

PROJECT REPORT FOR “AUTONOMOUS SYSTEMS 2020/2021”

2PPy

Updating the 2pKt library porting for Python

November 28, 2023

Luca Deluigi - `luca.deluigi3@studio.unibo.it`

Contents

1	Introduction	3
2	Requirements	4
2.1	Functional requirements	4
2.2	Non functional requirements	4
2.3	MoSCoW analysis	4
2.4	Deliverables	5
3	Analysis	6
3.1	2p-Kt runtime dependency	6
3.2	“Pythonic” API for 2PPy	6
3.3	Development process improvement	7
3.4	2PPy Documentation	7
4	Design & Implementation	8
4.1	Porting a Kotlin open source library to Python	8
4.1.1	JPytype	8
4.1.2	Resolving library dependencies during build	9
4.1.3	Providing Javadoc access through Python	9
4.1.4	Resolving JVM dependencies during build	9
4.1.5	Multiplatform Testing	10
4.1.6	Pythonic API generator	11
4.1.7	Porting patterns	12
4.2	Porting the 2p-Kt library to Python	15
4.2.1	Mapping <code>tuprolog</code> types, methods and constants	15
4.2.2	Testing the <code>tuprolog</code> API	15
4.2.3	Git based Semantic Versioning	16
4.2.4	REPL interpreters	16
5	Conclusions	17
6	Future steps	18
	References	19

Abstract

2P-Kt[1] is a multi-platform logic programming framework written in Kotlin, mainly targeting the Java Virtual Machine. The framework supports symbolic manipulation and reasoning in an extensible and flexible way.

2PPy is an experimental Python library that aims to port a subset of the 2P-Kt ecosystem to Python. The porting is enabled by JPype¹, a Python package that provides full access to Java from within Python.

This project consists of the design and implementation of the next iteration of the 2PPy project lifecycle. Indeed, the project currently lacks proper documentation and unit tests, as well as a proper CI/CD pipeline, the java runtime bundled in the deliverable and contributor friendliness. Each of these aspects will be addressed in this document.

¹<https://jpype.readthedocs.io/en/latest/>

1 Introduction

In the vast domain of *Artificial Intelligence* (AI), the *symbolic* approach to knowledge representation often stands in contrast to the *sub-symbolic* approach. Symbolic AI relies on a symbolic representation of knowledge, offering the advantage of human-readability and automatic explainability. On the other hand, sub-symbolic AI employs representations such as numbers in neural networks, providing a means to handle uncertainty but often operating as a black box, lacking human interpretability.

Over the last few decades, both approaches have witnessed alternating periods of progress and setbacks, concluding with the rise of Deep Learning, a sub-symbolic family of techniques, that marked a significant success for sub-symbolic AI research. Meanwhile, recent symbolic methods like Inductive Logic Programming have been addressing uncertainty in knowledge and environmental perception of intelligent agents. In order to benefit from all recent developments and successful cases of both the approaches, researches have increasingly explored the integration of symbolic and sub-symbolic AI.

Within the realm of logic programming, 2PPy aims to serve as technological bridge between symbolic manipulation and reasoning (as provided by 2P-Kt over the JVM) and the Python environment, which is the main platform adopted by sub-symbolic AI practitioners. The primary objective of 2PPy is to enable and facilitate seamless integration between the symbolic capabilities of 2P-Kt and Python world, potentially enabling the combination of logic programming with machine learning libraries such as PyTorch² or TensorFlow³. Through the use of 2PPy, developers can program intelligent agents, leveraging logic programming enabled by 2P-Kt while also harnessing the capabilities of machine learning models to address specific problems tied to the agent’s plans and intentions.

Unfortunately, 2PPy isn’t ready for real world applications, being mostly a work-in-progress and lacking an updated, “Pythonic” API for 2P-Kt and a proper management of the software dependencies and life-cycle. This project aims to address these issues and bring 2PPy to be a valid, stable candidate for public release and adoption as a library for real world applications and research.

²<https://pytorch.org/>

³<https://www.tensorflow.org/>

2 Requirements

This section outlines the requirements for this project, including goals and tasks that may not be achieved in the final version due to constraints in resources or time.

2.1 Functional requirements

This section enumerates the features considered in this project.

- **F.1** Update the Java API to the latest 2P-Kt version;
- **F.2** Make the Python API as much “Pythonic” as possible;
- **F.3** Distribute the Java runtime environment together with the package and 2P-Kt Java Archives (JARs);
- **F.4** Write a mapping for a custom predicates API;
- **F.5** Enable the selection and parameterization of the 2p-Kt version used by the project.

2.2 Non functional requirements

This section describes additional requirements considered in this project.

