Recreating Goto in Python: How and Why

Carl Cerecke

https://doi.org/10.34074/proc.240119
Correspondence: Carl.Cerecke@nmit.ac.nz

This publication may be cited as:

Cerecke, C. (2024). Recreating Goto in Python: How and Why. In H. Sharifzadeh (Ed.), *Proceedings: CITRENZ 2023 Conference, Auckland, 27–29 September* (pp. 150–159). ePress, Unitec. https://doi.org/10.34074/proc.240119

Contact:
epress@unitec.ac.nz
www.unitec.ac.nz/epress/
Unitec
Private Bag 92025, Victoria Street West
Tāmaki Makaurau Auckland 1142
Aotearoa New Zealand

ISBN: 978-1-99-118344-6

ePress
Research with impact
Unitec

Unitec is a business division of Te Pūkenga – New Zealand Institute of Skills and Technology

ABSTRACT

The Python language has no unrestricted goto statement, but because of Python's deep introspection capabilities, along with its dynamic nature, an unrestricted goto can be added to the language by rewriting a function's bytecode at runtime. Two primary use cases for goto in Python are discussed: porting existing historic code to Python while maintaining the existing code's flavour, and implementing state machines. While theoretically unnecessary, goto can make the first case easier, and the second case faster. An implementation is described and performance tests on an example state machine show using goto is a very fast method of implementing state machines in Python, and existing efforts to port historic code are already using the goto implementation.

KEYWORDS

Python, goto

INTRODUCTION

Dijkstra's well-known 1968 letter "Go To Considered Harmful" (Dijkstra, 1968, p. 147) popularised the problems that can be caused by careless use of the unrestricted jump that goto provides. The core of Dijkstra's complaint is that "unbridled use of the go to statement" makes reasoning about the behaviour of a running program difficult when presented with the static program code. In the programmer's struggle to constrain complexity, the goto statement provides far more opportunities for increasing complexity than it does to reduce it. Knuth (1974) brought some balance to Dijkstra's opinion and found that even with the significant then-recent advance of structured programming in academia, there were both cases where goto should be eliminated, and cases where goto was justified or even preferred. It was about two decades later that mainstream programming languages emerged (such as Java, Python, and then later JavaScript) with no native goto statement available. These languages instead have two restricted forms of the goto statement: *break* and *continue*. These restricted jump instructions cover many of the use cases where goto was used traditionally, but without the same potential for spaghettification of code.

While *break* and *continue* cover the majority of 'sensible' use cases for the goto statement, there are two main areas where goto still proves to be a useful construct for control flow: (1) porting existing software that uses goto statements; and (2) complex state machines.

Ceccato et al. (2008) compare different automatic goto-elimination techniques in porting a large (8 million lines of code) banking system to Java from a "BASIC-like language" which (initially, at least) had very limited control flow statements. Out of the 8 million lines, around 500,000 were goto statements. While most goto statements were automatically eliminated in the generated Java code, a minority could not be translated without a significant decrease in the understandability of the generated code. In these cases, placeholder **`jlabel()`** and **`jgoto()`** calls were introduced to the code, and a post-compilation step replaced these the **`jgoto()`** calls in the java byte code with a JVM goto instruction to the appropriate label. This post-compilation step is essentially a 'hack' to add goto to Java; the underlying bytecode interpreter has a goto, and the post-compilation step rewrites the bytecode to allow direct access to it. It is similar to the approach used in this paper, though with Java a separate post-compilation step is required, and in Python this can be done at runtime.

At the other end of the program size scale there is historical interest in early programs in languages that required the use of goto for control constructs, such as BASIC. There were many books and magazines with BASIC code listings in the 1970s and 1980s. These programs tended to be quite small, as not only did they have to be typed in from scratch, but the programs were intended for computers with very limited RAM: the Apple II, Commodore PET, and Tandy TRS-80 Model 1 were all available in 1977 with 4Kb of RAM (Reimer, 2005). A typical example is the program Hammurabi (Willaert, 2019), an economics simulator. It was already nearly a decade old when it was included in the early compendium *101 BASIC computer games* (Ahl, 1973). The code, to our modern eyes, looks very primitive and is littered with goto statements (Figure 1).

