Vython: a Language with Dynamic Version Checking for Gradual Updating

Satsuki Kasuya
satsuki.kasuya@prg.is.titech.ac.jp
Institute of Science Tokyo*
Tokyo, Japan

Abstract

Updating the version of upstream packages can break software behavior due to incompatibilities. To cope with this problem, inspired by the concept of programming with versions, we propose Vython, a Python subset with dynamic version checking. Vython enables programmers to safely and gradually update by allowing the simultaneous use of multiple versions and reporting incompatible versions that are used together within the same data flow. We discuss the design and naive implementation of Vython, evaluate its runtime performance, and explore future directions to facilitate smoother updates in practical development.

Keywords: Software maintenance, Software migration, Dependency management, Python

1 Introduction

Updating the version of upstream packages is one of the most troublesome tasks for downstream developers [8, 11]. An incompatible new version can break the behavior of downstream programs [5, 7]. Each new release of upstream packages requires downstream developers to assess its impact and modify their source code accordingly.

Replacing an upstream package with its new version is automated by package managers such as pip [1] in Python. For example, developers using NumPy [6] can automatically install the latest version by running `pip install -U numpy`. Many developers benefit from this automation, as packages like NumPy are widely used across various domains, such as data analysis, deep learning, and image processing, in libraries like Pandas [17], PyTorch [12], and OpenCV [2].

Downstream developers carefully coordinate existing programs in order to update fundamental packages such as NumPy. The first major update of NumPy, version 2.0.0, was released in 2024. If any of the packages in use depends on NumPy 1.x, the automatic installation of NumPy 2.0.0 via pip will fail. Manual installation, which is possible from the source, can break the existing behavior of downstream programs unintentionally, as some NumPy functions are incompatible with the old ones (see Appendix A).

Programming with Versions (PWV) [10, 13, 14] is a recent proposal designed to enable a gradual transition to new versions, thereby reducing update costs. The key ideas of PWV are (1) the simultaneous use of multiple versions, and (2) language mechanisms (i.e. types) that check version compatibilities. PWV languages ensure that programs use values created by compatible versions.

While previous research realized PWV in statically-typed languages, this research explores methods implementing PWV functionalities in dynamically-typed languages. To achieve this, we propose *dynamic version checking* (DVC) to alert when values of incompatible versions are used together dynamically . The DVC mechanism facilitates developers' communication regarding incompatibilities [9]; upstream developers specify compatibility for each function, allowing downstream developers to assess the impact of updates on their software through warnings.

We implement *Vython*, a Python subset with DVC, as a proof-of-concept. As a preliminary evaluation, we evaluate its runtime performance, and discuss optimizations and possible mechanisms that assist practical development.

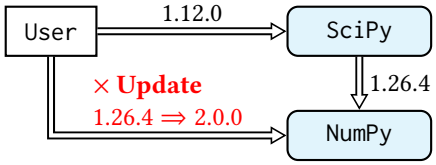

Figure 1. Dependencies of the `User` program.

```
class SciPy: # SciPy 1.12.0:
  def place_poles(A, B, poles):
    return NumPy().solve(..) # Using Numpy 1.26.4
```

```
def my_place_poles(A, B, poles): # User Program
  return NumPy().solve(..) # Using Numpy 2.0.0
NumPy().array_equal(
  my_place_poles(A, B, poles),
  scipy.place_poles(A, B, poles) ) # => False
```

Figure 2. A program that uses NumPy and SciPy in Python

2 Motivating Example

Consider a scenario where we update a user program that reimplements a function for solving pole placement problem[1] and test its behavior against `SciPy` [16] implementation. Figure 1 shows the dependencies of the `User` program. `User` depends on `SciPy` version 1.12.0, which indirectly depends

*Tokyo Institute of Technology until September 2024.

[1]This is a common task in control theory, placing closed-loop poles in desired locations to control the system response [15].

on NumPy 1.26.4, and `User` directly depends on NumPy and attempts to update it from version 1.26.4 to 2.0.0.