- **NF.1** Write unit tests of the API, maximizing the coverage;
- **NF.2** Improve the Continuous Integration/Delivery (CI/CD) workflow, for example by fixing the 2p-Kt version with a custom dependency file;
- **NF.3** Adopt the Git flow previously used in the project (working on `develop`, merging on `master` for releases) and document it;
- **NF.4** Write code documentation and generate a “read-the-docs” website;
- **NF.5** Write development documentation, including this report and contributors guidelines.

2.3 MoSCoW analysis

Each requirement is categorized into:

- **Must:** the requirement is mandatory;
- **Should:** the requirement should be implemented to achieve completeness;
- **Could:** the requirement is optional and should be implemented only if resources allow to;
- **Won't:** the requirement won't be implemented in this project;

See the table 1 for the result.

Requirement	MoSCoW	Priority
F.1	must	9
F.2	must	8
F.3	should	6
F.4	won't	1
F.5	should	7
NF.1	must	10
NF.2	must	7
NF.3	must	7
NF.4	should	5
NF.5	could	3

Table 1: MoSCoW table of the requirements enumerated above.

2.4 Deliverables

The practical deliverables of the project are one or more Pull Requests (PRs) submitted to the library's official GitHub repository. If PRs get reviewed and approved, they will eventually be merged into the main development line and become the next release of the Python module, available from the Python Package Index (PyPI).

3 Analysis

Requirements are grouped based on their affinity. Each group is examined independently from the others, although some interdependencies do exist.

3.1 2p-Kt runtime dependency

This requirement group consists of: **F.1**, **F.3**, **F.5** and part of **NF.2**.

Overall, these requirements aim to overhaul the way 2PPy declares and resolves its dependencies on the 2p-Kt Java deliverables, as well as practically update the dependencies version and consequently update the Python API wrapper.

Currently, the 2p-Kt version and dependency is resolved using a custom, hand-made *shell script* that uses the `curl` command to fetch the latest version from the official *2p-Kt repository*⁴. The official 2p-Kt repository adheres to semantic versioning⁵ for the releases. This way, the shell script can parse the latest released version and can compose direct URLs for downloading the needed *jar files* from the *GitHub release page*⁶.

The objective is to improve the reliability of the dependency management by pinning the supported version of 2p-Kt and by standardizing the dependency resolution process. This also requires to update the python API wrapper code to reflect changes in the latest 2p-Kt version. Ideally the system would also receive automatic notifications or pull requests which suggest upgrading the dependencies to the newest available versions of 2p-Kt on new releases.

3.2 “Pythonic” API for 2PPy

This requirement group consists of: **F.2**, **F.4** and **NF.1**.

The primary goal here is to improve the usability of the 2p-Kt environment for the average Python developer.

Porting Kotlin style and design paradigms into Python is not straightforward. First of all, Kotlin code gets compiled into Java bytecode, which is then encapsulated and delivered in JAR files. Finally, these JARs are imported by the Java Virtual Machine (JVM) that executes the bytecode. Integrating this pipeline with Python needs allowing a Python interpreter to manage a JVM instance at runtime, enabling to programmatically alter classes, create and destroy objects, call methods on them and so on. In practice, this is currently achieved using a Python library called JPyPe⁷, which satisfies most requirements but has some limitations, as discussed in section 4. Nevertheless, using JPyPe is not quite enough to have a *good* porting. In fact, JPyPe maps methods, objects, classes and interfaces one-to-one to their Python counterparts. This means that the API inherits the patterns, style and conventions of Kotlin programming, further altered by the Kotlin compiler.

⁴<https://github.com/tuProlog/2p-kt>

⁵<https://semver.org>

⁶<https://github.com/tuProlog/2p-kt/releases>

⁷<https://www.jpype.org>

At the current state, evident frictions are present between traditional “Pythonic” programming style and the hybrid approach adopted by 2PPy. The desired experience would be a seamless integration of 2p-Kt API into Pythonic styled programs, meaning that the underlying Java oriented code base is well hidden from a Python user perspective. Moreover, in order to offer a certain degree of reliability and correctness, the new API needs to be put under a series of meaningful tests.

3.3 Development process improvement

This requirement group consists of: **NF.2**, **NF.3** and **NF.5**.

Currently, 2PPy does not meet the standards of open-source projects. A lack of comprehensive documentation hinders third-party contributions, not only due to the absence of complete documentation but also a lack of clarity on the development guidelines. This mandates to write a clear contribution guidelines document that includes the Git conventions such as Git branching strategy and Semantic Versioning tagging information.

In addition, the project lacks a proper and exhaustive CI/CD pipeline. The current setup only covers Continuous Delivery with a basic GitHub Actions workflow, leaving room for potentially flawed releases, and unverified, potentially harmful PRs. Ideally, the new CI/CD would test the build of the package on multiple architectures and configurations, and assess its functionality once installed on different operating systems.

3.4 2PPy Documentation

This requirement group consists of just **NF.4**.