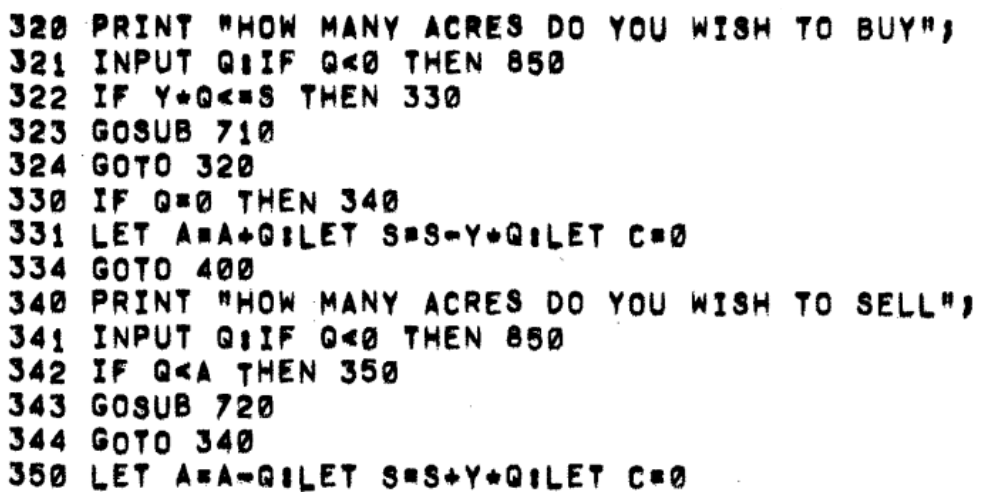

Figure 1. A scanned extract from the book *101 BASIC computer games*. Fourteen lines with 8 gotos!

In order to faithfully reproduce the code in a modern language while also retaining the flavour of the original code (Massey, 2014), some form of goto is required. As an example, here are the first 6 lines from the original code in Figure 1 translated to Python in Massey's port:

```
label .line320
Q=get_number("HOW MANY ACRES DO YOU WISH TO BUY")
if Q<0:
    goto .line850
if Y*Q<=S:
    goto .line330
sub710()
goto .line320
label .line330
if Q==0:
    goto .line340
```

A second common use of goto is where the control constructs of the programming language are not powerful or convenient enough to express the control flow that the programmer has in mind. One such example, "Centralized exiting of functions", is explicitly condoned in the Linux kernel coding style guide (The Linux kernel documentation, n.d.). In Python a centralised exit of a function might be idiomatically constructed using a try-except-else-finally block.

Another example where a goto is convenient is when coding a state machine. For example, a typical state machine implementing a shift-reduce parser for a modern programming language may have many hundreds of states in the state machine. Non-goto approaches to large state machines might use tables (e.g., YACC, Bison) or a loop with a switch statement. In either case, a core loop is required to iterate for each state transition.

Figure 2 shows an example of a table-style finite state machine for an LR parser with 12 states that implements a simple mathematical expression parser, including the four basic operators and parentheses. While typically such state machines are represented as tables, an early attempt at directly representing a parser's state machine using goto statements (Pennello, 1986) resulted in speedups of 6–10 times over a table-driven approach, though this approach directly generated assembly code. A later approach generated the parser state machine in c code and strived for compatibility with the popular early parser generator YACC, and yielded speedups of 2–6 times (Bhamidipaty & Proebsting, 1998).

Even with modern structured programming control flow constructs, the goto statement still has some potential niche applications where it is the best tool for the job.

LR(1) Parsing Table

	id	+	×	(	)	$	E	T	F
0	shift 5			shift 4			1	2	3
1		shift 6				accept			
2		$E \longrightarrow T$	shift 7		$E \longrightarrow T$	$E \longrightarrow T$			
3		$T \longrightarrow F$	$T \longrightarrow F$		$T \longrightarrow F$	$T \longrightarrow F$			
4	shift 5			shift 4			8	2	3
5		$F \longrightarrow id$	$F \longrightarrow id$		$F \longrightarrow id$	$F \longrightarrow id$			
6	shift 5			shift 4				9	3
7	shift 5			shift 4					10
8		shift 6			shift 11				
9		$E \longrightarrow E + T$	shift 7		$E \longrightarrow E + T$	$E \longrightarrow E + T$			
10		$T \longrightarrow T \times F$	$T \longrightarrow T \times F$		$T \longrightarrow T \times F$	$T \longrightarrow T \times F$			
11		$F \longrightarrow (E)$	$F \longrightarrow (E)$		$F \longrightarrow (E)$	$F \longrightarrow (E)$			

Figure 2. Table representation of a state machine for parsing simple mathematical expressions.

