved, then move on. This follows from the observation that first failures are useful because they are informative, need to be fixed, and their resolution may reveal deeper errors that were not apparent before.

To answer RQ2, we focus on gists where the first encountered failure was an `ImportError` and ask if we can configure the environment with all necessary dependencies. A naive attempt is made at performing environment inference by applying the inference procedure described in Section III-C. We attempt to install each inferred package with the Python package manager `pip`. This is based on our findings from Section IV-C3, which showed that attempting to install a resource name listed in an import error is often the first step developers take when attempting to fix environment configurations.

TABLE I: Gists per exit code in the baseline evaluation using Python 2.7.13.

Result	Count	Percent
ImportError	5379	52.4%
Success	2501	24.4%
NameError	852	8.3%
SyntaxError	753	7.3%
IOError	167	1.6%
IndentationError	153	1.5%
SystemExit	115	1.1%
EOFError	94	0.9%
OSError	48	0.5%
ValueError	34	0.3%

After applying our inference algorithm, the gist is then executed a second time with the new environment specification, and the evaluation results recorded under the same criteria as for the baseline.

3) RQ3: We performed a random sampling on failing gists in order to understand why they failed to execute. For this analysis, we performed descriptive coding [8] and composed *memos* [9], which described several reasons for a gist failing to execute. These memos captured interesting events or properties of environments and code snippets to promote depth and credibility, and to frame the information needs of an automated environment configuration technique. That is, they provide a *thick description* to contextualize the findings [10].

We then solicited 24 developers familiar with Docker to manually inspect gists. Each developer was given a disjoint random set of 10 gists and asked to create a Dockerfile that would enable successful execution of the snippet within a standard time period (one and a half weeks). The developers had between 6 months to 5 years of industry experience and familiarity with Python. Further, the developers had been trained in several workshops on configuration management skills, including Ansible and Docker.

We asked the developers to rate the difficulty of creating a Dockerfile and the steps they took to create it. We then performed a qualitative coding exercise over the Dockerfiles and reported steps using closed codes derived from our first qualitative coding. During the coding process, we employed the technique of *negotiated agreement* as a means to address the reliability of coding [11]. Using this technique, the first and second authors collaboratively code to achieve agreement and to clarify the definitions of the codes; thus, measures such as inter-rater agreement are not applicable.

V. Executability Results

A. RQ1 – Can gists be executed?

Table I provides the names and counts for the most common reasons a gist terminated when run in an isolated Python `v2.7.13` environment.

Consistent with the Yang et al. [3] study on Stack Overflow Python snippets, we observed that only 24.4% of Python gists were executable. The majority of gists (52.4%) failed to execute due to an `ImportError`, which is typically caused when a python dependency could not resolved or loaded. We observed that only 17.1% of gists failed to parse (i.e.,

`SyntaxError`, `NameError`, and `IndentationError`. Our observed rate of parse failures for gists is slightly lower when compared with Yang et al.'s observed rate of 25% for Stack Overflow snippets. We believe this may be caused by the difficulty of distinguishing exposition from code when parsing code snippets found on Stack Overflow [1]. For example, in a Stack Overflow post, it could be common to include code and output typed into an interactive shell in order to help explain a concept, which is not directly parsable.

Finally, we observed <8% of gists failed to execute due to some other runtime exceptions, such as `IOError` or `OSError`. These failures could be caused by missing resources, such as files, services, or platform specific dependencies.

Baseline results for executing in a Python 3 environment show 3,907 instances of `SyntaxError`, compared to the 753 for Python 2. In addition, the number of gists which exited with `Success` dropped to 1,445. The number of gists which exited with `ModuleNotFoundError`, a direct subclass of `ImportError` in Python 3.6, was 3,353. While this shows a decrease from the 5,379 in Python 2, the large set of `SyntaxError` may shadow an undetermined set of gists which would also see an `ImportError`.

Overall, we find that most gists are not executable in a default Python environment. Further, the exceptions raised when attempting to execute the gists suggests that an insufficiently configured environment is the primary cause.

