A New Python API for Webots Robotics Simulations

Justin C. Fisher[‡][*]

Abstract—Webots is a popular open-source package for 3D robotics simulations. It can also be used as a 3D interactive environment for other physics-based modeling, virtual reality, teaching or games. Webots has provided a simple API allowing Python programs to control robots and/or the simulated world, but this API is inefficient and does not provide many "pythonic" conveniences. A new Python API for Webots is presented that is more efficient and provides a more intuitive, easily usable, and "pythonic" interface.

Index Terms—Webots, Python, Robotics, Robot Operating System (ROS), Open Dynamics Engine (ODE), 3D Physics Simulation

1. Introduction

Webots is a popular open-source package for 3D robotics simulations [Mic01], [Webots]. It can also be used as a 3D interactive environment for other physics-based modeling, virtual reality, teaching or games. Webots uses the Open Dynamics Engine [ODE], which allows physical simulations of Newtonian bodies, collisions, joints, springs, friction, and fluid dynamics. Webots provides the means to simulate a wide variety of robot components, including motors, actuators, wheels, treads, grippers, light sensors, ultrasound sensors, pressure sensors, range finders, radar, lidar, and cameras (with many of these sensors drawing their inputs from GPU processing of the simulation). A typical simulation will involve one or more robots, each with somewhere between 3 and 30 moving parts (though more would be possible), each running its own controller program to process information taken in by its sensors to determine what control signals to send to its devices. A simulated world typically involves a ground surface (which may be a sloping polygon mesh) and dozens of walls, obstacles, and/or other objects, which may be stationary or moving in the physics simulation.

Webots has historically provided a simple Python API, allowing Python programs to control individual robots or the simulated world. This Python API is a thin wrapper over a C++ API, which itself is a wrapper over Webots' core C API. These nested layers of API-wrapping are inefficient. Furthermore, this API is not very "pythonic" and did not provide many of the conveniences that help to make development in Python be fast, intuitive, and easy to learn. This paper presents a new Python API [NewAPI01] that more efficiently interfaces directly with the Webots C API and provides a more intuitive, easily usable, and "pythonic" interface for controlling Webots robots and simulations.

* Corresponding author: fisher@smu.edu
‡ Southern Methodist University, Department of Philosophy

In qualitative terms, the old API feels like one is awkwardly using Python to call C and C++ functions, whereas the new API feels much simpler, much easier, and like it is fully intended for Python. Here is a representative (but far from comprehensive) list of examples:

- Unlike the old API, the new API contains helpful Python type annotations and docstrings.
- Webots employs many vectors, e.g., for 3D positions, 4D rotations, and RGB colors. The old API typically treats these as lists or integers (24-bit colors). In the new API these are Vector objects, with conveniently addressable components (e.g. `vector.x` or `color.red`), convenient helper methods like `vector.magnitude` and `vector.unit_vector`, and overloaded vector arithmetic operations, akin to (and interoperable with) NumPy arrays.
- The new API also provides easy interfacing between high-resolution Webots sensors (like cameras and Lidar) and Numpy arrays, to make it much more convenient to use Webots with popular Python packages like Numpy [NumPy], [Har01], Scipy [Scipy], [Vir01], PIL/PILLOW [PIL] or OpenCV [OpenCV], [Brad01]. For example, converting a Webots camera image to a NumPy array is now as simple as `camera.array` and this now allows the array to share memory with the camera, making this extremely fast regardless of image size.
- The old API often requires that all function parameters be given explicitly in every call, whereas the new API gives many parameters commonly used default values, allowing them often to be omitted, and keyword arguments to be used where needed.
- Most attributes are now accessible (and alterable, when applicable) by pythonic properties like `motor.velocity`.
- Many devices now have Python methods like `__bool__` overloaded in intuitive ways. E.g., you can now use `if bumper` to detect if a bumper has been pressed, rather than the old `if bumper.getValue()`.
- Pythonic container-like interfaces are now provided. You may now use `for target in radar` to iterate through the various targets a radar device has detected or `for packet in receiver` to iterate through communication packets that a receiver device has received (and it now automatically handles a wide variety of Python objects, not just strings).
- The old API requires supervisor controllers to use a wide variety of separate functions to traverse and interact with the simulation's scene tree, including dif-