Documenting open source projects is crucial to assist both current and future users as well as contributors. Currently, 2PPy lacks standalone documentation, and relies on redirecting to the API of 2p-Kt for user guidance.

It would be desirable to have comprehensive user documentation of the API methods and classes that follows the standards of Python projects. Some kind of automated import of the documentation from 2p-Kt would be optimal, although manually written documentation would be fine too. The documentation should be accessible through Python docs, and ideally a “read-the-docs” style website would be generated and published.

4 Design & Implementation

This section describes the methodologies employed to address the problems delineated in section 3. It is divided in two primary subsections:

- Section 4.1 concerns generic *Kotlin/Java* $\rightarrow$ *Python* porting solutions, that are applicable to any Python porting of JVM based libraries;
- Section 4.2 concerns the specific *2p-Kt* $\rightarrow$ *2ppy* porting challenges.

4.1 Porting a Kotlin open source library to Python

The general idea for porting JVM-based code (whether written in Kotlin, Scala or Java) to Python is to maximize the reuse of pre-existing code while crafting a Pythonic API. This idea translates into exposing a new API in Python which wraps the Java code base with Pythonic styled classes, methods and constants. The term “Pythonic” refers to the adoption of standard Python naming conventions and programming patterns, leveraging Python advanced language features like default method arguments, keyword method arguments and generators, with an overall emphasis on simplicity and intuitiveness.

In this context, 2p-Kt serves as a representative example for any JVM based library, regardless of its original programming language, provided it targets the JVM.

4.1.1 JPytype

JPytype⁸ is a Python package, available on PyPi, written with Python, C and Java, functioning as a cross-language bridge between Python and Java. It enables bidirectional communication between the Python interpreter and the JVM. To operate, JPytype requires access to a JVM installation and the JAR archives of the Java libraries to be loaded into the initiated JVM. Although JPytype allows Python to import data types defined in Java using a Python-esque import syntax, Java packages, classes and members in general are named differently than Python corresponding entities. Usually, Java packages are nested inside a reverse hostname-like path, while Python packages are not. Furthermore, Python libraries often exploit features like properties, operator overloading, default method parameters, keyword variable arguments, snake casing for symbol names and so on which Java neither employs nor support. Therefore, 2PPy acts as an API wrapper that conceals the underlying Java implementation, enhancing the experience for the average Python developer.

Given JPytype’s capabilities relative to the alternatives⁹, it remains the best way to bridge 2PPy with 2p-Kt. The code facilitating type mapping between Python and Java/Kotlin standard libraries has been retained from the previous version of 2PPy. This code also circumvents some of JPytype’s limitations, emerging especially when working with libraries originally written in Kotlin. For instance, that code helps avoiding frequent errors due to ambiguous method overload calls,

⁸<http://www.jpype.org>

⁹<https://jpype.readthedocs.io/en/latest/userguide.html#alternatives>

caused by the fact that Python does not support method overloading. Such issues are mitigated through careful method overriding in Python; however, overriding Java code from Python is not possible with JPyype, meaning that sometimes classes need to be re-implemented from scratch in pure Python.

4.1.2 Resolving library dependencies during build

Ideally, 2p-Kt JARs should be encapsulated within the Python package itself, during its build phase. Additionally, specifying the 2p-Kt version in a standard manner would enable the configuration of automated processes to detect new available versions of 2p-Kt.

Among the many Java build systems, **Maven**¹⁰ stands out for the user-friendliness of its command line interface and its capability to download and resolve transitive dependencies, including optional javadoc, by downloading JARs from the *Maven Central Repository*¹¹. For this reasons, Maven is selected as the build system in charge of resolving JVM dependencies, and Java dependencies are specified inside the standard `pom.xml` file and resolved during the build process by invoking Maven. The Maven invocation is executed in the canonical `build.py` script file, serving as support for the declarative Python package configuration file, `pyproject.toml`.

During the build process, `setup.py` runs the `mvn` command, ensuring all necessary JAR files are bundled withing the package `libs` directory. Java bytecode cross-platform nature guarantees that the JAR files are compatible with any device where 2PPy is installed.

Furthermore, using a standard `pom.xml` file allows to configure automatic dependency management tools such as **Dependabot**¹². Dependabot autonomously opens pull requests containing updates to dependency files such as `pom.xml`, `pyproject.toml` or `requirements.txt`, whenever new versions of dependencies are released.

4.1.3 Providing Javadoc access through Python

JPyype allows accessing the Javadoc of interfaces, classes and methods from Python through the `help(Name)` method, provided that the Javadoc of `Name` is present in the classpath. Thanks to the user-friendly Maven command line interface, it takes just a one line change in `setup.py` to download and include the javadoc JARs of 2p-Kt alongside the bytecode of the package. However, due to a known issue¹³ of JPyype, accessing the Javadoc currently raises errors. As a result, the javadoc download line in `setup.py` has been commented out. Once the JPyype issue is resolved, 2PPy will be able to consult the 2p-Kt javadoc from Python by simply invoking `help(object_or_class)` and passing the name of any class or object to the function.