B. RQ2 – Can a naive algorithm enable exectuable gists?

The baseline analysis for RQ1 showed the majority of Python gists require environment configuration. To determine if a simple algorithm is capable of resolving such errors, we applied our inference algorithm described in Section III-C to the 5,379 gists which failed due to `ImportError` using Python 2, attempting to install all third party imports with `pip` in both a Python 2.7.13 and Python 3.6.5 environment. Python 2 is used as a baseline for `ImportError` due to its lower frequency of `SyntaxError`.

We analyzed each gist after attempting to install all inferred dependencies and recorded the exit status according to the same criteria used for answering RQ2. For Python 2, 2,488 gists exited due to a reason other than `ImportError`, a gain of approximately 46%. Of these gists, 1,294 finished with `Success`. The remaining 1,194 finished with some error other than `ImportError`. When also considering Python 3, the number of gists which had become executable increased to 2,870. Overall, considering Python 3 resulted in an additional 428 gists becoming executable after inference when compared with only using Python 2.

While a naive approach can infer dependencies for some gists, it fails to do so in the majority case.

VI. Execution Failures

To answer, *RQ3 – Why might gists not be executable?*, we inspected the gists to better understand why they failed to execute, even after applying our naive algorithm. First, we focused on gists failing with `ImportError`, which was the most common failure status. Then, we also inspected gists which failed for other reasons, such as `IOError`. Finally, we characterize the effort reported by developers when manually creating Dockerfiles for the failing gists.

A. Gists Failing with ImportError

We report our findings in Table II. Overall, the 24 developers participating in this study were able to submit a response for 218 out of the 240 gists assigned to them as a group. The average number of Dockerfiles received from each developer was 9, with a minimum of 3 and a maximum of 10.

In addition to the failures reported in Table II, 24 gists were considered flaky. Inference of flaky gists may have failed due to network or memory issues. One developer reported needing to increase the memory Docker was configured to use in order to properly install dependencies for one such gist.

Collectively, the developers indicated that they were unsuccessful in creating a working Dockerfile for an additional 78 gists. The feedback we gathered for such gists showed that even developers familiar with environment configuration may be unable to correctly deduce the correct specification for an arbitrary snippet of code. One developer, after referring to an existing Dockerfile related to the gist they were working with, wrote

> *I attempted to adapt the Dockerfile listed above to run this gist, but was never able to get it working; needless to say I would not have been able to do it without the Dockerfile listed either; I attempted various other ways to install the android sdk (apt-get, etc), all of which failed; constantly ran into 404 errors with apt-add-repository; got "No space left on device" error when running listed Dockerfile in a virtual machine; the Dockerfile built when running natively, but I could not find a way to use the "monkeyrunner" command, as this gist is supposed to be run with "monkeyrunner" and not "python" (from what I understand); a great deal of time spent trying futilely to get this to work.*

We now focus on a selection of distinct failure causes.

Names. The most common case, as stated in Section V-B, is when a resource name does not match the name of the package it belongs to. Resolving this situation often required the developer to search the package index, test multiple packages, or query developer resources such as Stack Overflow.

For example, one gist relied on the module named `i3`, but the developer found they had to install the package `i3-py`, resulting in the following Dockerfile:

```
1 FROM python:2.7.13
2 ADD i3_focus_win.py /
3 RUN pip install i3-py argparse
4 CMD ["python", "/i3_focus_win.py"]
```

TABLE II: We had 24 developers familiar with environment configuration techniques attempt to manually create Dockerfiles for 218 of the gists for which naive inference failed to resolve import errors. This table summarizes reasons for failure as reported by the developers, focusing on the first failure reported. We manually inspected each gist in cases where no clear reason was found by a developer, applying our own failure category if possible, or labeling the gist as unconfirmed.