As shown in Figure 2, both `SciPy` and `User` use `solve` from NumPy[2]. In `User` (Figure 2 bottom), `my_place_poles` is implemented using `solve`, and its results are compared against the existing implementation in `SciPy`. `place_poles` in `SciPy` 1.12.0 (Figure 2 top) directly returns the result of `solve`. We try to update NumPy in the `User` project.

Updating NumPy via pip. This attempt fails as follows.

```
$ pip install numpy==2.0.0
ERROR: scipy 1.12.0 requires numpy<1.29.0,>=1.22.4, but you
    have numpy 2.0.0 which is incompatible.
```

The output error reports that this attempt has resulted in broken dependencies because the already installed `SciPy` is locked to NumPy versions below 1.29.0.

Updating NumPy from the Source. A natural solution to use NumPy 2.0.0 without waiting for `SciPy` updates is to separately use NumPy 1.26.4 for `SciPy` and 2.0.0 for `User`. Building NumPy from the source and dynamically importing specific versions makes this possible (see Appendix B). However, subtle differences between the two versions may result in an unintended behavior in the `User` program.

The `solve` implementation was incompatibly changed in the NumPy 2.0.0 release. As explained in Appendix A, the ambiguous broadcasting rule was corrected in 2.0.0, so `solve` in the two versions may return different outputs even with the same input. As a result, the test of `my_place_poles` against `place_poles` in Figure 2 line 5 fails, even if both implementations are logically the same.

Identifying the cause of this failure is challenging. Current build systems lack mechanisms to detect the mixed use of incompatible implementation versions. Additionally, such incorrect version usage is often reported as Python semantic errors, which do not provide the essential cause rooted in incompatibilities. Consequently, programmers must engage in tedious tasks such as reading release notes and reviewing implementations of all upstream packages.

3 Safely Use Multiple Versions in Vython

Vython is a python subset with the following features:

- **Using multiple versions in a code**: The programmer can selectively use multiple versions of a class definition by specifying a version when instantiating.
- **Dynamic version checking (DVC)**: Vython records information about the class and its version used for creating a value, ensuring that programs use values created by compatible combination of versions (objects).

Vython differentiates multiple versions of a class internally, allowing for their selective use. As shown in Figure 3, the current naive implementation requires version annotations in the surface language. Additionally, DVC is intended

[2]These programs are simplified, but are essentially identical to the actual implementation. For more details, see Appendix B.

```
class NumPy!1.26.4():
  def solve(self, A, B):
    return res
```

```
class NumPy!2.0.0():
  def solve(self, A, B):
    return incompatible(res,
        ..)
```

```
class SciPy!1.12.0():
  def place_poles(A, B, poles):
    return NumPy!1.26.4().solve(..) # Using Numpy 1.26.4
```

```
def my_place_poles(A, B, poles): # User Program
  return NumPy!2.0.0().solve(..) # Using Numpy 2.0.0
array_equal(
  my_place_poles(A, B, poles),
  SciPy!1.12.0().place_poles(A, B, poles) ) # => Warning!
```

Figure 3. A program that uses NumPy and SciPy in Vython

class	NumPy	Array
version	2.0.0	1.0.3
flag	True	False

Table 1. Version Table

to be enabled only in debug mode. Vython has a production mode that deploys programs without runtime checks.

Vython provides a mechanism for upstream developers to specify compatibility, which is utilized in DVC as follows.

Upstream Developer Specifies Compatibilities in Code. In Vython, upstream developers are responsible for specifying incompatibilities. In Numpy 2.0.0 (Figure 3 top right), the NumPy developer uses `incompatible()` to mark an expression as incompatible with previous versions. Additionally, upstream developers can provide guidance (as shown below) to help downstream developers. This information is recorded along with the class definition in the source code.

```
[Changed in 2.0.0] (How it differes from 1.26.4)
```

Notifying Downstream Developers of Incompatibility Causes. The downstream developer using both NumPy versions benefits from DVC and the guidance for updates specified by the NumPy developer. In the user program (Figure 3 bottom), the DVC mechanism reports runtime warnings (as shown below) on lines 3-5 because `array_equal` uses values derived from incompatible versions of the `solve` function.

```
Incompatible version usage found in Lines 3-5:
  - NumPy 1.26.4
  - NumPy 2.0.0
[Changed in version 2.0] `NumPy().solve(a,b)`:
  - If `b` is 1-dim, it is treated as a column vector (M,).
  - Otherwise, it is treated as a stack of (M, K) matrices.
  - Previously, `b` was treated as a stack of (M,) vectors
      if `b.ndim` equaled `a.ndim - 1`.
```