ADDING A GOTO STATEMENT TO PYTHON

The first attempt to add goto to Python (Hindle, n.d.), a 2004 April Fool's joke, used the **`sys.settrace`** function of Python, which registers a function that is then called before executing every line of code. The **`sys.settrace`** function was intended for implementing a debugger and greatly slows a running program, but it is powerful. Hindle's version also included the humorous **`comefrom`** statement (the nefarious opposite of the goto statement).

A more efficient way to add goto to Python is to take advantage of the dynamic nature of the Python runtime environment to rewrite the Python bytecode for a function. This is the approach used by the code implemented by the author (Cerecke, n.d.) and discussed in the remainder of this section. The source code is available at https://github.com/cdjc/goto. Only the reference implementation of Python (https://python.org) is compatible with the

goto implementation discussed in this paper, as other implementations (such as Jython, IronPython and PyPy) take a different runtime approach.

Though we often think of Python as interpreted, rather than compiled like other languages that use a bytecode interpreter (or "VM") such as Java (the JVM) or C# (the .NET CLR), this is technically incorrect. In Python the compilation step is hidden – there is no separate compile-then-run steps in Python, like there are in Java or C#.

These are the high-level requirements before goto statements can be added to Python using bytecode rewriting:

1. Access a function's bytecode at runtime.
2. Set a function's bytecode at runtime.
3. Mark a function as requiring a rewrite of its bytecode.
4. Specify labels and gotos within the bounds of current Python syntax.

Requirement 1 is met in Python, as each function has a corresponding code object (accessed via the function's **`__code__`** attribute), and each code object has an attribute to get the bytecode string for the code object. The helpful dis module in the standard library provides a function that can produce a disassembly of another function.

Requirement 2 is met by the **`CodeType`** constructor in the **`types`** standard library module, through which a new code object can be created. A recent addition to Python (version 3.11) makes this more convenient, with the **`replace`** function of a code object. Furthermore, this new code object can then be assigned to a function's code object at runtime.

Requirement 3 is met by using a Python function decorator – a convenient language feature for creating and using higher-order functions.

A convenient way requirement 4 can be met is by misusing object attribute access. For example, the line foo.bar in Python means 'access the attribute bar from the object foo'. Although this is an expression, expressions are also valid Python statements, and replacing the object with the word 'goto' (or 'label') will allow specification of goto (and label) statements using Python syntax. This does not result in an error during compilation, as Python's dynamic typing does not attempt to resolve variables during the compilation step. Without rewriting the function's bytecode, running the function would result in the exception: **`NameError: name 'goto' is not defined`**

Figure 3 shows a trivial example of all four requirements in the Python interpreter. Line 3 imports the **`dis`** module for viewing the disassembly of a function. Line 4 creates a simple function named fn, where the first line (line 5 of the listing) is specifying a label as if it was a Python attribute (requirement 4). The statement on line 8 prints a disassembly of the function. The **`label.start`** from the function results in two instructions in the bytecode, one for the global object **`label`** (lines 11–12) and one for **`start`** attribute access of that object. If the function were to be called at this point, Python would raise an exception because it cannot find any globals with the name **`label`**.

Line 17 extracts the bytecode of the function as an immutable Python bytes object (requirement 1); not all of the bytecode is visible in the Figure due to its truncation at the edge of the Figure. The many zeros in the bytecode (represented by **`\x00`**) are a new Python 3.11 feature that adds cache space within the bytecode to be used during runtime optimisations. In order to change the bytecodes, a list is created from the bytes, and the bytes representing the **`label .start`** line of the function are deleted (line 20).

Line 23 creates a new code object from the function's existing code object, with the bytecodes replaced, and assigns the new code object to the function code object (requirement 2). The **`replace`** function is a new addition to Python 3.11; prior versions required creating a new code object from scratch.