Cause	Count	Example
Package name did not match the resource imported in the gist	70	https://gist.github.com/syl20bnr/6623972
Gist dependencies have additional dependencies which need to be resolved	23	https://gist.github.com/kennethreitz/2901479
Relies on missing C library files or headers	16	https://gist.github.com/huyx/8069261
Requires a previous version of a package due to breaking changes	15	https://gist.github.com/segphault/9f2d7da68779a17a0890
Dependency can only be installed on a non-linux operating system	13	https://gist.github.com/mapleray/4189391
Relies on a standard package that was introduced in a later version	12	https://gist.github.com/fmasanori/4684752
Pip errored during installation, possibly timing out on large packages or propagating an exception raised by the package	12	https://gist.github.com/willwade/5330566
Unconfirmed. The exact failure could not be narrowed down to a single category.	9	
Gist is missing necessary environment configuration, such as settings files	8	https://gist.github.com/Sinkler/bfc2099235ac96937f34
Dependency wasn't available on PyPI, nor installable via the Ubuntu aptitude package manager.	7	https://gist.github.com/JudoWill/764262
Dependency is only supplied as part of a custom execution environment or interpreter	6	https://gist.github.com/Utopiah/a2b9c6ecdb24ca8fd6f4f41a9c0eb32e
Relies on a deprecated package that is no longer maintained and is no longer available to be installed	1	https://gist.github.com/matbor/6532185
Gist is not intended to be run and imports libraries which don't exist	1	https://gist.github.com/RichardBronosky/454964087739a449da04
No versions are available for install	1	https://gist.github.com/mclavan/276a2b26cab5bc22d882

System dependencies. Missing C libraries were also a common issue. Many Python dependencies serve as bindings into C libraries installed as a system dependency, and fail to compile on installation because the system dependency is not present. In some cases, a dependency failed to compile because the Python Docker image did not include C build tools, such as cmake, that they relied on.

One such gist made use of the Python `hunspell` package, a wrapper for the C program Hunspell. The developer found that before using pip to install `hunspell`, they needed to add `RUN apt-get install libhunspell-dev -y` to their Dockerfile.

Custom environments. In some cases, a dependency was distributed as part of a separate execution environment. For example, one developer reported that a gist relied on the `bpy` module that ships with Blender. After installing Blender and still seeing an `ImportError`, the developer discovered a Stack Overflow post saying `bpy` can only be imported when running in Blender's bundled Python interpreter.

Unlisted packages. Several gists depended on packages which were not available through the PyPI or Aptitude package managers by default. Such packages require being installed from a separate repo, such as an Aptitude Personal Package Archive (PPA) or directly from a git based public repo.

In one example, a user commented on the Gist that they had difficulty importing one of the modules, even though they had installed the correct package.

> *ScissorPush?*
>
> *from kivy.graphics import ScissorPush ImportError: cannot import name ScissorPush*

Resolving this issue required installing an unreleased version of python-kivy that needed to be installed from a PPA.

```
FROM ubuntu:16.04
RUN apt-get update
RUN apt-get install -y software-properties-common
  ↪  python-software-properties
RUN add-apt-repository ppa:kivy-team/kivy-daily
RUN apt-get update
RUN apt-get install -y python-kivy
ADD snippet.py /snippet.py
CMD ["python", "/snippet.py"]
```

Deprecated packages. In other cases, gists relied on packages that are no longer maintained and can no longer be installed. Common causes are not supporting SSL, which pip now requires, not fixing known bugs which prevent installation, or even an entire package no longer being provided for distribution.

For example, the Python Quartz package has an omission in the manifest that prevents the requirements file from shipping with the package source. The developer is aware of the issue, but has declared they will not create a patch.

> *To fix this problem, I have to include requirements.txt in MANIFEST.in so that the file will be shipped with the sources.*
> *Unfortunately, I abandoned this project a while ago and I am currently working on a complete rewrite...*

Sometimes, a package is still actively maintained, but the

gist relies on features from a version which had reached end-of-life and is no longer being distributed.

Configuration settings. Some gists require additional configuration files which are not provided with the gist itself. For example, it was common to read in secret keys and values from a non-existing app.config file in order to read a setting such as `TWITTER_API_KEY`. These configuration files are not preexisting dependencies which can be installed.

Language version. Python 3 has introduced several new modules, like `urllib.request`, that are not present in Python 2. Gists that rely on these modules must be run in a Python 3 environment, and are incompatible with the `python:2.7.13` Docker image being used. In some cases it may still be challenging to determine which Python version to use. For example, `pathlib` is a part of the Python 3 standard library, but was not introduced until Python 3.4, and support for it was only added to the standard library in Python 3.6.