4.1.4 Resolving JVM dependencies during build

Ideally, the runtime need of a valid Java Runtime Environment (JRE) should not be a concern for a Python developer. Therefore, it is preferable to download a JRE during or shortly after

¹⁰<https://maven.apache.org>

¹¹<https://mvnrepository.com>

¹²<https://docs.github.com/en/code-security/dependabot>

¹³<https://github.com/jpyype-project/jpyype/issues/1149>



Figure 1: CI/CD workflow diagram of a commit as displayed by GitHub. Release and publish steps are skipped unless a new release tag is submitted.

installation of 2PPy, without the user noticing, besides the fact that the installation needs a working internet connection.

This requirement is met by depending on the **install-jdk**¹⁴ package, which allows any Python script to download the right JRE for the host machine at runtime. During most of the development of 2PPy, **install-jdk** was the only dependency solver for the JRE. However, in the latter stages of development, **install-jdk** was replaced with **jdk4py**¹⁵. Unlike **install-jdk**, **jdk4py** downloads the right JRE during the installation process and not after, thanks to the fact that **jdk4py** is a Python package with multiple distributions specific for every major operating systems. Each of these distributions comes with a pre-packaged JRE, meaning it does not download anything at runtime. Consequently, 2PPy can run without the need of an internet connection from its first startup after the installation.

4.1.5 Multiplatform Testing

2PPy is designed to be portable, both for its development and execution. Python and Java bytecode languages are portable, and thanks to the custom JRE packaged with 2PPy, the entire library should function on any device meeting the requirements. These mainly concern the Python supported versions, with a lower bound set by the Python code to be ≥ 3.9 and an upper bound set by JPytype, which supports Python versions up to ≤ 3.11 . 2PPy could be delivered with any JRE version compatible with the compiled bytecode, upper bounded by the latest JRE version supported by **jdk4py**, currently 17.0.7.

To ensure code portability for both the build process and execution of 2PPy, a multiplatform automated testing facility has been established using **GitHub Actions**, which is free for open source projects. Using GitHub Actions, the CI/CD workflow (see Figure 1) creates a *test matrix* with all the possible combinations of supported operating systems (*Mac OS*, *Windows* and *Ubuntu Linux*) and supported Python versions (3.9, 3.10, 3.11). For every combination, it tests the build process and executes unit tests on each of the nine combinations.

¹⁴<https://pypi.org/project/install-jdk>

¹⁵<https://pypi.org/project/jdk4py>

Following a successful the build and test phase, two artifacts are generated: a *tarball* for **sdist**¹⁶ and a *wheel binary* for **wheels**¹⁷. Both distribution formats undergo testing on all the nine combinations of operating systems and Python versions, resulting in eighteen configurations tests. Each configuration executes a post-install test script to ensure that everything works as expected (see Figure 1). Additionally, the last two steps of the workflow handle the release of new versions on GitHub and PyPi when a new release git tag is pushed.

Besides the main continuous integration workflow, a separate `lint` job enforces common Python-style programming rules, maintaining the code base style consistent and improving its readability. Maintaining a consistent coding style with interpreted languages such as Python is important because the source code of dependencies sometimes serves as a substitute for the documentation.

4.1.6 Pythonic API generator

Python developers tend to adopt snake casing¹⁸ as the method naming convention, while Kotlin, Scala and Java use camel casing¹⁹. Additionally, Scala, Kotlin and Python support default parameter values for methods, while Java and Java bytecode do not. This results in the generation, from the Kotlin and Scala compilers, of numerous method overloads, at least one for every default argument. The issue arises from the fact that Python does not support method overloading. As a result, JPytype had to implement an overload selection algorithm that dynamically selects the correct overload to invoke at runtime based on the arguments provided when a method is called from Python. In other words, every method accessible from Python but originally implemented in Java is an instance of the `JMethod` type, defined by JPytype. This class handles overload selection, among other tasks.

Furthermore, Kotlin and Scala support public properties, whereas in Java bytecode they are compiled into pairs of accessor and mutator methods, commonly known as getter and setter methods (if the property is read-only, only the accessor is present). The problem here is that Python supports properties, but they are not used by JPytype default method import systems.