4 Implementation

4.1 Recording Compatibility Information in Values

Vython records the version information in a format called *Version Table* (VT). All values (objects) are assigned initialized VT upon their instantiation. When evaluating a program

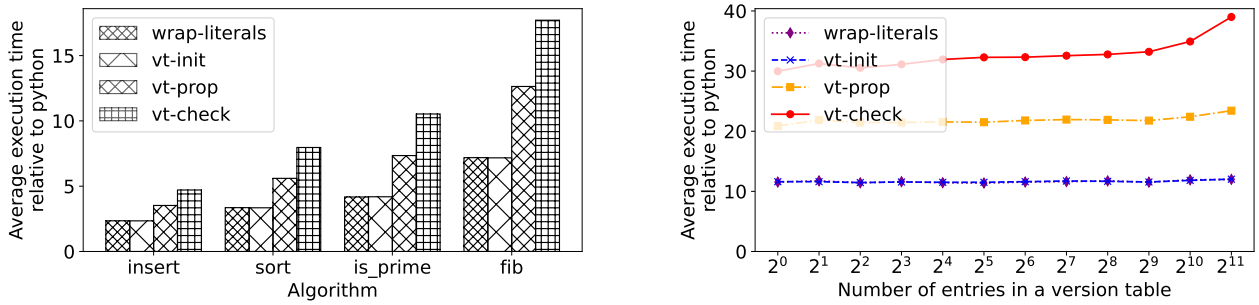

Figure 4. Overhead of the helper functions in Vython (left) for simple benchmarks and (right) for repeating additions 2000 times with the number of VT entries.

using multiple values, Vython concatenates their VTs and propagates it to the VT of the resulting value.

For example, Table 1 shows the VT for the result value of `my_place_poles(A, B, poles)` on line 4 in the user program in Figure 3. This VT records that the value depends on `Array` 1.0.3 and `NumPy` 2.0.0. The `flag` is a check flag indicating whether the value is potentially incompatible with other versions. The check flag is `False` by default and is set to `True` by a call to `incompatible()`.

4.2 Dynamic Version Checking Mechanism

Vython dynamically checks for compatibility by comparing VTs through a predefined helper function. This function is automatically invoked during the evaluation of prespecified method calls, including primitive operations by default.

For example, in lines 3-5 of Figure 3, DVC compares the VTs of the two arguments of `array_equal`. The first argument's VT indicates incompatibility with NumPy versions below 2.0.0 (as shown in Table 1), while the second's VT shows it derives from NumPy 1.26.4. DVC compares the VT's entries and detects the difference as an incompatibility.

4.3 Transpilation

The Vython transpiler transpiles Python programs with version annotations, as shown in Figure 3, into specific versions of Python programs. In the current implementation, we treat all values as objects with VTs for simplification; literals are transpiled into predefined classes and VT is implemented as an attribute of these objects. The transpiler also inserts global helper functions for VT initialization, VT propagation, and compatibility checking.

5 Evaluation

Settings. We conducted preliminary experiments on runtime performance. We run (1) simple benchmarks for several major algorithms[3] using a VT with a maximum of two entries, and (2) a program that repeats additions 2000 times, with the number of VT entries doubling from 2^0 to 2^{11}. These experiments were conducted with Python 3.12.1 on an Intel Core i5-10400F running Windows 11 23H2. We calculated the average over 10000 iterations for the following five cases.

1. **python**: Baseline, no proposed language feature.
2. **wrap-literals**: Compiling literals as with VTs.
3. **vt-init**: 2 + VT Initialization at object instantiations.
4. **vt-prop**: 3 + VT concatenation and propagation.
5. **vt-check (vython)**: 4 + Compatibility checking.

