



psychoco: all-inclusive Python bindings for the Choco-solver constraint programming library

Dimitri Justeau-Allaire ¹ and Charles Prud'homme ²✉

¹ AMAP, Univ Montpellier, CIRAD, CNRS, INRAE, IRD, Montpellier, France ² TASC, IMT-Atlantique, LS2N-CNRS, Nantes, France ✉ Corresponding author

DOI: [10.21105/joss.08847](https://doi.org/10.21105/joss.08847)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [Daniel S. Katz](#) 

Reviewers:

- [@IgnaceBleux](#)
- [@skadio](#)

Submitted: 12 August 2025

Published: 28 September 2025

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](#)).

Summary

Constraint Programming (CP) is a well-established and powerful Artificial Intelligence (AI) paradigm for modelling and solving complex combinatorial problems ([Rossi et al., 2006](#)). Many CP solvers are currently available, and despite a generally shared common base, each solver exhibits specific features that make it more or less suited to certain types of problems and tasks. Performance and flexibility are important features of CP solvers, which is why most state-of-the-art solvers rely on statically typed and compiled programming languages, such as Java or C++. Because of this, CP has long remained a niche field that is difficult for non-specialists to access. Recently, the emergence of high-level, solver-independent modelling languages such as MiniZinc ([Nethercote et al., 2007](#)), XCSP³ ([Audemard et al., 2020](#)), and CPMpy ([Guns, 2019](#)) has made CP more accessible by allowing users to seamlessly use state-of-the-art solvers from user-friendly interpreted languages such as Python. To make CP even more accessible to a wider audience, we developed psychoco, a Python library that provides an all-inclusive binding to the Java Choco-solver library ([Prud'homme & Fages, 2022](#)). By all-inclusive, we mean that psychoco has no external dependencies and does not require the installation of Choco-solver or Java on the user's system. The choice of Python was motivated by its widespread use in the data science and AI communities, as well as its extensive use in education. The psychoco Python library supports almost all features of Choco-solver, is regularly updated, and is automatically built and distributed through PyPI for Linux, Windows, and MacOSX at each release. As a result, psychoco can seamlessly integrate into high-level constraint modelling Python libraries such as CPMpy ([Guns, 2019](#)) and PyCSP³ ([Lecoutre & Szczepanski, 2024](#)). Moreover, users who need to use features specific to Choco-solver (e.g., graph variables and constraints) can now rely on psychoco without prior knowledge of Java programming. We believe that along with initiatives such as CPMpy and PyCSP, the availability of CP technologies in the Python ecosystem will foster new uses and the appropriation of CP by a wider scientific and industrial public.

Statement of need

Constraint programming (CP) offers an expressive and flexible modelling paradigm which has proven efficient and useful in many industrial and academic applications: production optimization, aircraft scheduling, nurse scheduling ([Wallace, 1996](#)), music ([Hooker, 2016](#)), cryptography ([Gerault et al., 2016](#)), bioinformatics ([Barahona et al., 2011](#)), biodiversity conservation ([Deléglise et al., 2024](#)), agroecology ([Challand et al., 2025](#)), wine blending ([Vismara et al., 2016](#)), etc. However, most CP solvers are difficult for non-specialists to access because they mostly rely on statically typed and compiled programming languages such as Java or C++. As most data science and AI technologies are available in the Python ecosystem, it seems timely to make CP technologies more easily accessible in Python. High-level Python modelling libraries such as CPMpy ([Guns, 2019](#)) and PyCSP³ ([Lecoutre & Szczepanski, 2024](#))

have opened up many perspectives in this direction, but several solvers still require a separate installation, especially commercial ones and those based on JVM (Java Virtual Machine) languages, such as Choco-solver. For most Python users, this can be an obstacle. It also limits direct solver access and the use of specific features and fine-tuning options that may not be available in high-level modelling libraries. Making Choco-solver more accessible to Python users and facilitating its integration as a backend solver into high-level modelling libraries were the main motivations for the creation of pychoco. In addition, the widespread use of Python in education was also an argument in favour of pychoco's implementation.

Design

For several years, the main obstacle to implementing Python bindings for Choco-solver was the necessity to set up communication between the Python interpreter and the Java Virtual Machine (JVM). Indeed, we believe that the main interest of such bindings was to offer Python users a way to use Choco-solver without installing the JVM. The [GraalVM](#) project removed this obstacle with the ahead-of-time Native image Java compilation feature. Inspired by the work of (Michail et al., 2020) to make Python bindings for the JGraphT Java library, we implemented [choco-solver-capi](#), which contains entry points to the Choco-solver library that GraalVM compiles as a shared C library. This shared library is embedded into pychoco with the [SWIG](#) wrapper. pychoco's API relies on this SWIG interface and has been designed to mirror the main concepts of the Choco-solver API while simplifying its usage in a Pythonic way. We implemented pychoco with software quality standards: unit tests, code review, and continuous integration. We also rely on the [cibuildwheel](#) Python library to automatically build and publish Python wheels on PyPI for Windows, MacOSX, and Linux, from Python 3.6 to Python 3.13. Finally, in addition to an comprehensive code documentation, we integrated pychoco code snippets in Choco-solver online documentation (<https://choco-solver.org/>) and we designed a Cheat Sheet to summarize the main features of pychoco's API [Figure 1](#).

