developerWorks®

Charming Python: Using state machines

Algorithms and programming approaches in Python

David Mertz
President
Gnosis Software, Inc.

01 August 2000

State machines, in a theoretical sense, underlie almost everything related to computers and programming. And it also turns out that state machines, in a practical sense, can help solve many ordinary problems (especially for Python programmers). In this article, David Mertz discusses some practical examples of when and how to code a state machine in Python.

What is Python?

Python is a free, high-level, interpreted language developed by Guido van Rossum. It combines a clear syntax with powerful (but optional) object-oriented semantics. Python is widely available and highly portable.

What is a state machine?

An overly accurate description of a state machine is that it is a directed graph, consisting of a set of nodes and a corresponding set of transition functions. The machine "runs" by responding to a series of events. Each event is in the domain of the transition function belonging to the "current" node, where the function's range is a subset of the nodes. The function returns the "next" (perhaps the same) node. At least one of these nodes must be an end-state. When an end-state is reached, the machine stops.

But an abstract mathematical description (like the one I just gave) does not really illustrate what use a state machine might serve in practical programming problems. A different tack is to define a state machine as any imperative programming language in which the nodes are also the source lines. This definition, though accurate, is equally as pedantic and useless as the first, from a practical point of view. (This condition is not necessarily true for a declarative, functional, or constraint-based languages such as Haskell, Scheme, or Prolog.)

Let's try using an example more appropriate to the actual task at hand. Every regular expression is logically equivalent to a state machine, and the parser of every regular expression implements a state machine. In fact, most programmers often write state machines without really thinking about it.

Following this example, let's look at a practical heuristic definition of a state machine. We often have certain distinct ways of responding to a finite set of events. In some cases, the response depends only on the event itself. But in other cases, the appropriate action depends on prior events.

The state machines discussed in this article are high-level machines intended to demonstrate a programming solution to a class of problems. If it makes sense to talk about your programming problem in terms of categories of behavior in response to events, your solution is likely to be an explicit state machine.

A text-processing state machine

One of the programming problems most likely to call for an explicit state machine involves processing text files. Processing a text file often consists of sequentially reading a unit of information (typically either a character or a line) and doing something in response to the unit you've just read. In some cases, this processing is "stateless" (that is, each such unit contains enough information to determine exactly what to do). In other cases, even though the text file is not completely stateless, the data has a limited context (for example, the action might not depend on much more than the line number). However, in other common text-processing problems, the input files are highly "stateful." The meaning of each piece of data depends on what preceded it (and maybe even on what follows). Reports, mainframe data-feeds, human-readable texts, programming source files, and other sorts of text files are stateful. A simple example is a line that might occur in a Python source file:

```
myObject = SomeClass(this, that, other)
```

That line means something different if it happens to be surrounded by these lines:

```
"""How to use SomeClass:
myObject = SomeClass(this, that, other)
"""
```

We need to know that we are in a "blockquote" **state** to determine that the line is part of a comment rather than a Python action.

When not to use a state machine

When you begin the task of writing a processor for any stateful text file, ask yourself what types of input items you expect to find in the file. Each type of input item is a candidate for a state. These types should be several in number. If the number is huge or indefinite, a state machine is probably not the right approach. (In such cases, some sort of database solution might be more appropriate.)

Also consider whether you even need a state machine. In many cases it's better to start with a simpler approach. It might turn out that even though your text file is stateful, there is an easy way to read it in chunks (where each chunk is a single type of input value). A state machine is really only worth implementing if the transitions between types of text require some calculation based on the content within a single state-block.

The following simple example is a case when you need a state machine. Think of two rules for dividing a list of numbers into blocks. Under the first rule, a zero in the list indicates a break between blocks. Under the second rule, a break between blocks occurs when the sum of the elements in a block exceeds 100. Since it takes an accumulator variable to determine when the threshold has been reached, you can't see the sub-list boundaries "at a glance." Hence, the second rule may be a much better candidate for something resembling a state machine.

An example of a somewhat stateful text file that is nonetheless probably *not* best handled with a state machine is a Windows-style .ini file. This file consists of section headers, comments, and a number of value assignments. For example:

```
; set the colorscheme and userlevel
[colorscheme]
background=red
foreground=blue
title=green

[userlevel]
login=2
title=1
```

Our example has no real-life meaning, but it indicates some interesting features of the .ini format.