Discussion. Figure 4 (left) shows that the overhead is 18x relative to Python in the worst case (`fib` and `vt-check`), and the programs dominated by arithmetic and boolean operations, such as `fib` and `is_prime`, exhibit higher overhead than the other two. In comparison to other dynamic analysis tools for Python, such as DynaPyt [4], whose overhead ranges from 1.2x to 16x, this result is not overly excessive and indicate that it is acceptable for debugging purposes in terms of runtime performance. Figure 4 (right) shows that the overhead does not increase significantly as the VT size grows. These results imply that Vython is scalable, although further case studies are necessary to ensure the VT size remains $< 2^{11}$ for real-world applications.

6 Conclusion and Future Work

We implement Vython and conduct a preliminary evaluation. The results indicate that while the current implementation is prototypical, its performance is acceptable for debugging purposes. We plan to undertake the following future work.

Toward Better Feedback. Actual packages have multiple versions and evolve non-linearly [3], while the current DVC mechanism assumes two versions and linear evolution. By using tools to manage source code differences and incompatibilities, we can synthesize feedback that takes into account the history of updates.

Surface Language Design. The current Vython requires specifying class versions in the surface program. We plan to develop a method to automatically infer versions working on Python programs. This will help minimize the annotations given by downstream developers, identify dependencies on old versions, and automate updates.

[3]See Appendix C for more details.

Acknowledgments

The author would like to thank to the members of PRG, especially to Yudai Tanabe and Hidehiko Masuhara. This work was supported by JSPS KAKENHI Grant Numbers JP23K19961 and JP23K28058.

References

[1] Python Packaging Authority. 2024. pip: The PyPA recommended tool for installing Python packages. https://pip.pypa.io/

[2] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

[3] Reidar Conradi and Bernhard Westfechtel. 1998. Version models for software configuration management. *ACM Comput. Surv.* 30, 2 (jun 1998), 232–282. https://doi.org/10.1145/280277.280280

[4] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: a dynamic analysis framework for Python. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 760–771. https://doi.org/10.1145/3540250.3549126

[5] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. 2018. Efficient static checking of library updates. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 791–796. https://doi.org/10.1145/3236024.3275535

[6] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[7] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, and Kelly Blincoe. 2024. Understanding the Impact of APIs Behavioral Breaking Changes on Client Applications. *Proc. ACM Softw. Eng.* 1, FSE, Article 56 (jul 2024), 24 pages. https://doi.org/10.1145/3643782

[8] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (Feb. 2018), 384–417. https://doi.org/10.1007/s10664-017-9521-5

[9] Patrick Lam, Jens Dietrich, and David J. Pearce. 2020. Putting the semantics into semantic versioning. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Virtual, USA) *(Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 157–179. https://doi.org/10.1145/3426428.3426922

[10] Luthfan Anshar Lubis, Yudai Tanabe, Tomoyuki Aotani, and Hidehiko Masuhara. 2022. BatakJava: An Object-Oriented Programming Language with Versions. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, Auckland New Zealand, 222–234. https://doi.org/10.1145/3567512.3567531

[11] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*. IEEE Press, Urbana-Champaign, IL, USA, 84–94.

[12] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[13] Yudai Tanabe, Luthfan Anshar Lubis, Tomoyuki Aotani, and Hidehiko Masuhara. 2021. A Functional Programming Language with Versions. *The Art, Science, and Engineering of Programming* 6, 1 (July 2021), 5:1–5:30. https://doi.org/10.22152/programming-journal.org/2022/6/5

[14] Yudai Tanabe, Luthfan Anshar Lubis, Tomoyuki Aotani, and Hidehiko Masuhara. 2023. Compilation Semantics for a Programming Language with Versions. In *Programming Languages and Systems*, Chung-Kil Hur (Ed.). Springer Nature Singapore, Singapore, 3–23.

[15] SciPy Teams. 2024. place_poles — SciPy v1.14.0 Manual. https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.place_poles.html

[16] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[17] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 56 – 61. https://doi.org/10.25080/Majora-92bf1922-00a

A Incompatible Behaviours Between NumPy 2.0.0 and 1.26.4

This section outlines some of the major incompatibilities between NumPy 1.26.4 and NumPy 2.0.0. All 11 examples we collect, along with the scripts to reproduce them, are available on the GitHub repository (https://github.com/prg-titech/numpy_diff).