The pychoco Python library for Constraint Programming

Learn more about Choco, pychoco, and Constraint Programming at <https://choco-solver.org/>

pychoco

The **pychoco** library provides Python bindings to the Choco Constraint Programming solver. It relies on a native build of the original Java Choco library and is available through PyPI.

Installing pychoco

```
$ pip install pychoco
```

in a terminal

Loading pychoco

```
>>> from pychoco import *
```

in Python

The Model object

The **Model** object is the key component of Choco's and pychoco's API. It provides access to variables, constraints, and to the Solver.

Instantiating a Model

```
>>> m = Model() # default constructor
>>> m2 = Model("my model") # named
```

You can see the API using autocompletion from the Python console or your IDE.

```

name?
@ name                                Model
get_solver(solver)                     Model
vars(self, intvars, boolvars, opt...)  IntConstraintFactory
intvars(self, size, lb, ub, name, bound, variableFactory)
boolvars(self, lb, ub, name, bound, name, variableFactory)
set_objective(self, objective, maximize) Model
n_values(self, intvars, n_values)      IntConstraintFactory
all_different(self, intvars)            IntConstraintFactory
boolvars(self, size, value, name)       VariableFactory
boolvar(self, value, name)              VariableFactory
new_clause_true(self, boolvars)         IntFactory
add_clause_topo(self, true)             IntFactory
add_clause(self, pos_list, neg_list)    IntFactory
arithm(self, x, opt, y, mod, z)         IntConstraintFactory

```

Press **enter** to move, **tab** to select

Variables

pychoco supports four main types of variables: **boolvars** (taking values in {0, 1}), **intvars** (taking values in a set of integers, enumerated or bounded), **setvars** (taking values in a set interval), and **graphvars** (taking values in a graph interval, directed or undirected).

Declaring intvars

```

# Declaring single intvars
>>> v0 = m.intvar(42, name="v0") # constant
>>> v1 = m.intvar(1, 3) # values in {1,2,3}
>>> v2 = m.intvar([1,3,4,5]) # values in {1,3,4,5}
# Declaring an array of 5 intvars in [-2,2]
>>> vs = m.intvars(5, -2, 2)
# Declaring a 5x6 matrix of intvars in [-1,1]
>>> ws = [m.intvars(6, -1, 1) for i in range(5)]

```

Declaring boolvars

```

>>> b = m.boolvar(name="b")
>>> t = m.boolvar(True) # boolvar fixed to True

```

Declaring setvars

```

# Constant setvar equal to {2,3,12}
>>> s1 = m.setvar([2,3,12], name="s1")
>>> s2 = m.setvar([2,3,12]) # using a Python set
# setvar representing a subset of {1,2,3,5,12}
# -> possible values: {}, {2}, {1,3,5}, ...
>>> y = m.setvar(1, {1,2,3,5,12}, name="y")
# superset of {2,3} and subset of {1,2,3,5,12}
# -> possible values: {2,3}, {2,3,5}, {1,2,3}, ...
>>> z = m.setvar([2,3], {1,2,3,5,12})

```

Declaring graphvars

```

### Directed graphs ###
from pychoco.objects.graphs.directed_graph import *
# lower and upper bound of a directed graphvar
>>> n = 3 # maximum number of nodes
>>> lb = create_directed_graph(m, n, [], [])
>>> ub = create_directed_graph(m, n,
    [[0,1],[1,2],[2,0]] # edges
)
# declare the directed graphvar
>>> g = m.digraphvar(lb, ub, "g")

### Undirected graphs ###
from pychoco.objects.graphs.undirected_graph import *
# undirected graphvar with complete graph as ub
>>> lb = create_undirected_graph(m, n, [], [])
>>> ub = m.create_complete_undirected_graph(m, 3)
>>> g = m.graphvar(lb, ub, "g")

```

Constraints

Constraints are logic formulas defining allowed combinations of values for a set of variables, i.e. restrictions that must be respected by feasible solutions. Constraints can be declared in extension, by specifying the valid/invalid tuples, or in intention, by defining a relation between the variables. Constraints can be unary (involving one variable), binary, ternary, ..., or global (unfixed number of variables).