The differences between Python and Java result in non-Pythonic imported method sets, sometimes even causing runtime errors such as overload ambiguity. This brings the necessity to write many custom Pythonic overloads with snake-cased names, default arguments, and sometimes property accessor and mutator methods. Fortunately, JPytype allows to enhance already existing Java types (classes and interfaces) with additional Python logic, including new methods and properties. Nevertheless, manually crafting properties for each getter-setter pair and converting each method into its Pythonic overload can be a cumbersome task. This burden can be mitigated through the implementation of a method and property generator, based on Java and Python conventions, that provides an additional Pythonic mapping for methods from Java to Python. This would represent a great improvement over the default one-to-one method mapping

¹⁶*sdist* stands for source distribution and is the legacy distribution format for Python packages

¹⁷*wheels* are the newer distribution format for Python packages, which is build system independent

¹⁸https://en.wikipedia.org/wiki/Snake_case

¹⁹https://en.wikipedia.org/wiki/Camel_case

done by JPy.

Drawing inspiration from the JPy *bean* module, a Pythonic method and property generator has been implemented and incorporated to every subclass of `java.lang.Object`, which affects every object imported from the JVM. This generator simply creates “smart” Pythonic aliases of existing methods, although it still delegates overload selection to JPy’s `JMethod` objects. This generator is limited to generating member aliases, meaning that it cannot produce smart default parameters for methods using overloads, as it only has visibility into Python members provided by JPy. The implementation is in pure Python, but it leverages certain JPy internal features implemented with C, such as annotations on the original method signatures, and utilizes JPy features for adding methods to loaded classes²⁰ (see Figure 2).

4.1.7 Porting patterns

Programming with statically typed object-oriented languages like Java or Kotlin often leads to the application of various object-oriented design patterns. Some of these programming patterns rely on language features that are absent in Python, such as abstract classes and methods, interface segregation, private or protected visibility for methods and constructors, and more. Others depend on features not supported by JPy by design, like class inheritance from Java to Python. Consequently, certain well-known patterns from the book *Design Patterns: Elements of Reusable Object-Oriented Software*[2] had to be entirely reconsidered and adapted to be suit the Python environment, JPy and *Pythonic* style. The following paragraphs provide detailed insights on the patterns that required specific transformation strategies.

Builder The builder pattern naturally fits in statically typed languages because the compiler can validate the correctness of the fluent method call chain used to set the parameters of a built entity. In Pythonic style, however, it is more common to rely on the default keyword arguments of a builder function to set the parameters of the entity being built. This approach reduces the need for multiple class definitions, object instantiations and fluent call chains that may not fit well the Pythonic style. Fluent call chains spanning multiple lines are particularly cumbersome in Python, because starting a new line of code with a fluent call, when outside of a bracket pair, requires a continuation character at the end of the previous line (see the examples below).