A.1 Incompatibilities in `numpy.linalg.solve`

The `numpy.linalg.solve` function solves a linear matrix equation. It solves the equation $ax = b$ for x, where a is a square matrix and b is a vector or matrix provided as arguments to the function. The implementation was changed in NumPy 2.0.0. The following note is from the official NumPy documentation:

> *Changed in version 2.0*: The b array is only treated as a shape `(M,)` column vector if it is exactly 1-dimensional. In all other instances it is treated as a stack of `(M, K)` matrices. Previously b would be treated as a stack of `(M,)` vectors if `b.ndim` was equal to `a.ndim - 1`.

As a result, when b is not strictly one-dimensional, the output of the `solve` function differs for the same input. For

example, consider the following program run with NumPy 1.26.4 and NumPy 2.0.0.

```
import numpy as np

# Shape (2, 2, 2)
a = np.array(
  [ [[3, 1], [1, 2]]
  , [[2, 1], [1, 3]] ])
# Shape (2, 2)
b = np.array(
  [ [9, 8]
  , [7, 10] ])

x = np.linalg.solve(a, b)
print(x)
```

When we run the above program with NumPy versions 1.26.4 and 2.0.0, we get the following different outputs due to incompatibility in broadcasting rules.

```
@ Running linalg_solve.py with numpy 1.26.4
[[2. 3. ]
 [2.2 2.6]]
@ Running linalg_solve.py with numpy 2.0.0
[[[2.2 1.2]
  [2.4 4.4]]

  [[4. 2.8]
  [1. 2.4]]]
```

The reason for this difference lies in how the `b` array is treated in different versions of NumPy. In version 1.26.4, if `b`'s number of dimensions (`b.ndim`) is equal to one less than the number of dimensions of `a` (`a.ndim - 1`), `b` is interpreted as a stack of `(M,)` vectors. This means that in version 1.26.4, the `b` array is treated as a stack of 1-dimensional vectors, each corresponding to a 2x2 matrix in `a`. Therefore, the program is interpreted as follows:

$$\begin{pmatrix}3 & 1\\ 1 & 2\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix}=\begin{pmatrix}9\\ 8\end{pmatrix},\quad \begin{pmatrix}2 & 1\\ 1 & 3\end{pmatrix}\begin{pmatrix}y_1\\ y_2\end{pmatrix}=\begin{pmatrix}7\\ 10\end{pmatrix}.$$

However, in version 2.0.0, the behavior was modified such that the `b` array is treated as a column vector only if it is strictly 1-dimensional. In all other cases, it is treated as a stack of `(M, K)` matrices. Consequently, for the given input, `b` is treated as a stack of 2-dimensional matrices. Therefore, the program is interpreted as follows:

$$\begin{pmatrix}3 & 1\\ 1 & 2\end{pmatrix}\begin{pmatrix}x_1 & x_2\\ x_3 & x_4\end{pmatrix}=\begin{pmatrix}9 & 8\\ 7 & 10\end{pmatrix},\quad \begin{pmatrix}2 & 1\\ 1 & 3\end{pmatrix}\begin{pmatrix}y_1 & y_2\\ y_3 & y_4\end{pmatrix}=\begin{pmatrix}9 & 8\\ 7 & 10\end{pmatrix}.$$

A.2 Incompatibilities in Other Functions

In addition to `numpy.linalg.solve`, NumPy 2.0.0 introduces several other backward-incompatible modifications. Among the programs we collected that produce different outputs solely due to version differences in NumPy, we list some notable input-output pairs below. For other examples where downstream developers might easily notice incompatibilities due to Python runtime errors, such as differences in output types, please refer to the repository.