Posting constraints

```

>>> m.all_different(vars).post()
>>> m.arithm(v1, "+", v2, "<=", v0).post()
>>> m.table([v1,v2], [[1,3],[2,5],[3,1]]).post()

```

Reifying constraints

```

>>> x = m.intvar(0, 5)
>>> y = m.intvar(0, 5)
>>> cs = m.arithm(x, "<=", y) # cs is NOT posted
>>> b = constraint.reify()
# b = True only if x < y, otherwise b = False

```

Solving and retrieving solutions

Once your model is ready, you can launch the solver through the **Solver** object associated to the **Model** object. The **Solver** object also offers methods to configure the search.

Finding one solution

```

>>> solver = m.get_solver()
>>> solver.show_statistics()
>>> solver.set_dom_over_w_deg_search([v0,v1,v2])
>>> solution = solver.find_solution()
>>> opt = solver.find_optimal_solution(
    objective=v0,
    maximize=True)
>>> v0val = solution.get_int_val(v0)

```

Enumerating several solutions

```

>>> solver = m.get_solver()
>>> solver.show_statistics()
>>> solution = solver.find_all_solutions(
    solution_limit=10,
    time_limit="10s")
>>> optimals = solver.find_all_optimal_solutions(
    v0, True)

```

Figure 1: The [pychoco Cheat Sheet](#) provides a concise reference for discovering the API.

A classical example: the Sudoku solver

We illustrate the use of `pychoco` with one of the most emblematic CP examples: the Sudoku. In the following code, we instantiate a Sudoku instance.

```
sudoku = [
    [4, 0, 2, 0, 0, 0, 9, 7, 3],
    [0, 0, 0, 7, 4, 0, 6, 0, 8],
    [8, 0, 0, 0, 0, 9, 5, 1, 4],
    [7, 0, 0, 0, 0, 8, 0, 0, 0],
    [5, 9, 3, 0, 0, 6, 1, 8, 0],
    [0, 2, 8, 0, 0, 0, 0, 5, 9],
    [3, 1, 0, 0, 0, 2, 8, 9, 5],
    [0, 0, 0, 0, 8, 0, 0, 4, 1],
    [9, 0, 7, 1, 0, 4, 2, 0, 6]
]
```

As a reminder, the goal of Sudoku is to fill every empty cell (in our case, zeros) with a number between 1 and 9 such that each row, each column, and each of the nine 3x3 subgrids contains the numbers from 1 to 9 without repetition. It is easy to model our Sudoku solver with `pychoco`.

```
from pychoco import * # Import pychoco

model = Model("Sudoku Solver") # Instantiate a model object

# Instantiate the variables of the model
intvars = []
for row in range(0, 9):
    var_row = []
    for col in range(0, 9):
        value = sudoku[row][col]
        # integer variable with a lower/bound or a fixed value
        is_fixed = value != 0
        intvar = model.intvar(value) if is_fixed else model.intvar(1, 9)
        var_row.append(intvar)
    intvars.append(var_row)

# For each row, post an all_different constraint
for row in range(0, 9):
    var_row = intvars[row]
    model.all_different(var_row).post()

# For each column, post an all_different constraint
for col in range(0, 9):
    var_column = [row[col] for row in intvars]
    model.all_different(var_column).post()

# For each 3x3 subgrid, post an all_different constraint
for i in range(0, 3):
    line = intvars[i * 3 : i * 3 + 3]
    for j in range(0, 3):
        var_subgrid = sum([l[j * 3 : j * 3 + 3] for l in line], [])
        model.all_different(var_subgrid).post()
```

We can now solve our Sudoku by calling the solver, and then we can display the solution.

```
solver = model.get_solver()
solution = solver.find_solution() # Call the solver to retrieve a solution
for row in range(0, 9):
    line = [solution.get_int_val(v) for v in intvars[row]]
    print(line)

# Output:
# [4, 5, 2, 8, 6, 1, 9, 7, 3]
# [1, 3, 9, 7, 4, 5, 6, 2, 8]
# [8, 7, 6, 2, 3, 9, 5, 1, 4]
# [7, 4, 1, 5, 9, 8, 3, 6, 2]
# [5, 9, 3, 4, 2, 6, 1, 8, 7]
# [6, 2, 8, 3, 1, 7, 4, 5, 9]
# [3, 1, 4, 6, 7, 2, 8, 9, 5]
# [2, 6, 5, 9, 8, 3, 7, 4, 1]
# [9, 8, 7, 1, 5, 4, 2, 3, 6]
```

Note that, although this problem is a constraint satisfaction problem with exactly one solution (by definition of the classic Sudoku), psychoco supports solving constrained optimisation problems, and enumerating multiple solutions. Several search strategies are also available, as well as parallel portfolio search, as defined in Choco-solver. Other usage examples are available as a Jupyter notebook in [psychoco's GitHub repository](#).