²⁰<https://jpy.readthedocs.io/en/latest/api.html#jpy.JImplementationFor>

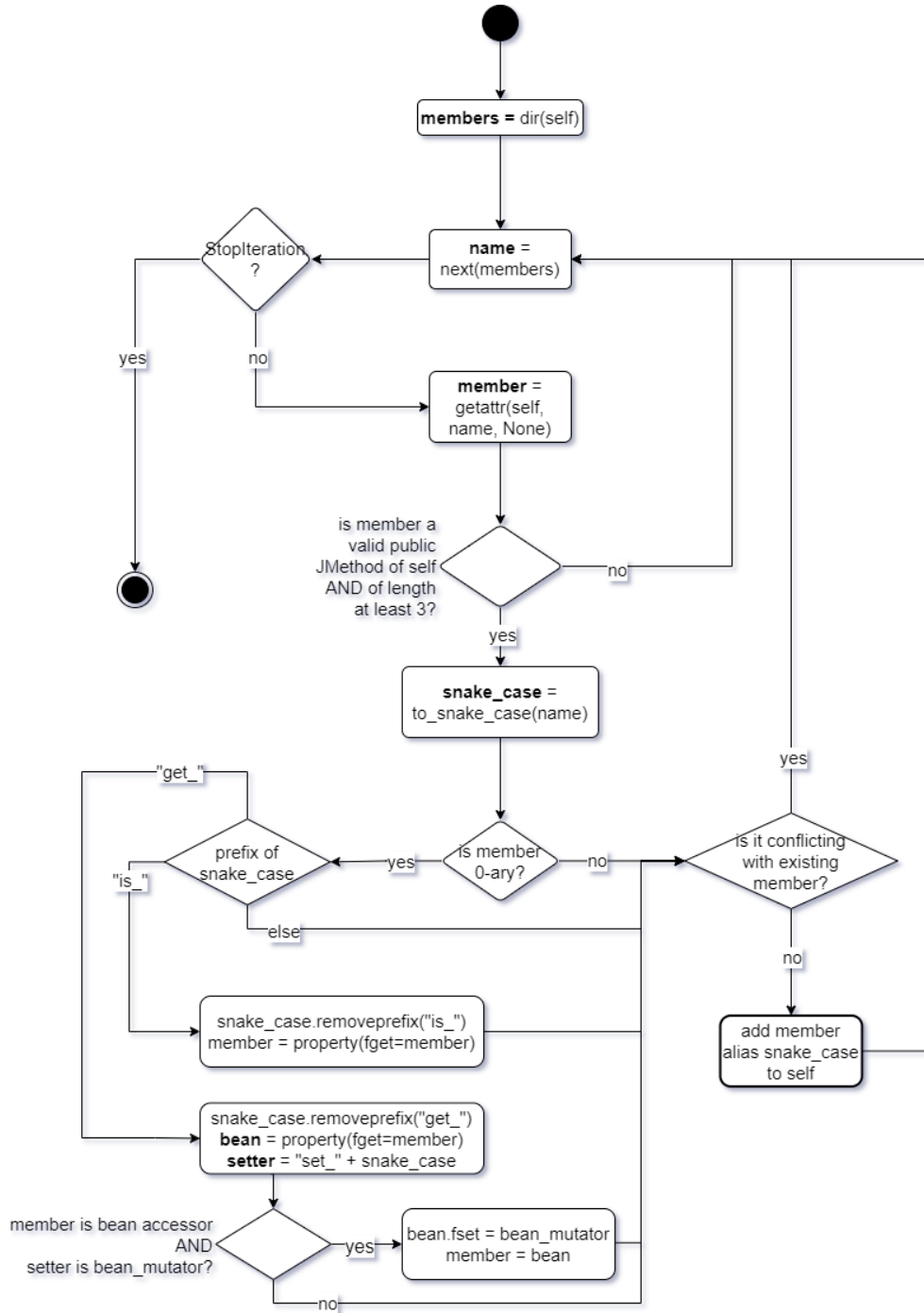


Figure 2: UML activity diagram for the Pythonic method generator. In this context, self refers to the object running the algorithm during its initialization phase. Adding member as alias is possible thanks to JPyype `customize` method.

```
# Using a fluent builder in Python
solver_builder = PROLOG.SOLVERFACTORY.new_builder() \
    .static_kb(theory) \
    .flag(TrackVariables, TrackVariables.ON) \
    .standard_output(output_channel(self.print_output))
```

```
# Replacing the builder with default parameters
solver = prolog_solver(
    libraries=self.thermostat_agent.library,
    flags=DEFAULT_FLAG_STORE.set(TrackVariables, TrackVariables.ON),
    static_kb=theory,
    std_out=output_channel(self.print_output)
)
```

Singleton In Java, singletons usually consist of an instance of a public entity with a private constructor, made accessible through a public static read-only field within entity class. In Kotlin, singletons are implemented with public constants in companion objects, which are then compiled to Java bytecode using a technique similar to how Java programmers implement them, with a static read-only field called `INSTANCE` created by the compiler. In Python, due to the absence of visibility modifiers, accessing the main instance of a type from its class is uncommon. Instead, a single instance of a type is often accessed through a constant imported using the Python module containing the type (see example below).

```
# Porting a singleton from Kotlin in Python
CommonBuiltins: _stdlib.CommonBuiltins = _stdlib.CommonBuiltins.INSTANCE
```

Template method In Java and Kotlin, template methods are enabled by the presence of abstract classes. Inheriting from these abstract classes and implementing the template method(s) can be convenient, especially if the language allows the creation of anonymous classes that implement the template method(s) on the fly. With JPyype, inheriting from an abstract class defined in Java bytecode is not allowed. This results in complete incompatibility of template methods with Python when using JPyype. Consequently, type trees based on abstract classes must be rewritten from the root, encapsulating the same logic as their Java counterparts (see example below).

```
# Re-implementing in Python an abstract class that implemented
# the LogicFunction interface with a template method
@JImplements(LogicFunction)
class AbstractLogicFunction(object):
    @JOverride
    def compute(self, request: ComputeRequest) -> ComputeResponse:
        raise NotImplementedError()
```

Iterable Iterable adapters from Java to Python and vice versa are partially provided by JPytype out-of-the-box and further enriched by a custom JPytype *JConversion* with an adapter class. Python programs frequently make extensive use of iterables, especially in the form of lazy generator functions, thanks to the `yield` keyword. Thanks to JPytype, generator functions written in Python can effectively replace lazy iterables required by Java methods (see example below).

```
def get_temp(self, request: SolveRequest) -> Iterable[SolveResponse]:
    ensuring_argument_is_variable(request, 0)
    sub = mgu(request.arguments[0], integer(self.temperature))
    yield request.reply_with(sub)
```

4.2 Porting the 2p-Kt library to Python

This section concerns the porting stories lived during the development of 2PPy, specifically tailored for 2p-Kt.

4.2.1 Mapping tuprolog types, methods and constants

Every 2p-Kt kotlin package has been ported into one corresponding Python module, nested approximately in the same way, with the exception of the root. In 2p-Kt, the root package is `it.unibo.tuprolog`, whereas in 2PPy the root module is simply `tuprolog`.