ferent functions for different VRML datatypes (like `SFVec3f` or `MFInt32`). The new API automatically handles these datatypes and translates intuitive Python syntax (like dot-notation and square-bracket indexing) to the Webots equivalents. E.g., you can now move a particular crate 1 meter in the x direction using a command like `world.CRATES[3].translation += [1,0,0]`. Under the old API, this would require numerous function calls (calling `getNodeFromDef` to find the CRATES node, `getMFNode` to find the child with index 3, `getSFField` to find its translation field, and `getSFVec3f` to retrieve that field's value, then some list manipulation to alter the x-component of that value, and finally a call to `setSFVec3f` to set the new value).

As another example illustrating how much easier the new API is to use, here are two lines from Webots' sample `supervisor_draw_trail`, as it would appear in the old Python API.

```
f = supervisor.getField(supervisor.getRoot(),
                        "children")
f.importMFNodeFromString(-1, trail_plan)
```

And here is how that looks written using the new API:

```
world.children.append(trail_plan)
```

The new API is mostly backwards-compatible with the old Python Webots API, and provides an option to display deprecation warnings with helpful advice for changing to the new API.

The new Python API is planned for inclusion in an upcoming Webots release, to replace the old one. In the meantime, an early-access version is available, distributed under Apache 2.0 licence, the same permissibe open-source license that Webots is distributed under.

In what follows, the history and motivation for this new API is discussed, including its use in teaching an interdisciplinary undergraduate Cognitive Science course called Minds, Brains and Robotics. Some of the design decisions for the new API are discussed, which will not only aid in understanding it, but also have broader relevance to parallel dilemmas that face many other software developers. And some metrics are given to quantify how the new API has improved over the old.

2. History and Motivation.

Much of this new API was developed by the author in the course of teaching an interdisciplinary Southern Methodist University undergraduate Cognitive Science course entitled Minds, Brains and Robotics (PHIL 3316). Before the Covid pandemic, this course had involved lab activities where students build and program physical robots. The pandemic forced these activities to become virtual. Fortunately, Webots simulations actually have many advantages over physical robots, including not requiring any specialized hardware (beyond a decent personal computer), making much more interesting uses of altitude rather than having the robots confined to a safely flat surface, allowing robots to engage in dangerous or destructive activities that would be risky or expensive with physical hardware, allowing a much broader array of sensors including high-resolution cameras, and enabling full-fledged neural network and computational vision simulations. For example, an early activity in this class involves building Braitenburg-style vehicles [Bra01] that use light sensors and cameras to detect a lamp carried by a hovering drone, as well as ultrasound and touch sensors to detect obstables. Using these sensors, the robots navigate towards the lamp in a cluttered playground sandbox that includes sloping sand, an exterior wall, and various obstacles including a puddle of water and platforms from which robots may fall.

This interdisciplinary class draws students with diverse backgrounds, and programming skills. Accomodating those with fewer skills required simplifying many of the complexities of the old Webots API. It also required setting up tools to use Webots "supervisor" powers to help manipulate the simulated world, e.g. to provide students easier customization options for their robots. The old Webots API makes the use of such supervisor powers tedious and difficult, even for experienced coders, so this practically required developing new tools to streamline the process. These factors led to the development of an interface that would be much easier for novice students to adapt to, and that would make it much easier for an experienced programmer to make much use of supervisor powers to manipulate the simulated world. Discussion of this with the core Webots development team then led to the decision to incorporate these improvements into Webots, where they can be of benefit to a much broader community.

3. Design Decisions.

This section discusses some design decisions that arose in developing this API, and discusses the factors that drove these decisions. This may help give the reader a better understanding of this API, and also of relevant considerations that would arise in many other development scenarios.

3.1. Shifting from functions to properties.

The old Python API for Webots consists largely of methods like `motor.getVelocity()` and `motor.setVelocity(new_velocity)`. In the new API these have quite uniformly been changed to Python properties, so these purposes are now accomplished with `motor.velocity` and `motor.velocity = new_velocity`.

Reduction of wordiness and punctuation helps to make programs easier to read and to understand, and it reduces the cognitive load on coders. However, there are also drawbacks.

