goto in Python 3. Yes. Really.

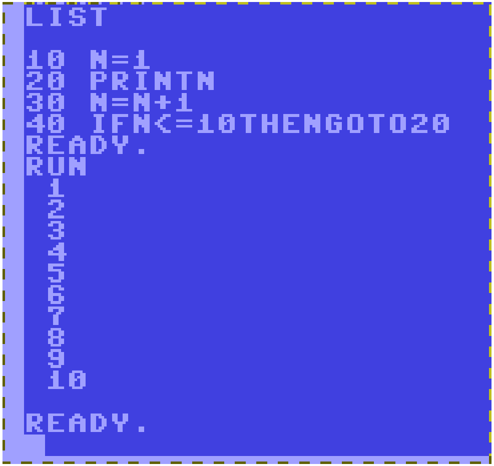
Kiwi PyCon 2014

Carl Cerecke
carl@free.org.nz

https://github.com/cdjc/goto

September 13-14, 2014

BASIC on the Commodore 64



```
LIST

10  N=1
20  PRINTN
30  N=N+1
40  IFN<=10THENGOTO20
READY.
RUN
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10

READY.
```

History

- In the beginning was the `goto`
- 1958 Heinz Zemanek expresses doubts about goto at pre-ALGOL meeting.
- 1968 Edsgar Dijkstra "GOTO Considered Harmful"
- 1974 Don Knuth "Structured Programming with go to statements"
- 1987 Frank Rubin ' "GOTO Considered Harmful" Considered Harmful'

Why add goto to Python?

It seemed like a good idea at the time...

Also useful for:

- State machines
- Breaking out of a nested loop
- Generating python code programmatically
- Translating goto-filled code to python

But it's already been done before!

- April 1 2004, http://entrian/goto
- Uses `sys.settrace`
- Checks before the execution of *every line* for goto. Slow
- Module scope, not function scope.

Goto using bytecode manipulation

- Python source code is compiled into python *bytecode* instructions.
- Each bytecode instruction is 1-3 bytes long.
- Python bytecodes already have gotos:
  - JUMP_FORWARD(delta)
  - JUMP_ABSOLUTE(target)
  - also exotics like JUMP_IF_FALSE_OR_POP(target)
- CPython only.
- See the `dis` module.

Simple example function

```
from goto import goto

@goto                # enables goto in decorated function
def simple(n):
    goto .skip
    print(n)
    label .skip
```

We can see python bytecodes:

```
import dis
dis.dis(fn)          # pretty print byte code
```

Disassembly of simple function (without goto decorator)

line	addr	opcode	par	interpretation
302	0	LOAD_GLOBAL	0	(goto)
	3	LOAD_ATTR	1	(skip)
	6	POP_TOP		
303	7	LOAD_GLOBAL	2	(print)
	10	LOAD_FAST	0	(n)
	13	CALL_FUNCTION	1	(1 positional, 0 keyword pair)
	16	POP_TOP		
304	17	LOAD_GLOBAL	3	(label)
	20	LOAD_ATTR	1	(skip)
	23	POP_TOP		
	24	LOAD_CONST	0	(None)
	27	RETURN_VALUE		

Changes required for goto

- Python treats `goto` statement as attribute access.
- Likewise for `label` statement.
- Need to change `goto` into `JUMP_ABSOLUTE`
- and `label` into `NOP`

Byte code with goto changes

line	addr	opcode	par	interpretation
302	0	JUMP_ABSOLUTE	24	
	3	~~LOAD_ATTR~~	1	(skip)
	6	~~POP_TOP~~		
303	7	LOAD_GLOBAL	2	(print)
	10	LOAD_FAST	0	(n)
	13	CALL_FUNCTION	1	(1 positional, 0 keyword pair
	16	POP_TOP		
304	17	NOP		
	18	NOP		
	19	NOP		
	20	NOP		
	21	NOP		
	22	NOP		
	23	NOP		
target	24	LOAD_CONST	0	(None)
	27	RETURN_VALUE		

How to change bytecodes?

Decorator outline (code at http://github.com/cdjc/goto)

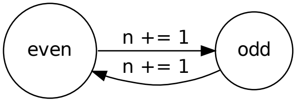
- `c = fn.__code__`   # code object. Not read only :-)
- `c.co_code`   # bytecode string. Read only :-(
- Find all labels and gotos in `c.co_code`
- `NOP` all labels.
- Make gotos into `JUMP_ABSOLUTE`
- Make new code object
- `fn.__code__` = new code object
- `return fn`

Problems!

```
@goto
def infinite(n):
    label .start
    for i in 'oops':
        goto .start
```

- At loop-start, python adds a 'block'.
- At loop-end python does `POP_BLOCK`
- Jumping out of a loop must `POP_BLOCK` before jump.
- Illegal:
  - Jump into a loop (Segmentation Fault on `POP_BLOCK`)
  - Jump into/out of `try`, `except`, `finally`, `with`
  - Multiple identical labels (or missing label)
  - Jump out of loop nested more than four deep.

Performance



- Function-based state machine within a class
- Goto-based state machine within a function
- `while` loop in plain code

The *even* state breaks at n = 100000000
Python 3.3.1 on Linux VM

Performance (function-based state machine)

```
class state_machine:
    def even_state(self):
        ...
        return self.odd_state
    def odd_state(self):
        ...
        return self.even_state
    def go():
        state = self.even_state
        while state:
            state = state()
```

35.0 seconds

Performance (plain while loop)

```
n = 0
while n != limit:
    n += 1          # even -> odd
    n += 1          # odd -> even
```

11.5 seconds

Performance (goto-based state machine)

```
@goto
def goto_state_machine(limit):
    n = 0

    label .state_even  ### even_state
    if n == limit:
        return
    n += 1
    goto .state_odd
    ################
    label .state_odd   ### odd_state
    n += 1
    goto .state_even
```

Performance (goto-based state machine)

```
@goto
def goto_state_machine(limit):
    n = 0

    label .state_even  ### even_state
    if n == limit:
        return
    n += 1
    goto .state_odd
    ################
    label .state_odd   ### odd_state
    n += 1
    goto .state_even
```

7.2 seconds! (over 4 seconds *faster* than a `while` loop!)

Performance (goto-based state machine)

```
@goto
def goto_state_machine(limit):
    n = 0

    label .state_even  ### even_state
    if n == limit:
        return
    n += 1
    goto .state_odd
    ################
    label .state_odd   ### odd_state
    n += 1
    goto .state_even
```

7.2 seconds! (over 4 seconds *faster* than a `while` loop!)
But... `while` loop *inside* function: 7.1 seconds. :-(

The End

Questions?