Operating System. Developers also saw dependencies which could only be installed on a specific operating system, such as Windows or macOS. One developer, when asked to create a configuration for a gist, found that the gist was designed to interact with the Windows registry, and reported

Packages are dependent on Windows (not Ubuntu).

Such gists cannot be run in the Ubuntu based Python image.

B. Other Failing Gists

To characterize the gists in our dataset and gain a better understanding of how they are used on GitHub, we computed basic metrics across all gists using tools developed for our execution harness. Additionally, we performed an inspection on 30 randomly selected gists from the 10.3k in our dataset with the focus on characterizing what resources they might rely on, including, but not limited to, dependencies.

Our random sample found that 14 out of 30 gists (46%) did not rely on a third party package. Approximately 13% did not import any packages, and 76.7% relied on Python library packages. 6.7% optionally loaded a third party package if it existed in the environment. We found that many gists rely on connecting to networked resources, or on interacting with configuration files and executables on the file system. Other gists required interaction from the user in some manner, either requiring input over `stdin`, command line arguments, creating an interactive prompt, or displaying information through a graphics interface. In the worst case, a gist does nothing because it is either recognizably not correct Python syntax, or because it defined classes or functions but did not otherwise execute code. This happened nearly 10% of the time.

Overall, the gists in our dataset import over 1,700 unique third party packages and on average have 92 lines of code.

C. Developer Extraction Effort and Effectiveness

The median difficulty rating reported for configuring a gist was 3 on a scale of 1-5, reported for 24.3% of all gists. Only 13.7% of the gists were reported as very easy to execute by our developers, whereas 22.4% were reported as very difficult to execute. Developers reported spending between 20 minutes to 2 hours to setup the environment for executing each gist.

Of the 140 gists developers found an environment configuration for, the average Dockerfile was less than 10 lines and installed less than 5 packages. However, we found that not all of the submitted Dockerfiles were capable of executing their gists without `ImportError`. For example, one developer submitted the following Dockerfile, claiming that the gist ran without any errors in the provided environment.

```
1 FROM python:2.7.13
2 ADD https://gist.githubusercontent.com/⌋
    ↪ awesomebytes/⌋
    ↪ cb5a28fa8d4db3fc1ba51894663c1aed/raw/⌋
    ↪ cba597a5219d807c5e4940e9d2018d47b5eca809/⌋
    ↪ watson_ros_publish_string
    ↪ /snippet8.py
3 RUN pip install ws4py
4 CMD ["python","/snippet8.py"]
```

However, we found that executing the gist still failed with the configuration error `ImportError: No module named rospy`. One interpretation is that not only can this be a time-consuming task for developers, but the process can be also error-prone.

D. Developer Responses

Section VI revealed common properties of gists that made environment configuration difficult. We now highlight a selection of developer responses which illuminate the process that developers employ when faced with these challenges.

Version errors. Developers reported several experiences related to resolving errors that were present due to mismatches in versions of dependencies and code.

> *django was the only import required. But that didnt simply resolve the error. There was a import error for CompatCookie. Tried in python 3 as well but no luck. Later found out from django release notes that it was deprecated after v1.4. So tried to pip install older version of django and was finally able to resolve the import error. Docker file builds and runs without any error.*

Another developer described how the requirements could shift depending on the version of a dependency used.

> *Spent over an hour to find the imports needed for text.blob. It was replaced to textblob from version 0.7.1 and when I tried the lower version I received another error that required dependencies on a higher version.*

Unlisted or unknown dependencies. Developers reported several instances where they had difficulty determining the provenance of a dependency.

> *1. Git clone basic-python-logger repo from https://github.com/vehrka/basic-python-logger (Have to google and find out that basiclogger.py is not a module, but rather a script wrote by the creator of the Gist itself) 2. pip install psycopg2, pandas, sqlalchemy for satisfying the dependencies. 3. Upon doing this, the images builds successfully*

Another developer had difficult working with a cloud provider package.

> *Couldn't find clouddns module. Couldn't solve dependency. Spent 2 hours on it.*

Resource limitations. Several developers reported experiences related to memory or disk limitations on their personal computers when building environments.

