Everything You Wanted To Know About Pickling, But Were Afraid to Ask!

[A look at the internals of pickling]

(Please take a moment to read the 2-sided “Cheat Sheet” that's being passed around)

Everything You Wanted To Know About Pickling, But Were Afraid to Ask (in 30 minutes or less, or your pickle is free!)

A PyCon 2011 Presentation

by Richard T. Saunders

Rincon Research Corporation

Outline

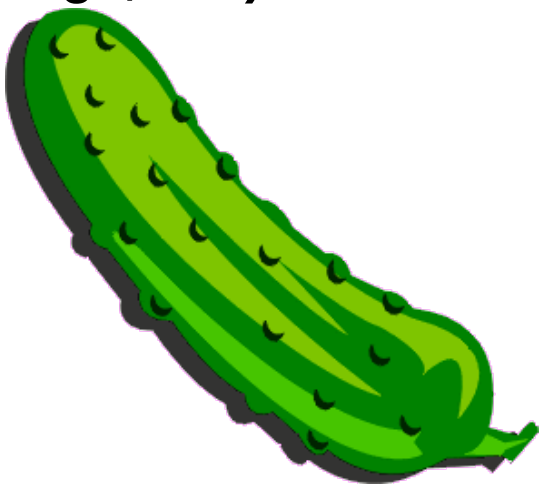
- Who Am I?
- What Is Pickling and Why Do I Care About It?
- Alternatives to Pickling/Timings
- Basics
- What Gets Stored with Different Protocols: 0,1,2,3
- The Stack Machine Model
- Advice, Experience, Don't-Do-That
- Where Can I Learn More?

Who Am I?

- Richard T. Saunders
  - maintain the PicklingTools library
    - http://www.picklingtools.com
    - open-source, BSD type license
    - library is 10+ years old, in current maintenance
  - reverse-engineered pickling to work with C++
    - colors my perspective: different lens than standard Python programmer
- Rincon Research Corporation (RRC) uses Python/C+/PicklingTools in 30+ projects
  - support engineers to get their job done

What Is Pickling and Why Do I Care About It?

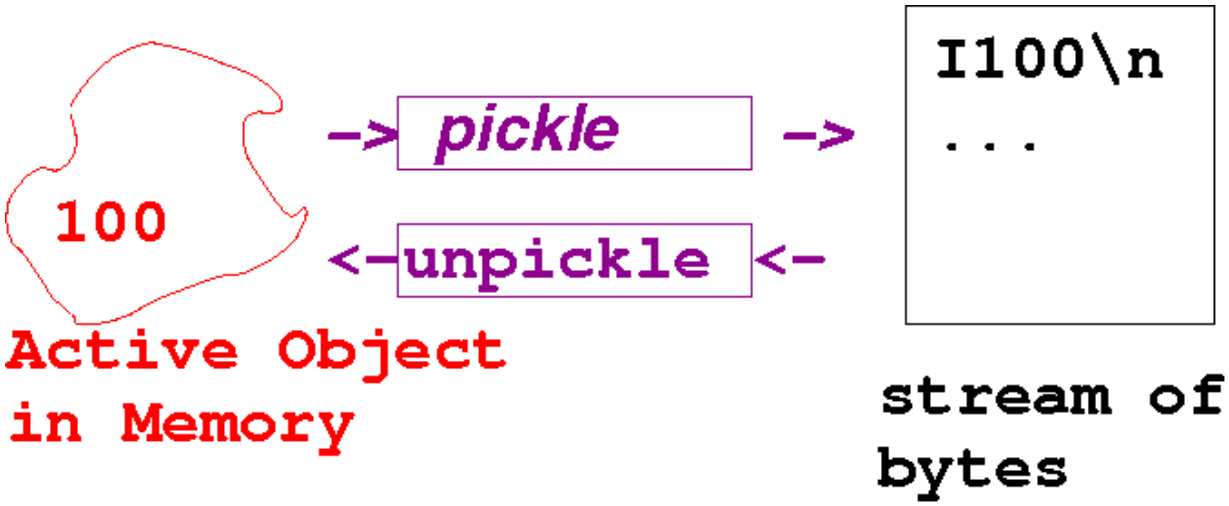
- Nothing to do with cucumbers and vinegar!
  - ...except that “pickling” traditionally means preserving food (cucumbers, cabbage, etc.) for a later date



  - We want to preserve state of our program for a later date!
    - Pickling is Python-speak for 'serializing'

Pickling / Unpickling

- **Pickling: taking a "snapshot" of some object in Python to store ("preserve") for later retrieval**
  - **store a pickle on disk**
  - **send a pickled object into a socket**
- **Unpickling: extracting object from "snapshot"**
  - **load a pickle from a file to get back object**
  - **grab a pickled object from a socket to get object**



```
>>> state={'inv':'key','room':1}

>>> import pickle
>>> save_game = file('1.pkl','w')
>>> pickle.dump(state,save_game,
                protocol=2)
>>> save_game.close() # flush
```

object to pickle

file to save to

protocol to use

```
... at a later date, restore game
>>> import pickle
>>> restore = file('1.pkl', 'r')
>>> state = pickle.load(restore)
# Automatically detects protocol

>>> print state
{'inv':'key','room':1}
```

file to unpickle from

unpickled object!