For each package, the corresponding Python module includes *definitions*, the functions API and possibly JPytype *adapters*. Definitions consist of the following:

- 2p-Kt or Kotlin classes imported as-is using JPytype,
- redefinitions of abstract classes present in 2p-Kt that can't otherwise be used by simple importing with JPytype,
- imported singletons and constants.

The functions API contains function definitions used to create or manipulate instances of types defined in definitions. If adapters are present, they define custom JPytype mappings of Java types to Python classes, often adding functionality and disambiguating method overloads.

A similar import process has been undertaken for Java and Kotlin language library types, such as `Object`, `Map`, `Pair`, `Iterator`, `Sequence`, and others, inside the `tuprolog.jvmutils` module.

4.2.2 Testing the tuprolog API

To ensure the usability of the API and to detect potential breaking changes, automated tests have been established. The test framework used to define these tests is `unittest`, available in the Python standard library, and `pytest`²¹ is used to run them.

²¹<https://docs.pytest.org/en/>

One important Key Performance Indicator (KPI) for API testing is code **coverage**, as mentioned in the requirements. Coverage is measured from unit tests using the **coverage**²² tool, which, by the end of development, reaches a total of 82%. This percentage can be increased by adding additional tests (see section 6).

4.2.3 Git based Semantic Versioning

A new automated release workflow has been implemented in order to enhance the integration with the new CI. New 2PPy versions are released and delivered to PyPI²³ using the Continuous Delivery GitHub Actions workflow, triggered by pushing new annotated git tags representing new 2PPy versions. These Git annotated tags are expected to adhere to progressive *Semantic Versioning*²⁴ rules. During the development of 2PPy, test releases have been delivered to TestPyPI²⁵.

4.2.4 REPL interpreters

2p-Kt includes a package called **repl** that implements an interactive Read Evaluate Print Loop (REPL) environment for experimenting with Prolog and Prolog extensions. Initially, 2PPy lacked support for a Prolog REPL but provided a Python REPL interpreter with pre-imported **tuprolog** modules for rapid prototyping and experimentation. The Python library module **cmd** enables the creation of custom REPLs with arbitrary command sets, which has allowed to extend the **tuprolog** suite of REPL interpreters. Despite being initially beyond the project's scope, the decision to include this feature felt justified considering the minimal time investment. As a result, 2PPy now offers the following interactive REPL environments:

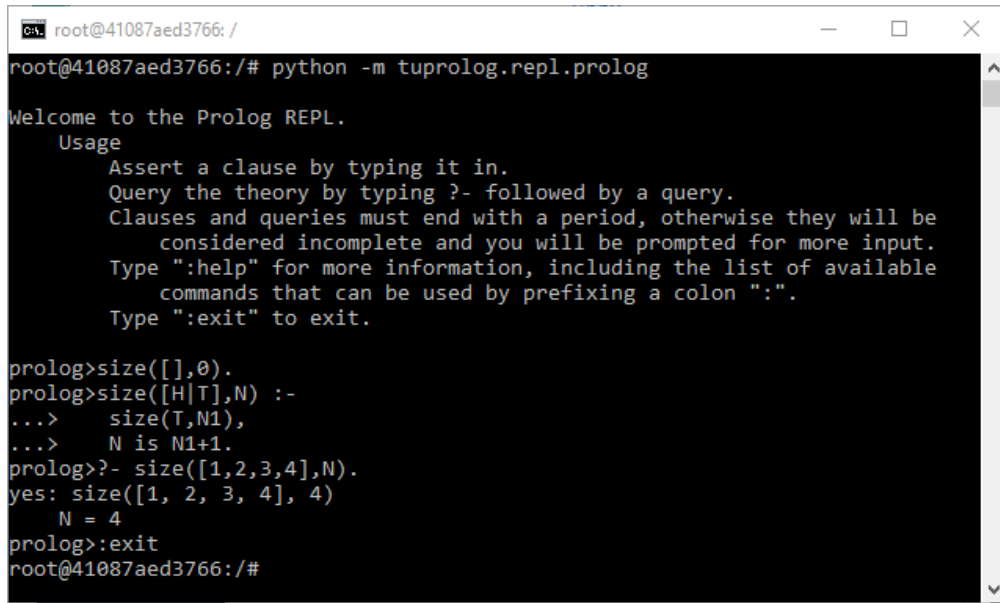
- the already existing Python interpreter with pre-imported **tuprolog** types, functions and constants, which can be invoked with the command `python -m tuprolog`;
- a new **Prolog interpreter** that supports multi-line input and help commands. It works by extending a theory with every new clause submission and optionally running a query against the theory using the “?-” prefix. This interpreter can be invoked with the command `python -m tuprolog.repl.prolog` (see Figure 3);
- a new **ProbLog interpreter** that functions similarly to the Prolog interpreter but supports ProbLog theory syntax.