Finally, the function is called (line 24) and the function's new bytecode, now containing only code for the return 1 statement, is executed and the resulting value (1) is returned.

```
1. Python 3.11.2 (tags/v3.11.2:878ead1, Feb  7 2023, 16:38:35) [MSC v.193
2. Type "help", "copyright", "credits" or "license" for more information.
3. >>> import dis
4. >>> def fn():
5. ...     label .start
6. ...     return 1
7. ...
8. >>> dis.dis(fn)
9.   1           0 RESUME                   0
10.
11.  2           2 LOAD_GLOBAL              0 (label)
12.             14 LOAD_ATTR                1 (start)
13.             24 POP_TOP
14.
15.  3          26 LOAD_CONST               1 (1)
16.             28 RETURN_VALUE
17. >>> fn.__code__.co_code
18. b'\x97\x00t\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00j\x01\x00\x00\x
19. >>> bytecode = list(fn.__code__.co_code)
20. >>> del bytecode[2:-4]
21. >>> bytes(bytecode)
22. b'\x97\x00d\x01S\x00'
23. >>> fn.__code__ = fn.__code__.replace(co_code = bytes(bytecode))
24. >>> fn()
25. 1
```

Figure 3. Trivial example showing the function bytecode rewriting mechanism required to implement goto in Python.

While Figure 3 illustrates the main mechanism by which a goto statement can be added to Python, there are still a few tasks remaining to create a working function decorator. At a high level these are:

1. Identify all the labels and gotos in the function. Check whether any gotos are missing labels, and labels are unique, and no illegal jumps are attempted.
2. Overwrite the instructions in the bytecode for each label and goto (and the following CACHE instructions) to NOP instructions.
3. For each position in the bytecode where the goto statement was, insert a JUMP_FORWARD or a JUMP_BACKWARD to the corresponding label. Take care to stop any in-progress iterators if we are jumping out of an iterator loop.

There are a few cautions that must be exercised:
No jumping into any part of function that is inside a block that has some special initialiser or meaning. For example, **`for`** blocks, **`with`** blocks, and **`try`** blocks. It's not clear what the semantics should be in those cases. Jumping into a **`while`** block is perfectly fine, though. Jumping out of a **`try`** block is also forbidden. There should be no way to bypass the **`finally`** block.

A new feature of the recent Python 3.11 is the removal of unreachable code during bytecode compilation. This means that code that is only reachable by goto may be in danger of being optimised away. This makes the following code fail:

```
for ls in matrix:
    for num in ls:
        if num == 1234:
            goto .foundIt

# didn't find it
return False

label .foundIt  # "unreachable" code removed by Python compiler
return True
```

To work around the unreachable code elimination, the `return False` would need to have an always-true 'guard', such as `if __name__: return False` The Python special variable `__name__` is normally always set to some value, either the name of the current module, or `"__main__"` (*The Python language reference*, n.d.) so the `if`-statement will always be true, barring any explicit meddling with the module's `__name__` attribute.

There is a maximum depth of iterators (`for` loops) that can be accommodated in the bytecode before lengthening the bytecode is required – lengthening the bytecode is tricky, as any existing jumps in the bytecode may also have their destinations recalculated. Each iterator that must be stopped before jumping must be removed from the top of the stack with a POP_TOP bytecode. By reusing the CACHE instructions where the goto statements appear in the bytecode, up to 12 instructions can be overwritten before the bytecode list itself has to be extended. This allows up to 11 POP_TOP instructions before the jump instruction. The current implementation limits the nested iterators to 10, which seems more than adequate for a reasonable function. The remaining 2 bytes allow for an extra instruction in case the jump offset is greater than what can be represented in one byte, and an EXTENDED_ARG opcode is required.

Each instruction in Python bytecode uses 2 bytes: one for the opcode, and one for the argument. For an opcode requiring a value larger than one byte, up to three preceding EXTENDED_ARG opcodes are used to add as many higher-order bytes as required. If a jump over 255 instructions is required, then an extra EXTENDED_ARG instruction will need to be inserted. This must be done with care, as inserting an EXTENDED_ARG instruction also changes the length of the jump by +1 (for a backward jump) or -1 (for a forward jump). The code assumes that jumps will never be more than what would fit in 2 bytes (65536 would be a very long jump!) and so will only ever need a single EXTENDED_ARG bytecode instruction.