Current usages and perspectives

Since its first release in October 2022, psychoco has been downloaded more than 97k times from PyPI. It is available as a backend solver in the [CPMpy](#) high-level modelling library. We also witness academic uses of psychoco that seem to be made possible or facilitated by the Python ecosystem. For example, the availability of psychoco in CPMpy seems to facilitate comparative analyses between different solvers accessible from Python ([Bleukx et al., 2024](#)). The richness of Python's ecosystem also fosters the integration of CP in workflows involving several AI techniques ([Hotz et al., 2024](#)) and the development of new tools based on CP (e.g., [pyagroplan](#)). Finally, as Python is increasingly used in teaching and training, it seems natural to teach CP using Python, especially for non-computer-scientist audiences (e.g., [AI for ecologists' training course](#)).

Acknowledgement

We acknowledge the developers of Python-JGraphT, whose work inspired the development of psychoco, and all contributors to psychoco.

References

- Audemard, G., Boussemart, F., Lecoutre, C., Piette, C., & Roussel, O. (2020). XCSP3 and its ecosystem. *Constraints*, 25(1), 47–69. <https://doi.org/10.1007/s10601-019-09307-9>
- Barahona, P., Krippahl, L., & Perriquet, O. (2011). Bioinformatics: A challenge to constraint programming. In *Hybrid optimization* (pp. 463–487). Springer. https://doi.org/10.1007/978-1-4419-1644-0_14
- Bleukx, I., Verhaeghe, H., Tsouros, D., & Guns, T. (2024). Efficient modeling of half-reified global constraints. *The 23rd Workshop on Constraint Modelling and Reformulation, Date: 2024/09/02-2024/09/02, Location: Gerona, Spain*.

- Challand, M., Vismara, P., & de Tourdonnet, S. (2025). Combining constraint programming and a participatory approach to design agroecological cropping systems. *Agricultural Systems*, 222, 104154. <https://doi.org/10.1016/j.agsy.2024.104154>
- Deléglise, H., Justeau-Allaire, D., Mulligan, M., Espinoza, J.-C., Isasi-Catalá, E., Alvarez, C., Condom, T., & Palomo, I. (2024). Integrating multi-objective optimization and ecological connectivity to strengthen Peru's protected area system towards the 30*2030 target. *Biological Conservation*, 299, 110799. <https://doi.org/10.1016/j.biocon.2024.110799>
- Gerault, D., Minier, M., & Solnon, C. (2016). Constraint programming models for chosen key differential cryptanalysis. *International Conference on Principles and Practice of Constraint Programming*, 584–601. https://doi.org/10.1007/978-3-319-44953-1_37
- Guns, T. (2019). Increasing modeling language convenience with a universal n-dimensional array, Cppy as Python-embedded example. *Proceedings of the 18th Workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, 19.
- Hooker, J. N. (2016). Finding alternative musical scales. *International Conference on Principles and Practice of Constraint Programming*, 753–768. https://doi.org/10.1007/978-3-319-44953-1_47
- Hotz, L., Bähnisch, C., Lubos, S., Felfernig, A., Haag, A., & Twiefel, J. (2024). *Exploiting large language models for the automated generation of constraint satisfaction problems*. 3812, 91–100.
- Lecoutre, C., & Szczepanski, N. (2024). *PyCSP3: Modeling combinatorial constrained problems in Python* (No. arXiv:2009.00326). arXiv. <https://doi.org/10.48550/arXiv.2009.00326>
- Michail, D., Kinable, J., Naveh, B., & Sichi, J. V. (2020). JGraphT—a Java library for graph data structures and algorithms. *ACM Transactions on Mathematical Software (TOMS)*, 46(2), 1–29. <https://doi.org/10.1145/3381449>
- Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., & Tack, G. (2007). MiniZinc: Towards a standard CP modelling language. In C. Bessière (Ed.), *Principles and Practice of Constraint Programming – CP 2007* (Vol. 4741, pp. 529–543). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-74970-7_38
- Prud'homme, C., & Fages, J.-G. (2022). Choco-solver: A Java library for constraint programming. *Journal of Open Source Software*, 7(78), 4708. <https://doi.org/10.21105/joss.04708>
- Rossi, F., Beek, P. van, & Walsh, T. (Eds.). (2006). *Handbook of constraint programming* (Vol. 2). Elsevier. ISBN: 978-0-444-52726-4
- Vismara, P., Coletta, R., & Trombettoni, G. (2016). Constrained global optimization for wine blending. *Constraints*, 21(4), 597–615. <https://doi.org/10.1007/s10601-015-9235-5>
- Wallace, M. (1996). Practical applications of constraint programming. *Constraints*, 1(1-2), 139–168. <https://doi.org/10.1007/BF00143881>