One drawback is that properties can give the mistaken impression that some attributes are computationally cheap to get or set. In cases where this impression would be misleading, more traditional method calls were retained and/or the comparative expense of the operation was clearly documented.

Two other drawbacks are related. One is that inviting ordinary users to assign properties to API objects might lead them to assign other attributes that could cause problems. Since Python lacks true privacy protections, it has always faced this sort of worry, but this worry becomes even worse when users start to feel familiar moving beyond just using defined methods to interact with an object.

Relatedly, Python debugging provides direct feedback in cases where a user misspells `motor.setFoo(v)` but not when someone mispells 'motor.foo = v'. If a user inadvertently types `motor.setFool(v)` they will get an `AttributeError` noting that `motor` lacks a `setFool` attribute. But if a user inadvertently types `motor.fool = v`, then Python will silently create a new `.fool` attribute for `motor` and the user will often have no idea what has gone wrong.

These two drawbacks both involve users setting an attribute they shouldn't: either an attribute that has another purpose, or one that doesn't. Defenses against the first include "hiding" important attributes behind a leading "_", or protecting them with a Python property, which can also help provide useful doc-strings. Unfortunately it's much harder to protect against misspellings in this piece-meal fashion.

This led to the decision to have robot devices like motors and cameras employ a blanket `__setattr__` that will generate warnings if non-property attributes of devices are set from outside the module. So the user who inadvertently types `motor.foo1 = v` will immediately be warned of their mistake. This does incur a performance cost, but that cost is often worthwhile when it saves development time and frustration. For cases when performance is crucial, and/or a user wants to live dangerously and meddle inside API objects, this layer of protection can be deactivated.

An alternative approach, suggested by Matthew Feickert, would have been to use `__slots__` rather than an ordinary `__dict__` to store device attributes, which would also have the effect of raising an error if users attempt to modify unexpected attributes. Not having a `__dict__` can make it harder to do some things like cached properties and multiple inheritance. But in cases where such issues don't arise or can be worked around, readers facing similar challenges may find `__slots__` to be a preferable solution.

3.2 Backwards Compatibility.

The new API offers many new ways of doing things, many of which would seem "better" by most metrics, with the main drawback being just that they differ from old ways. The possibility of making a clean break from the old API was considered, but that would stop old code from working, alienate veteran users, and risk causing a schism akin to the deep one that arose between Python 2 and Python 3 communities when Python 3 opted against backwards compatibility.

Another option would have been to refrain from adding a "new-and-better" feature to avoid introducing redundancies or backward incompatibilities. But that has obvious drawbacks too.

Instead, a compromise was typically adopted: to provide both the "new-and-better" way and the "worse-old" way. This redundancy was eased by shifting from `getFoo` / `setFoo` methods to properties, and from `CamelCase` to pythonic `snake_case`, which reduced the number of name collisions between old and new. Employing the "worse-old" way leads to a deprecation warning that includes helpful advice regarding shifting to the "new-and-better" way of doing things. This may help users to transition more gradually to the new ways, or they can shut these warnings off to help preserve good will, and hopefully avoid a schism.

3.3 Separating `robot` and `world`.

In Webots there is a distinction between "ordinary robots" whose capabilities are generally limited to using the robot's own devices, and "supervisor robots" who share those capabilities, but also have virtual omniscience and omnipotence over most aspects of the simulated world. In the old API, supervisor controller programs import a `Supervisor` subclass of `Robot`, but typically still call this unusually powerful robot `robot`, which has led to many confusions.

In the new API these two sorts of powers are strictly separated. Importing `robot` provides an object that can be used to control the devices in the robot itself. Importing `world` provides an object that can be used to observe and enact changes anywhere in the simulated world (presuming that the controller has such permissions, of course). In many use cases, supervisor robots don't actually have bodies and devices of their own, and just use their supervisor powers incorporeally, so all they will need is `world`. In the case where a robot's controller wants to exert both forms of control, it can import both `robot` to control its own body, and `world` to control the rest of the world.