`numpy.nonzero`. The function return the indices of the elements that are non-zero. The function previously ignored whitespace so that a string only containing whitespace was considered `False`, however, whitespace is now considered `True` in string arrays newly in NumPy 2.0.0.

```
import numpy as np

arr = np.array([' ', 'a', ''])
print(np.nonzero(arr))
```

```
@ Running nonzero.py with numpy 1.26.4
(array([1]),)
@ Running nonzero.py with numpy 2.0.0
(array([0, 1]),)
```

`numpy.linalg.lstsq`. The function returns the least squares solution to a linear matrix equation. The default value of the `rcond` (cut-off ratio) parameter in `lstsq` was changed in NumPy 2.0.0. This change introduces a subtle incompatibility: while most inputs yield the same output regardless of the NumPy version, inputs with elements near machine precision can produce different results depending on the NumPy version. The following example illustrates such a case.

```
import numpy as np

a = np.zeros((10**2, 2))
a[0, 0] = 1
a[m-1, 1] = 2.22e-16
b = np.zeros(m)
b[m-1] = 1

x, res, rank, s = np.linalg.lstsq(a, b)
print(...)
```

```
@ Running linalg_lstsq.py with numpy 1.26.4
Solution with default rcond: [0.0000000e+00 4.5045045e+15]
Residuals: [4.93038066e-32]
Rank: 2
Singular values: [1.00e+00 2.22e-16]
@ Running linalg_lstsq.py with numpy 2.0.0
Solution with default rcond: [0. 0.]
Residuals: []
Rank: 1
Singular values: [1.00e+00 2.22e-16]
```

`numpy.loadtxt` *and* `numpy.genfromtxt`. The functions provide readers for simly formatted files. Default encoding for these functions was changed in NumPy 2.0.0. Previously, these two functions selected `encoding=bytes` as the default parameter, but starting from version 2.0.0, it has been changed to `encoding=string`. As a result, programs that expect custom converters assuming a byte value will be broken by the update.

```
import numpy as np
import io
def custom_converter(byte_string):
  return float(byte_string.decode('utf-8'))

data = b"1.1\n2.2\n3.3\n"
with open('data.txt', 'wb') as f:
  f.write(data)

```

```
# Load the data using loadtxt with the custom converter
try:
  data = np.loadtxt('data.txt', converters={0:
       custom_converter})
  print(f"Data loaded successfully: {data}")
except Exception as e:
  print(f"An error occurred: {e}")
```

```
@ Running loadtxt_genfromtxt.py with numpy 1.26.4
Data loaded successfully: [1.1 2.2 3.3]
@ Running loadtxt_genfromtxt.py with numpy 2.0.0
An error occurred: could not convert string '1.1' to
      float64 at row 0, column 1.
```

B Dynamically Switching NumPy Versions

This section describes the reproduction of the motivating examples from Section 2 in actual Python programs. The complete source code and instrucations to reproduce the results of this paper are available on the GitHub repository (https://github.com/prg-titech/use-multi-versions).

B.1 Installing Multiple NumPy Versions from Sources

For example, to install numpy version 1.26.4 into a directory named numpy-1.26.4 using pip on a Linux OS, use the following command.

```
$ mkdir numpy-1.26.4
$ pip donwload numpy==1.26.4
$ pip install numpy-1.26.4- ... .whl -t numpy-1.26.4
```

B.2 Simultaneouslly Using Multiple NumPy Versions in Code

The following `load_numpy` function dynamically loads a specified version of NumPy. It takes a string representing the version, sets the appropriate NumPy path, and removes any cached instances of NumPy from `sys.modules`. The function then temporarily modifies the system path to include the specified version's path installed in the last section and imports the NumPy module from its initialization file. Finally, `load_numpy` returns the module object for the specified version of NumPy.

```
# version_dispatch.py
def load_numpy(version):
    if version == '1.26.4':
        numpy_path = os.path.abspath('numpy-1.26.4')
    elif version == '2.0.0':
        numpy_path = os.path.abspath('numpy-2.0.0')
    else:
        raise ValueError(f"Unsupported numpy version: {
            version}")

    # Clear cache
    if 'numpy' in sys.modules:
        del sys.modules['numpy']
    for mod_name in list(sys.modules):
        if mod_name.startswith('numpy'):
            del sys.modules[mod_name]

    # Set environment pathes
    original_path = sys.path.copy()
    sys.path.insert(0, numpy_path)

    try:
        numpy_init_path = os.path.join(numpy_path, 'numpy',
             '__init__.py')
        spec = importlib.util.spec_from_file_location("numpy
             ", numpy_init_path)
        if spec is None:
            raise ImportError(f"Cannot find numpy module in
                 {numpy_path}")

        numpy = importlib.util.module_from_spec(spec)
        spec.loader.exec_module(numpy)
    finally:
        # Restore sys.path
        sys.path = original_path
```