> *MEMORY ERROR while installing keras and pyspark resolved by –no-cache-dir flag*

E. Summary

We conclude RQ3 with the following observation:

> Python gists often require non-trivial environment configuration in order to run. There are multiple reasons why configuration for any particular gist might be difficult, but the most common challenges are finding dependencies without obvious names and installing dependencies with transitive dependence on system modules.

VII. Discussion

A. Towards automated environment configuration and beyond

While the inference procedures presented in Gistable are simple, we show that they successfully lead to a correct environment specification in a number of cases. This indicates that reliable environment inference is possible, and highlights areas of research where techniques can improve. For example, we may consider combining dynamic inspection of packages and machine learning algorithms for inferring possible environment specifications.

Although we focus on Python gists, we believe our insights can generalize to other programming languages. For example, dependency on system build tools can also be a problem in the Node.js ecosystem when packages compile native addons. Common compilation troubles in Node eventually prompted Microsoft to publish developer guidelines [1].

Given a language which supports third-party packages, our approach only assumes two things. First, that packages have a set of named resources that they make available for use by client code. Second, that the identifier for a package resource used by client code has at least a substring match for a resource provided by the package. This is the case for popular languages like Javascript and Ruby, and so we believe that our approach will generalize to these, and similar, languages.

Furthermore, our inference procedure only requires a code snippet, and could easily be modified to work with another context, such as code snippets in Stack Overflow answers, blog posts [12], or online documentation [13]. Gistable focuses on configuring and running single file scripts. However, many projects have a large number of interacting tools that make configuration challenging. We believe insights from our work can inform configuration techniques for larger projects in the long-term.

[1] https://github.com/Microsoft/nodejs-guidelines

B. Challenges in mining gists

Querying for unique gists isn't directly possible: Instead, we rely on manipulation of search parameters in the GitHub UI to return results. On subsequent runs, the gists returned by a UI search are often different, allowing the use of a dictionary approach for collecting unique gists by ID. However, some gists may still be duplicates, either by forking or simple duplication of content. Forked gists have metadata available indicating the origin, but, in the worst case, it is generally undecidable if two gists are equivalent.

Gists can be complex: While most of them are relatively simple, there is no requirement that a gist consist of only one file, or even of files in a single programming language. If a gist has more than one file, the entry point is often ambiguous, unless the programming language runtime supports running a default file, and such a file exists in the gist. We discarded gists with more than one file to avoid having to deal with this situation.

C. Challenges in automated configuration inference

There are several challenges identified by our work.

Name resolution: An important task in automatically creating an environment specification from code is: *given a code snippet, infer the set of installable packages associated with the code*. Luckily, package import statements within the code snippet can help; however, there are still several complications that must be resolved. In the simplest case, many package names may not match the name they are imported by (e.g. the `i3`/`i3-py` mismatch encountered by one of the developers participating in the study).

Another consideration is that many gists have imports structured as follows: `import kazoo.client`. In our evaluation, the naive algorithm attempts to install `kazoo.client`, and fails. The actual package is `kazoo`. However, in other cases, like `zope.interface`, the appropriate package name is indeed `zope.interface`. Finally, it could be possible that some code snippets are incomplete; that is, they may omit import statements for packages being used in code.

A first step to addressing this challenge may be to preprocess known packages by extracting a list of resources that each exports. When performing inference, resources might be mapped to installable packages by a reverse look up. However, this introduces its own challenge of dealing with packages which have conflicting resource names.

System Dependencies: Other packages have implicit dependence on system environment configuration or other system packages. Unfortunately, this type of error often presents itself as a compile time error when a header file cannot be found. Header files, like package resources, do not necessarily have a name related to their project. Like the Python package name resolution challenge, this could be addressed by preprocessing and reverse look up. Another option is to analyze existing configuration scripts for Python projects and perform association rule mining to infer dependence between a Python package and a system dependency.