This distinction helps to make things more intuitively clear. It also frees `world` from having all the properties and methods that `robot` has, which in turn reduces the risk of name-collisions as `world` takes on the role of serving as the root of the proxy scene tree. In the new API, `world.children` refers to the `children` field of the root of the scene tree which contains (almost) all of the simulated world, `world.WorldInfo` refers to one of these children, a `WorldInfo` node, and `world.ROBOT2` dynamically returns a node within the world whose Webots DEF-name is "ROBOT2". These uses of `world` would have been much less intuitive if users thought of `world` as being a special sort of robot, rather than as being their handle on controlling the simulated world. Other sorts of supervisor functionality also are very intuitively associated with `world`, like `world.save(filename)` to save the state of the simulated world, or `world.mode = 'PAUSE'`.

Having `world.attributes` dynamically fetch nodes and fields from the scene tree did come with some drawbacks. There is a risk of name-collisions, though these are rare since Webots field-names are known in advance, and nodes are typically sought by ALL-CAPS DEF-names, which won't collide with `world`'s lower-case and MixedCase attributes. Linters like MyPy and PyCharm also cannot anticipate such dynamic references, which is unfortunate, but does not stop such dynamic references from being extremely useful.

4. Readability Metrics

A main advantage of the new API is that it allows Webots controllers to be written in a manner that is easier for coders to read, write, and understand. Qualitatively, this difference becomes quite apparent upon a cursory inspection of examples like the one given in section 1. As another representative example, here are three lines from Webots' included `supervisor_draw_trail` sample as they would appear in the old Python API:

```
trail_node = world.getFromDef("TRAIL")
point_field = trail_node.getField("coord")\
                         .getSFNode()\
                         .getField("point")
index_field = trail_node.getField("coordIndex")
```

And here is their equivalent in the new API:

```
point_field = world.TRAIL.coord.point
index_field = world.TRAIL.coordIndex
```

Brief inspection should reveal that the latter code is much easier to read, write and understand, not just because it is shorter, but also because its punctuation is limited to standard Python syntax for traversing attributes of objects, because it reduces the need to introduce new variables like `trail_node` for things that it already makes easy to reference (via `world.TRAIL`, which the new API automatically caches for fast repeat reference), and because it invisibly handles selecting appropriate C-API functions like `getField` and `getSFNode`, saving the user from needing to learn and remember all these functions (of which there are many).

Metric	New API	Old API
Lines of Code (with blanks, comments)	43	49
Source Lines of Code (without those)	29	35
Logical Lines of Code (single commands)	27	38
Cyclomatic Complexity	5 (A)	8 (B)

TABLE 1
Length and Complexity Metrics. Raw measures for `supervisor_draw_trail` as it would be written with the new Python API for Webots or the old Python API for Webots. The "lines of codes" measures differ with respect to how they count blank lines, comments, and lines that combine multiple commands. Cyclomatic complexity measures the number of potential branching points in the code.

Halstead Metric	New API	Old API
Vocabulary = (n1)operators+(n2)operands	18	54
Length = (N1)operator + (N2)operand instances	38	99
Volume = Length * $\log_2$(Vocabulary)	158	570
Difficulty = (n1 * N2) / (2 * n2)	4.62	4.77
Effort = Difficulty * Volume	731	2715
Time = Effort / 18	41	151
Bugs = Volume / 3000	0.05	0.19

TABLE 2
Halstead Metrics. Halstead metrics for `supervisor_draw_trail` as it would be written with the new and old Python API's for Webots. Lower numbers are commonly construed as being better.

This intuitive impression is confirmed by automated metrics for code readability. The measures in what follows consider the full `supervisor_draw_trail` sample controller (from which the above snippet was drawn), since this is the Webots sample controller that makes the most sustained use of supervisor functionality to perform a fairly plausible supervisor task (maintaining the position of a streamer that trails behind the robot). Webots provides this sample controller in C [SDTC], but it was re-implemented using both the Old Python API and the New Python API [Metrics], maintaining straightforward correspondence between the two, with the only differences being directly due to the differences in the API's.