The following program shows the full version of the program shown in Figure 2. The implementation of the pole placement problem (`place_poles` and `my_place_poles`) has been simplified, as it is not the focus of this section. Using `./version_dispatch.py`, which defines the `load_numpy` function described in the previous subsection, `place_poles` is evaluated with NumPy 1.26.4 on line 26, and `my_place_poles` is evaluated with NumPy 2.0.0 on line 27. Finally, the results of the two functions are compared on line 29.

As mentioned in Section 2, despite `place_poles` and `my_place_poles` being identical implementations except for the NumPy version they use, the result evaluates to `False`.

```
1  from version_dispatch import load_numpy
2
3  # SciPy
4  class SciPy():
5    def place_poles(self, A, B, desired_poles):
6      np = load_numpy('1.26.4')
7      res = np.linalg.solve(A, B)
8      return res
9
10 # User Program
11 def my_place_poles(A, B, desired_poles):
12   np = load_numpy('2.0.0')
13   res = np.linalg.solve(A, B)
14   return res
15
16 def main():
17   np = load_numpy('2.0.0')
18   A = np.array(
19     [ [[3, 1], [1, 2]]
20     , [[2, 1], [1, 3]] ])
21   B = np.array(
22     [ [9, 8]
23     , [7, 10] ])
24   desired_poles = np.array([-1.0, -2.0])
25
26   expect = SciPy().place_poles(A,B,desired_poles).tolist()
27   actual = my_place_poles(A,B,desired_poles).tolist()
28
29   test = np.array_equal(expect, actual) # => False
30
31 main()
```

C Programs Used for Simple Benchmarks

`insert.py`. This program inserts one thousand `Node` instances to a binary tree.

```
1  class Node!1():
2    def __init__(self, value):
3        self.value = value
4        self.left = None
5        self.right = None
6
7    def insert_right(self, v):
8        if self.right == None:
9            self.right = Node!1(v)
10       else:
11           self.right.insert(v)
12
13   def insert_left(self, v):
14       if self.left == None:
15           self.left = Node!1(v)
16       else:
17           self.left.insert(v)
18
19   def insert(self, v):
20       if(self.value <= v):
21           self.insert_right(v)
22       else:
23           self.insert_left(v)
24
25 root = Node!1(5)
26 a = [...] # Array of 1000 elements, random numbers
       between 1 and 10000
27 for i in a:
28     root.insert(i)
```

`sort.py`. This program performs a merge sort on a Python list of 1000 elements.

```
1  def sort(list):
2    if len(list) < 1:
3        return []
4    elif len(list) == 1:
5        return list
6    pivot = list[0]
7    lower_list = []
8    upper_list = []
9    middle_list = []
10
11   for item in list:
12       if item < pivot:
13           lower_list.append(item)
14       elif item > pivot:
15           upper_list.append(item)
16       else:
17           middle_list.append(item)
18
19   sorted_lower_list = sort(lower_list)
20   sorted_upper_list = sort(upper_list)
21
22   return sorted_lower_list + middle_list +
         sorted_upper_list
23
24 a = [...] # Array of 1000 elements, random numbers
       between 1 and 10000
25 sort(a)
```

`is_prime.py`. This program uses a simple algorithm to determine the primality of 128456903.

```
1  def is_prime(n):
2    if n <= 1:
3        return False
4    if n == 2 or n == 3:
5        return True
```

```
  if n % 2 == 0 or n % 3 == 0:
      return False
  return is_prime_recursive(n, 5)

def is_prime_recursive(n, i):
    if i * i > n:
        return True
    if n % i == 0 or n % (i + 2) == 0:
        return False
    return is_prime_recursive(n, i + 6)

is_prime(128456903)
```

`fib.py`. This program recursively computes the 20th Fibonacci number.

```
def fib(n):
  if n<=2:
      return 1
  else:
      return fib(n-1) + fib(n-2)

fib(20)
```