Language Version: Even with a gist consisting of a single snippet in the desired language, it is often non-trivial to decide which language version to use. For our Python gists, most of them are capable of running successfully with Python 2. However, reported instances of `SyntaxError` may be due to use of syntax created in Python 3. Gists can be checked for syntax errors by attempting to compile them, so Python 2 or Python 3 syntax compatibility could be checked by compiling under each language runtime. Python 3 dependence may also be inferred by checking gist imports against the Python 3 standard libarary.

Unlisted and Deprecated Packages: Packages may not be installable from a general package repository, like the Python package PyTorch. In this case, it may be possible to install directly from a git based repo, if one can be inferred from previously seen configuration scripts.

D. Future Applications

While the focus of our paper was evaluating the executability of Python gists, we envision several additional research applications for Gistable.

- **Text summarization of tasks:** Recent work ([14], [15], [16], [17]) has focused on performing semantic code summarization. Because gists typically correspond to idiomatic programming uses and tasks, there is an opportunity to use gists as a dataset for learning models which support semantic summarization of code.
- **API usage analysis:** We observed over 1,700 unique third-party python packages in our initial version of Gistable. This suggests that gists can provide a rich source of information for mining and understanding how APIs are used in practice by programmers.
- **Test input generation:** Gists often have hard-coded input text for running the code example. An interesting research opportunity would be to use gists as a benchmark for generating test inputs that can also successfully run (or fail) in a gists.
- **Configuration repair:** In addition to inference of configuration environments, it is possible to support research in repair of configuration scripts [18]. For example, if a user updates the code to use new library, pushes changes to production, but did not update the Dockerfile [19], an exception can be thrown. However, a configuration repair tool can suggest a repair that updates the environment specification to use the correct version of the package: e.g., `"networkx==2.0"`, which eliminates the exception.
- **Resource repair:** Gists may have external resources, such as URLs, or publicly hosted APIs, such as the Google Maps API. One interesting application would be to study the decay of resources overtime (bitrot). Further, if resources or API URLs change, is it possible to repair the invalid resources and code?
- **Understanding the Python community's use of gists:** Wang et al. [5] studied the use of Public gists on GitHub, observing a variety of uses, but did not focus on usage categories per language or file type. We observed several different usage patterns for Python gists in the Gistable database. Future research can inspect and categorize the types of practices that emerge from creating and sharing public gists in the Python community.

VIII. Limitations

Our analysis may overestimate the number of executable snippets. For example, a gist may define only a single function containing all code, but never call it. So long as the function definition succeeds, the gist is marked as successful regardless of whether or not the code works. Any additional dependency errors caused by executing the function will not be triggered. Further, our analysis may misclassify an exception. For example, it is possible for a gist to implicitly hide import errors by catching them and then raising an error of a different type. Future analysis can use several measures to increase the certainty of successfully execution: annotating gists with test assertions, increasing path coverage of executed gists, manual inspection and verification, and iteratively fixing configuration issues and evaluating gist execution.

Another concern is that running a gist once will only produce the first fatal error encountered, although the gist may have more than one. As a result, we may underestimate the distribution of some category of errors. However, research from Becker et al. [7] argues that the practical approach in such a case is to focus on the first failure encountered, as this mirrors how developers typically resolve errors.

We caution readers to not overgeneralize our results. While we analyzed a large sample of Python gists, these results may not extend to other programming languages. Numerous factors, such as the experience of programmers, the quality of modules and package management, the degree of third-party modules usage, and language design can influence how executable a code snippet is in practice. In other languages and ecosystems, these factors may be less of a concern. Further, we examine public gists, which may differ from private gists.

Our environment inference algorithm can have several limitations. Even if all dependencies install correctly and gist execution succeeds, there is no guarantee the package API will not undergo a breaking change between the time the Dockerfile is created and the time the image is built.

IX. Related Work

The work by Yang et al. [3] is the closest related work in terms of research approach and methodology. Yang et al. [3] examined Python snippets on Stack Overflow and found that 75% were parsable and only 25% were runnable. In this paper, our work differs in several important ways. First, our primary focus is on the ability to execute Python snippets, whereas Yang et al. were primarily focused on the ability to parse snippets. Second, we investigate the effectiveness of a naive inference algorithm in recovering an execution environment for the snippets. Third, we manually construct execution environments and characterize reasons why code may not be executable. Finally, our research context differs in that we examine gists shared on GitHub instead of code snippets found

in Stack Overflow answers. Overall, our research complements Yang et al.'s [3] work in understanding challenges for sharing and using snippets on the web, while providing new directions for research in automated configuration inference.