- In one sense, the type of each line is determined by its first character (either semicolon, left brace, or alphabetic).
- In another sense, the format is "stateful" insofar as the keyword "title" presumably means something independent when it occurs in each section.

You could program a text processor that had a COLORSCHEME state and a USERLEVEL state, and still processed the value assignments of each state. But that does not seem like the *right* way to handle this problem. For example, you could simply create the natural chunks in this text file with some Python code like:

Chunking Python code to process .INI file

```
import string
txt = open('hypothetical.ini').read()
sects = string.split(txt, '[')
for sect in sects:
   # do something with sect, like get its name
   # (the stuff up to ']') and read its assignments
```

Or, if you wished, you could use a single `current_section` variable to keep place:

Counting Python code to process .INI file

```
for line in open('hypothetical.ini').readlines():
     if line[0] == '[':
          current_section = line(1:-2)
     elif line[0] == ';' :
          pass     # ignore comments
     else:
          apply_value(current_section, line)
```

When to use a state machine

Now that we've decided not to use a state machine if the text file is "too simple," let's look at a case where a state machine *is* worthwhile.

"Smart ASCII" is a text format that uses a few spacing conventions to distinguish types of text blocks such as headers, regular text, quotations, and code samples. While it is easy for a human reader or writer to visually parse the transitions between these text block types, there is no simple way for a computer to split a Smart ASCII file into its constituent text blocks. Unlike the .ini file example, text block types can occur in any order. There is no single delimiter that separates blocks in all cases (a blank line *usually* separates blocks, but a blank line within a code sample does not necessarily end the code sample, and blocks need not be separated by blank lines). Since you do need to reformat each text block type differently to produce the correct HTML output, a state machine seems like a natural solution.

The general behavior of the Txt2Html reader is as follows:

1. Start in an initial state.
2. Read a line of input.
3. Depending on the input and the current state, either transition to a new state or process the line as appropriate for the current state.

This example is about the simplest case you would encounter, but it illustrates the following pattern that we have described:

A simple state machine input loop in Python

```
global state, blocks, bl_num, newblock

#-- Initialize the globals
state = "HEADER"
blocks = [""]
bl_num = 0
newblock = 1

for line in fhin.readlines():
     if state == "HEADER":       # blank line means new block of header
          if blankln.match(line): newblock = 1
          elif textln.match(line): startText(line)
          elif codeln.match(line): startCode(line)
          else:
               if newblock: startHead(line)
               else: blocks[bl_num] = blocks[bl_num] + line
     elif state == "TEXT":       # blank line means new block of text
          if blankln.match(line): newblock = 1
          elif headln.match(line): startHead(line)
          elif codeln.match(line): startCode(line)
          else:
               if newblock: startText(line)
               else: blocks[bl_num] = blocks[bl_num] + line
     elif state == "CODE":      # blank line does not change state
          if blankln.match(line): blocks[bl_num] = blocks[bl_num] + line
          elif headln.match(line): startHead(line)
          elif textln.match(line): startText(line)
          else: blocks[bl_num] = blocks[bl_num] + line
     else:
          raise ValueError, "unexpected input block state: "+state
```

The source file from which this code is taken can be downloaded with Txt2Html (see Resources). Notice that the variable `state` is declared `global`, and its value is changed in functions like `startText()`. The transition conditions, such as `textln.match()`, are regular expression patterns, but they could just as well be custom functions. The formatting itself is actually done later in the program. The state machine just parses the text file into labeled blocks in the `blocks` list.

An abstract state machine class

It is easy to use Python to implement an abstract state machine in form as well as function. This makes the state machine model of the program stand out more clearly than the simple conditional block in the previous example (in which the conditionals don't look all that different from any other conditionals, at first glance). Furthermore, the following class and its associated handlers do a good job of isolating in-state behavior. This improves both encapsulation and readability in many cases.