A full example, showing the import, the decorator, a goto, a label, and two calls of the goto-decorated function, is shown in Figure 4.

```
>>> from goto import goto
>>> @goto
... def matrix_find(matrix, n):
...     for square in matrix:
...         for line in square:
...             for value in line:
...                 if value == n:
...                     rval = "found"
...                     goto.found
...     rval = "not found"
...     label.found
...     # ... other code here ...
...     return rval
...
>>> matrix_find([[[1,2]],[[3,4]]], 3)
'found'
>>> matrix_find([[[1,2]],[[3,4]]], 5)
'not found'
```

Figure 4. An example of using the goto decorator in Python.

RESULTS

Implementing state machines is one of the main uses of a goto statement. To gain an approximate measure of the speed of using gotos for state machines compared to other methods, an example state machine was implemented using four methods:

1. The popular python-statemachine library (Macedo, n.d.).
2. Python **`for`**-loop around the Python **`match`** statement. Each case in the **`match`** statement matches an input character, and the next state is set based on the current state using **`if`** statements.
3. Using the equivalent regular expression in Python. The regular expression matching (and building) code is implemented in C inside the Python interpreter. The regular expression corresponding to the state machine was compiled in Python using the following regular expression string:
   `r'\d+(\.\d+)?(e[+-]\d+)?(\+\d+(\.\d+)?(e[+-]\d+)?)*\$'`
4. Using the goto library described in this paper and using labels for states, and goto for transitions.

The state machine used for the test (Figure 5) recognises valid expressions consisting of a sum of floating point numbers. Each character in the input string triggers a transition to another state if that character is part of a valid string. The state machine is non-trivial enough for a reasonable test, yet small enough to be easily and quickly implemented (and debugged!) using a variety of methods. The implementation details are in the **`goto_test_speed.py`** file of the associated GitHub project (https://github.com/cdjc/goto).

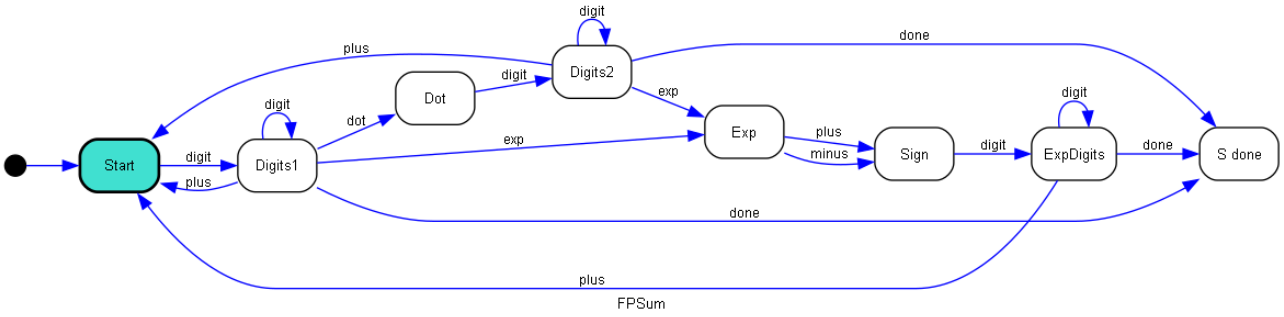

Figure 5. Finite state machine for recognising an expression of a sum of floating point numbers.

For the speed comparison between the methods, a 30,000 character randomly generated string exercising all non-final transitions of the finite state machine was generated. The Python standard library timeit module was used to time each method. Performance was measured on an i7-7820HQ laptop with 32GB RAM running Windows 11.

Table 1. Speed of different finite state machine implementations.

	Time to process 30,000 characters	Slowdown compared to goto method
Python-statemachine	504ms	180×
For-loop with match	21ms	7.5×
Regular expression	3.1ms	1.1×
Github.com/cdjc/goto	2.8ms	1×

Table 1 shows that the goto implementation presented in this paper is far superior in speed to the two other 'pure' Python implementations for programming state machines. The regular expression speed was comparable, which is perhaps surprisingly slow considering it does not execute any Python bytecode when matching the input (other than the regular expression **`match`** method call), but rather calls in to the regular expression module written in C inside core Python.