Several researchers have characterized the buildability of software projects. Sulír and Porubän [20] performed a study on 7,200 Java projects and studied the ability to automatically build them by attempting a maven or ant build in a virtual machine. They found that more than 38% of builds ended in failure. The authors identified that the largest portion of errors are dependency-related. Incidentally, Urli et al.'s study [21] on program repair of Java programs is related to our work. Urli et al. found that by attempting to automatically build 1,609 Java projects on GitHub with Maven, they could only reliably reproduce 31.82% of test failures due to the complexity of mimicking configuration for test environments. A notable difference is that our work focuses on automated configuration inference, whereas Urli et al. focus on repairing Java code in order to pass test failures and thus does not investigate why projects could not build or run tests. *Buildability* and *executability* are related yet distinct concerns in software maintenance. First, build failures can be associated with difficulty inherent in build maintenance that is independent of reproduction. For instance, McIntosh et al. [4] find that the effort involved in maintaining the build configuration can introduce 27% overhead on source code development and a 44% overhead on test development. Such high effort could increase the odds of out-of-date or non-buildable projects. Second, while building large and complex projects can be daunting, this process does not necessarily *run* the code, which can require further environment specifications. Finally, our research context differs from buildability of software projects in that we are interested in automatically executing isolated code snippets without build specifications, which is common in learning or documentation contexts.

German et al. [22] describe multiple problems associated with managing and specifying dependencies, including downloading, building, and satisfying inter-dependent artifacts, which may not always be explicitly documented. They propose a framework for categorizing dependency types and a method for building and visualizing an inter-dependency graph of a package. Lungu and colleagues [23] note that dependencies also exist between projects in a software ecosystem. They propose a model which can capture inter-project dependencies. In our work, we are interested in characterizing both dependencies as well as other environment resources that when absent can prevent code from being executable. We believe our empirical findings complement these models and together, they can be used to inform the design of an automated configuration inference tool.

X. Conclusion

Code snippets can be a useful way to explain and demonstrate a programming concept, but may not always be directly executable. We investigated the executability of Python gists hosted on GitHub and the ability for a naive inference algorithm to recover a Dockerfile capable of executing the Python gist. Finally, we investigated the types of execution failures encountered when running Python gists and the effort involved in manually creating a Dockerfile able to run a gist.

Overall, we find that most gists are not executable in a default Python environment. Further, the exceptions raised when attempting to execute the gists suggests that an insufficiently configured environment is the primary cause.

Our inference algorithm shows that, at least in some cases, correct application environment configurations can be automatically recovered. While a naive approach can infer dependencies for some gists, it fails to do so in the majority case. Additional strategies promise greater success, and will be the subject of future research.

Our investigation of Python gists finds that they often require non-trivial environment configuration in order to run. There are multiple reasons why configuration for any particular gist might be difficult, but the most common challenges are finding dependencies without obvious names and installing dependencies with transitive dependence on system modules.

Finally, we envision multiple applications for Gistable that extend beyond empirical studies of executability. Gistable can automatically configure and execute approximately 5,000 public Python gists hosted on GitHub. Each gist has an accompanying Dockerfile which can be used to build a Docker image based off of the `python:2.7.13` image which contains both the gist and its dependencies. Running the Docker image executes the gist without `ImportError`. Gistable also ships with a simple command line utility for cloning gists in the dataset, and building and running Docker images.

Acknowledgements

This work is funded in part by the NSF grant #1814798.