Some raw measures for the two controllers are shown in Table 1. These were gathered using the Radon code-analysis tools [Radon]. (These metrics, as well as those below, may be reproduced by (1) installing Radon [Radon], (2) downloading the source files to compare and the script for computing Metrics [Metrics], (3) ensuring that the path at the top of the script refers to the local location of the source files to be compared, and (4) running this script.) Multiple metrics are reported because theorists disagree about which are most relevant in assessing code readability, because some of these play a role in computing other metrics discussed below, and because this may help to allay potential worries that a few favorable metrics might have been cherry-picked. This paper provides some explanation of these metrics and of their potential significance, while remaining neutral regarding which, if any, of these metrics is best.

The "lines of code" measures reflect that the new API makes it easier to do more things with less code. The measures differ in how they count blank lines, comments, multi-line statements, and multi-statement lines like `if p: q()`. Line counts can be misleading, especially when the code with fewer lines has longer lines, though upcoming measures will show that that is not the case here.

Cyclomatic Complexity counts the number of potential branching points that appear within the code, like `if`, `while` and `for`. [McC01] Cyclomatic Complexity is strongly correlated with other plausible measures of code readability involving indentation structure [Hin01]. The new API's score is lower/"better" due to its automatically converting vector-like values to the format needed for importing new nodes into the Webots simulation, and due to its automatic caching allowing a simpler loop to remove unwanted nodes. By Radon's reckoning this difference in complexity already gives the old API a "B" grade, as compared to the new API's "A". These complexity measures would surely rise in more complex controllers employed in larger simulations, but they would rise less quickly under the new API, since it provides many simpler ways of doing things, and need never do any worse since it provides backwards-compatible options.

Another collection of classic measures of code readability was developed by Halstead. [Hal01] These measures (especially volume) have been shown to correlate with human assessments of code readability [Bus01], [Pos01]. These measures generally penalize a program for using a "vocabulary" involving more operators and operands. Table 2 shows these metrics, as computed by Radon. (Again all measures are reported, while remaining neutral about which are most significant.) The new API scores significantly lower/"better" on these metrics, due in large part to its automatically selecting among many different C-API calls without these needing to appear in the user's code. E.g. having `motor.velocity` as a unified property involves fewer unique names than having users write both `setVelocity()` and `getVelocity()`, and often forming a third local `velocity` variable. And having `world.children[-1]` access the last child that field in the simulation saves having to count `getField`, and `getMFNode` in the vocabulary, and often also saves forming additional local variables for nodes or fields gotten in this way. Both of these factors also help the new API to greatly reduce parentheses counts.

Lastly, the Maintainability Index and variants thereof are intended to measure of how easy to support and change source code is. [Oman01] Variants of the Maintainability Index are commonly used, including in Microsoft Visual Studio. These measures combine Halstead Volume, Source Lines of Code, and Cyclomatic Complexity, all mentioned above, and two variants (SEI and Radon) also provide credit for percentage of comment lines. (Both samples compared here include 5 comment lines, but these compose a higher percentage of the new API's shorter code). Different versions of this measure weight and curve these factors somewhat differently, but since the new API outperforms the old on each factor, all versions agree that it gets the higher/"better" score, as shown in Table 3. (These measures were computed based on the input components as counted by Radon.)

There are potential concerns about each of these measures of code readability, and one can easily imagine playing a form of "code golf" to optimize some of these scores without actually improving readability (though it would be difficult to do this for all scores at once). Fortunately, most plausible measures of readability have been observed to be strongly correlated across ordinary cases, [Pos01] so the clear and unanimous agreement between these measures is a strong confirmation that the new API is indeed

Maintainability Index version	New API	Old API
Original [Oman01]	89	79
Software Engineering Institute	78	62
Microsoft Visual Studio	52	46
Radon	82	75

TABLE 3
Maintainability Index Metrics. Maintainability Index metrics for `supervisor_draw_trail` as it would be written with the new and old versions of the Python API for Webots, according to different versions of the Maintainability Index. Higher numbers are commonly construed as being better.

more readable. Other plausible measures of readability would take into account factors like whether the operands are ordinary English words, [Sca01] or how deeply nested (or indented) the code ends up being, [Hin01] both of which would also favor the new API. So the mathematics confirm what was likely obvious from visual comparison of code samples above, that the new API is indeed more "readable" than the old.