File: statemachine.py

```
from string import upper
classStateMachine:
  def __init__(self):
      self.handlers = {}
      self.startState = None
      self.endStates = []

  def add_state(self, name, handler, end_state=0):
      name = upper(name)
      self.handlers[name] = handler
      if end_state:
           self.endStates.append(name)

  def set_start(self, name):
      self.startState = upper(name)

  def run(self, cargo):
      try:
         handler = self.handlers[self.startState]
      except:
         raise "InitializationError", "must call .set_start() before .run()"

      if not self.endStates:
         raise  "InitializationError", "at least one state must be an end_state"

      while 1:
         (newState, cargo) = handler(cargo)
         if upper(newState) in self.endStates:
            break:
         else
            handler = self.handlers[upper(newState)]
```

The `StateMachine` class is really all you need for an abstract state machine. Because passing function objects in Python is so easy, this class uses far fewer lines than any similar class in another language would require.

To actually *use* the `StateMachine` class, you need to create some handlers for each state you want to use. A handler must follow a pattern. It loops and processes events until it's time to transition to another state, at which time the handler should pass back a tuple consisting of the new state's name and any cargo the new state-handler will need.

The use of `cargo` as a variable in the `StateMachine` class encapsulates the data needed by the state handler (which doesn't necessarily call its argument `cargo`). A state handler uses `cargo` to pass whatever is needed to the next handler so that the new handler can take over where the last handler left off. `cargo` typically includes a filehandle, which allows the next handler to read more data after the point where the last handler stopped. `cargo` might also be a database connection, a complex class instance, or a list with several items in it.

Now let's look at a test sample. In this case (outlined in following code example) the cargo is just a number that keeps getting fed back into an iterative function. The next value of `val` is always simply `math_func(val)`, as long as `val` stays within a certain range. Once the function returns a value outside that range, either the value is passed to a different handler or the state machine exits after calling a do-nothing end-state handler. One thing the example illustrates is that an *event* is not necessarily an input event. It can also be a computational event (atypically). The state-handlers differ from one another only in using a different marker when outputting the events they handle. This function is relatively trivial, and does not require using a state machine. But it illustrates the concept well. The code is probably easier to understand than its explanation!

File: statemachine_test.py

```
from statemachine import StateMachine
def ones_counter(val):
     print "ONES State: ",
     while 1:
         if val <= 0 or val >= 30:
             newState =  "Out_of_Range" ; breakelif 20 <= val < 30:
             newState =  "TWENTIES"; breakelif 10 <= val < 20:
             newState =  "TENS"; breakelse:print  " @ %2.1f+" % val, val = math_func(val)
     print " >>"
return (newState, val)

def tens_counter(val):
     print  "TENS State: ",
     while 1:
         if val <= 0 or val >= 30:
             newState =  "Out_of_Range"; breakelif 1 <= val < 10:
             newState =  "ONES"; breakelif 20 <= val < 30:
             newState =  "TWENTIES"; breakelse:print " #%2.1f+" % val,
          val = math_func(val)
     print  " >>"
return (newState, val)

def twenties_counter(val):
     print  "TWENTIES State:",
     while 1:
         if val <= 0  or  val >= 30:
             newState =  "Out_of_Range"; breakelif 1 <= val < 10:
             newState =  "ONES"; breakelif 10 <= val < 20:
             newState =  "TENS"; breakelse:print  " *%2.1f+" % val,
         val = math_func(val)
     print  " >>"
return (newState, val)
```

```
def math_func(n):
     from math import sin
     return abs(sin(n))*31

if __name__== "__main__":
       m = StateMachine()
       m.add_state("ONES", ones_counter)
       m.add_state("TENS", tens_counter)
       m.add_state("TWENTIES", twenties_counter)
       m.add_state("OUT_OF_RANGE", None, end_state=1)
       m.set_start("ONES")
       m.run(1)
```

Resources

- Read the previous installments of *Charming Python*.
- Download Txt2Html
- Download files used and mentioned in this article
- The concept of a state machine is, at a deeper level, closely related to the concepts of coroutines. If you want to make your brain hurt, you can read about Christian Tismer's Stackless Python, which efficiently implements coroutines, generators, continuations, and micro-threads. This is not for the faint of heart.
- Visit the home page of Pyxie, an open source XML processing library for Python
- Tour the Python Package Index, a Python code/tool repository

About the author

David Mertz

In a reticulated career, David Mertz has contributed more than his share to the web of knowledge. Most contributions have been in areas of academic "postmodern" philosophy (but then, this article also occupies several levels of descriptive "states"). You can contact David at mertz@gnosis.cx and find his life pored over at http://gnosis.cx/dW/. Suggestions and recommendations on this, past, or future columns are welcome.