It can be argued that the goto implementation's much faster speed is somewhat offset by its lower readability (the regular expression suffers from this as well) and by being non-standard. Knuth's aphorism "premature optimization is the root of all evil" (Knuth, 1974, p. 268), in a paper about structured programming with gotos, is perhaps appropriate here.

CONCLUSION AND RECOMMENDATIONS

The dynamic nature of Python, along with its object introspection capabilities, provides a way to implement goto in Python at runtime. This technique could be used for other situations where directly manipulating bytecode is required. A major disadvantage is the instability of the Python bytecode specification; it is subject to significant changes between language versions with no effort to maintain backward compatibility. Also, the technique as implemented is only applicable to the standard reference implementation Python (CPython).

Wanting to use the goto statement in Python is a niche use case mostly restricted to porting some old code while retaining the flavour of the original code. State machines are another possible use case, and the goto statement provides a very fast alternative to other pure-Python state machine implementations. This speed does come at the cost of both reliability (because the underlying bytecode is subject to incompatible changes between versions) and readability, as the non-standard use of familiar Python constructs will be unfamiliar to many.

REFERENCES

Ahl, D. (1973). *101 BASIC computer games*. Ditigal Equipment Corporation. http://www.bitsavers.org/pdf/dec/_Books/101_BASIC_Computer_Games_Mar75.pdf

Bhamidipaty, A., & Proebsting, T. A. (1998). Very fast YACC-compatible parsers (for very little effort). *Software: Practice and Experience*, *28*(2), 181–190. https://onlinelibrary.wiley.com/doi/10.1002/(SICI)1097-024X(199802)28:2%3C181::AID-SPE139%3E3.0.CO;2-4

Ceccato, M., Tonella, P., & Matteotti, C. (2008). Goto elimination strategies in the migration of legacy code to Java. In *2008 12th European Conference on Software Maintenance and Reengineering* (pp. 53–62). IEEE. https://doi.org/10.1109/CSMR.2008.4493300

Cerecke, C. (n.d.). *Goto in Python* [Computer software]. Retrieved July 22, 2023, from https://github.com/cdjc/goto

Dijkstra, E. W. (1968). Letters to the editor: Go to statement considered harmful. *Communications of the ACM*, *11*(3), 147–148. https://doi.org/10.1145/362929.362947

Hindle, R. (n.d.). *Goto for Python*. Retrieved July 18, 2023, from http://entrian.com/goto/

Knuth, D. E. (1974). Structured programming with *go to* statements. *ACM Computing Surveys*, *6*(4), 261–301. https://doi.org/10.1145/356635.356640

Macedo, F. (n.d.). *Python StateMachine*. Retrieved July 22, 2023, from https://python-statemachine.readthedocs.io/en/latest/readme.html

Massey, B. (2014). *HAMURABI.BAS in Python 3* [Computer software]. https://github.com/BartMassey/hamurabi

Pennello, T. J. (1986). Very fast LR parsing. *ACM SIGPLAN Notices*, *21*(7), 145–151. https://doi.org/10.1145/13310.13326

Reimer, J. (2005, December 15). Total share: 30 years of personal computer market share figures. *Ars Technica*. https://arstechnica.com/features/2005/12/total-share/

The Linux kernel documentation. (n.d.). *Linux kernel coding style*. Retrieved July 17, 2023, from https://docs.kernel.org/process/coding-style.html

The Python language reference. (n.d.). The import system. Retrieved July 22, 2023, from https://docs.python.org/3/reference/import.html

Willaert, "Critical Kate." (2019, September 9). The Sumerian game: The most important video game you've never heard of. *A Critical Hit!* https://www.acriticalhit.com/sumerian-game-most-important-video-game-youve-never-heard/

AUTHOR

Dr Carl Cerecke is an IT Tutor at Nelson Marlborough Institute of Technology Te Whare Wānanga o te Tau Imu. He has a wide array of research interests, some of which are useful.
carl.cerecke@nmit.ac.nz