5. Conclusions

A new Python API for Webots robotic simulations was presented. It more efficiently interfaces directly with the Webots C API and provides a more intuitive, easily usable, and "pythonic" interface for controlling Webots robots and simulations. Motivations for the API and some of its design decisions were discussed, including decisions use python properties, to add new functionality alongside deprecated backwards compatibility, and to separate robot and supervisor/world functionality. Advantages of the new API were discussed and quantified using automated code readability metrics.

More Information

An early-access version of the new API and a variety of sample programs and metric computations: https://github.com/Justin-Fisher/new_python_api_for_webots

Lengthy discussion of the new API and its planned inclusion in Webots: https://github.com/cyberbotics/webots/pull/3801

Webots home page, including free download of Webots: https://cyberbotics.com/

References

[Brad01] Bradski, G. The OpenCV Library. Dr Dobb's Journal of Software Tools. 2000.

[Bra01] Braitenberg, V. *Vehicles: Experiments in synthetic psychology.* Cambridge, MA: MIT Press. 1984.

[Bus01] Buse, R and W Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4): 546-58. 2010. doi: 10.1109/TSE.2009.70.

[Metrics] Fisher, J. Readability Metrics for a New Python API for Webots Robotics Simulations. 2022. doi: 10.5281/zenodo.6813819.

[Hal01] Halstead, M. *Elements of software science.* Elsevier New York. 1977.

[Har01] Harris, C., K. Millman, S. van der Walt, et al. Array programming with NumPy. *Nature* 585, 357–62. 2020. doi: 10.1038/s41586-020-2649-2.

[Hin01] Hindle, A, MW Godfrey and RC Holt. "Reading beside the lines: Indentation as a proxy for complexity metric." Program Comprehension. The 16th IEEE International Conference, 133-42. 2008. doi: 10.1109/icpc.2008.13.

[McC01] McCabe, TJ. "A Complexity Measure" , 2(4): 308-320. 1976.

[Mic01] Michel, O. "Webots: Professional Mobile Robot Simulation. *Journal of Advanced Robotics Systems.* 1(1): 39-42. 2004. doi: 10.5772/5618.

[NewAPI01] https://github.com/Justin-Fisher/new_python_api_for_webots

[NumPy] Numerical Python (NumPy). https://www.numpy.org

[ODE] Open Dynamics Engine. https://www.ode.org/

[Oman01] Oman, P and J Hagemeister. "Metrics for assessing a software system's maintainability," *Proceedings Conference on Software Maintenance*, 337-44. 1992. doi: 10.1109/ICSM.1992.242525.

[OpenCV] Open Source Computer Vision Library for Python. https://github.com/opencv/opencv-python

[PIL] Python Imaging Library. https://python-pillow.org/

[Pos01] Posnet, D, A Hindle and P Devanbu. "A simpler model of software readability." *Proceedings of the 8th working conference on mining software repositories*, 73-82. 2011.

[Radon] Radon. https://radon.readthedocs.io/en/latest/index.html

[Sca01] Scalabrino, S, M Linares-Vasquez, R Oliveto and D Poshyvanyk. "A Comprehensive Model for Code Readability." *Jounal of Software: Evolution and Process*, 1-29. 2017. doi: 10.1002/smr.1958.

[Scipy] https://www.scipy.org

[SDTC] https://cyberbotics.com/doc/guide/samples-howto#supervisor_draw_trail-wbt

[SDTNew] https://github.com/Justin-Fisher/new_python_api_for_webots/blob/d180bcc7f505f8168246bee379f8067dfaf373ea/webots_new_python_api_samples/controllers/supervisor_draw_trail_python/supervisor_draw_trail_new_api_bare_bones.py

[SDTOld] https://github.com/Justin-Fisher/new_python_api_for_webots/blob/d180bcc7f505f8168246bee379f8067dfaf373ea/webots_new_python_api_samples/controllers/supervisor_draw_trail_python/supervisor_draw_trail_old_api_bare_bones.py

[Vir01] Virtanen, P, R. Gommers, T. Oliphant, et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17(3), 261-72. 2020. doi: 10.1038/s41592-019-0686-2.

[Webots] Webots Open Source Robotic Simulator. https://cyberbotics.com/