References

[1] J. Sillito, F. Maurer, S. M. Nasehi, and C. Burns, "What makes a good code example?: A study of programming q&a in stackoverflow," in *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ser. ICSM '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 25–34. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2012.6405249

[2] D. Yang, P. Martins, V. Saini, and C. Lopes, "Stack overflow in github: Any snippets there?" in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 280–290. [Online]. Available: https://doi.org/10.1109/MSR.2017.13

[3] D. Yang, A. Hussain, and C. V. Lopes, "From query to usable code: An analysis of stack overflow code snippets," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 391–402. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901767

[4] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 141–150. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985813

[5] W. Wang, G. Poo-Caamaño, E. Wilde, and D. M. German, "What is the gist?: Understanding the use of public gists on github," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 314–323. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820556

[6] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 92–101. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597074

[7] B. A. Becker, C. Murray, T. Tao, C. Song, R. McCartney, and K. Sanders, "Fix the first, ignore the rest: Dealing with multiple compiler error messages," in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '18. New York, NY, USA: ACM, 2018, pp. 634–639. [Online]. Available: http://doi.acm.org/10.1145/3159450.3159453

[8] J. Saldaña, *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2009.

[9] M. Birks, Y. Chapman, and K. Francis, "Memoing in qualitative research: Probing data and processes," *Journal of Research in Nursing*, vol. 13, no. 1, pp. 68–75, jan 2008.

[10] J. Ponterotto, "Brief note on the origins, evolution, and meaning of the qualitative research concept thick description," *The Qualitative Report*, vol. 11, no. 3, 2006.

[11] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding in-depth semistructured interviews," *Sociological Methods & Research*, vol. 42, no. 3, pp. 294–320, aug 2013.

[12] C. Parnin, C. Treude, and M. A. Storey, "Blogging developer knowledge: Motivations, challenges, and future directions," in *2013 21st International Conference on Program Comprehension (ICPC)*, May 2013, pp. 211–214.

[13] C. Treude and M. Aniche, "Where does google find api documentation?" in *IEEE/ACM 2nd International Workshop on API Usage and Evolution*, ser. WAPI'18. New York, NY, USA: ACM, 2018.

[14] A. Lucia, M. Penta, R. Oliveto, A. Panichella, and S. Panichella, "Labeling source code with information retrieval methods: An empirical study," *Empirical Softw. Engg.*, vol. 19, no. 5, pp. 1383–1420, Oct. 2014. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9285-5

[15] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, ser. WCRE '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 35–44. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2010.13

[16] P. W. McBurney, C. Liu, C. McMillan, and T. Weninger, "Improving topic model source code summarization," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 291–294. [Online]. Available: http://doi.acm.org/10.1145/2597008.2597793

[17] B. P. Eddy, J. A. Robinson, N. A. Kraft, and J. C. Carver, "Evaluating source code summarization techniques: Replication and expansion," in *2013 21st International Conference on Program Comprehension (ICPC)*, May 2013, pp. 13–22.

[18] A. Weiss, A. Guha, and Y. Brun, "Tortoise: Interactive system configuration repair," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 625–636. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155641

[19] G. Schermann, S. Zumberi, and J. Cito, "Structured Information on State and Evolution of Dockerfiles on GitHub," Mar. 2018, Detailed information on the dataset can be found in the paper "Structured Information on State and Evolution of Dockerfiles on GitHub" accepted at the Data Showcase Track of the International Conference on Mining Software Repositories 2018 (MSR 2018). The software used to collect the dataset and instructions on how to use the dataset can be found in the paper's online appendix: https://github.com/sealuzh/msr18-docker-dataset. [Online]. Available: https://doi.org/10.5281/zenodo.1200869

[20] M. Sulír and J. Porubän, "A quantitative study of java software buildability," in *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU 2016. New York, NY, USA: ACM, 2016, pp. 17–25. [Online]. Available: http://doi.acm.org/10.1145/3001878.3001882

[21] L. S. M. M. Simon Urli, Zhongxing Yu, "How to design a program repair bot? insights from the repairnator project," in *40th International Conference on Software Engineering, Track Software Engineering in Practice (SEIP)*, ser. ICSE 2018, 2018, pp. 1–10. [Online]. Available: https://hal.inria.fr/hal-01691496/document

[22] D. M. German, J. M. Gonzalez-Barahona, and G. Robles, "A model to understand the building and running inter-dependencies of software," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, Oct 2007, pp. 140–149.

[23] M. Lungu, R. Robbes, and M. Lanza, "Recovering inter-project dependencies in software ecosystems," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 309–312. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859058