²²<https://coverage.readthedocs.io/en/>

²³<https://pypi.org>

²⁴<https://semver.org>

²⁵<https://test.pypi.org/project/2ppy>



```
root@41087aed3766: /
root@41087aed3766:/# python -m tuprolog.repl.prolog

Welcome to the Prolog REPL.
Usage
  Assert a clause by typing it in.
  Query the theory by typing ?- followed by a query.
  Clauses and queries must end with a period, otherwise they will be
  considered incomplete and you will be prompted for more input.
  Type ":help" for more information, including the list of available
  commands that can be used by prefixing a colon ":".
  Type ":exit" to exit.

prolog>size([],0).
prolog>size([H|T],N) :-
...>   size(T,N1),
...>   N is N1+1.
prolog>?- size([1,2,3,4],N).
yes: size([1, 2, 3, 4], 4)
      N = 4
prolog>:exit
root@41087aed3766:/#
```

Figure 3: Screenshot of a terminal interacting with the Prolog REPL on linux (ubuntu).

5 Conclusions

In conclusion, 2PPy has reached a significant milestone in its development cycle. It now boasts enhanced development support and dependency management, a cleaner and more comprehensive API, automated tests with coverage measurement, and new features, including the REPL interpreters and a bundled JRE.

Despite the development of some non-mandatory features, certain aspects had to be left behind to adhere to time and scope constraints. The most prominent of these can be found in section 6, which lists possible starting points for future contributions to 2PPy.

The deliverable of the project consists of a pull request, reachable at the address <https://github.com/tuProlog/2ppy/pull/20>.

The developer is satisfied with the project's outcome, even though it took longer than initially expected, primarily due to personal commitments.

6 Future steps

This section enumerates a series of ideas, missing features and tasks that have yet to be completed for improving and extending 2PPy:

- **Type Hints** Currently, importing type definitions from JPytype, whether they are written in Kotlin or Java, results in the total failure of the Python type hint system to correctly interpret type hints. This issue can be traced back to the lack, from JPytype, for a proper type hint generation of imported classes, a known issue of the JPytype project²⁶.
- **Documentation for imported types** At the current state of 2PPy, working with defined types almost always requires prior knowledge of the 2p-Kt framework. It would be helpful to provide the Python help system with the same documentation that imported methods and classes have in Java or Kotlin, namely the Javadoc. To enable this, given JPytype support for Javadoc importing, it should have been sufficient to add javadoc JAR archives to the classpath alongside with 2p-Kt JVM JARs, but it's currently not working due to a bug of JPytype²⁷. Once both the aforementioned JPytype issues get addressed and resolved, it should be possible to enable autocompletion both during development and during experimentation with the Python REPL.
- **More unit tests** Unit tests are few but quite meaningful for their purpose. With just a handful of current test classes the coverage is already as high as 82%, as noted in sub-subsection 4.2.2, so not that much work should be needed to reach 100%. Nevertheless, coverage is only meaningful for the Python API, but doesn't cover all the method generated by the Pythonic API generator, which sometimes lead to overload ambiguity errors with certain parameters. For this reason and as a general quality improvement, more tests should be written, at least until most of the public imported and generated API is used and its functioning asserted.
- **Python 3.12** Upgrading Python support for the recently released version 3.12 requires a new update from JPytype. More specifically, some issues must be addressed in order to have JPytype work with Python 3.12²⁸.
- **Hiding internal code from the public API** At the current state of the project, almost every `__init__.py` file imports the definitions from other files using the `*` (star) operator, meaning that everything is exposed as public API for usage. The reason is that it would take too much time for the scope of this project to replace every star import with the `from` keyword, listing all the definitions to be exposed. Here the issue is that not everything defined in other files is meant to be exposed as API of the package, and `*` should always be avoided inside `__init__.py` files in favor of explicitly selecting and listing what definitions should be visible from the point of view of a consumer.

²⁶<https://github.com/jpytype-project/jpytype/issues/714>

²⁷<https://github.com/jpytype-project/jpytype/issues/1149>

²⁸<https://github.com/jpytype-project/jpytype/issues/1147>

References

- [1] Giovanni Ciatto, Roberta Calegari, and Andrea Omicini. “2P-Kt: A logic-based ecosystem for symbolic AI”. In: *SoftwareX* 16 (2021), p. 100817. ISSN: 2352-7110. DOI: <https://doi.org/10.1016/j.softx.2021.100817>. URL: <https://www.sciencedirect.com/science/article/pii/S2352711021001126>.
- [2] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994. ISBN: 9780321700698. URL: <https://books.google.it/books?id=6oHuKQe3TjQC>.