Alternatives to Pickling

- Text: print and eval
  - Advantages: Very easy to read/edit/modify with vi, emacs
  - Disadvantages: no sharing, very slow, lose Numeric prec
- Python marshall module
  - Advantages: fast
  - Disadvantages: no guarantees across versions of Python, no sharing, can't save user-defined classes
- XML and JSON: xml.dom, xml.sax, PyXB, json
  - Advantages: mindshare, tools, “sorta” readable
  - Disadvantages: ad-hoc mechanisms to translate Python data structures to XML/JSON
- Google protocol buffers
  - Advantages: very fast (10x? pickling), cross language
  - Disadvantages: not built-in, static preallocation

Why Pickling?

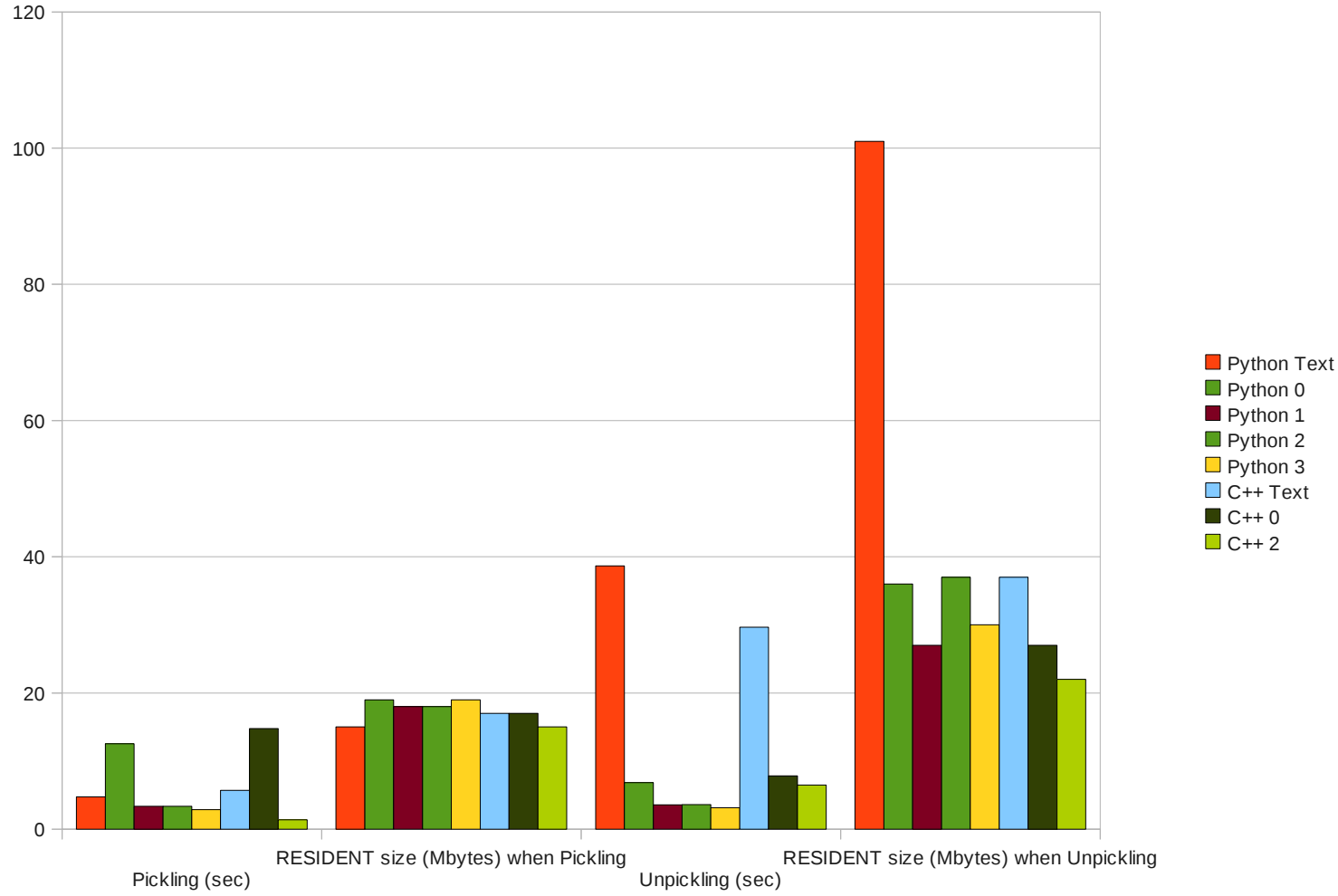
- Advantages
  - Supports automatic pickle/unpickle user-defined classes
  - Allows sharing in data structures (arbitrary “graphs”)
  - Reasonably Fast
  - Standard module: works across versions of Python
  - Built-in
  - Endemic:
    - Other Python libraries: shelve
    - C++ support: picklingtools.com, Boost Python
- Disadvantages
  - Other protocols may be faster
    - Marshall, Google Protocol Buffers
  - Not as readable on disk as other protocols
    - text, JSON, XML

Python Pickling Protocols

- 0: 7-bit ASCII clean, “text-readable”
- 1: Original binary protocol
- 2: Binary protocol introduced in Python 2.3
  - more efficient storage of new-style classes
- 3: Introduced in Python 3.0
  - explicit support for bytes
  - things pickled with Python 3.x can't be unpickled with Python 2.x modules

Rincon Research Corporation

Time / Space Costs of Different Protocols (shorter bars=better)



Basics: Pickling Examples

- See how the different protocols (0,1,2,3) dump the integer 15
- The `dump` pickles to a file, `dumps` pickles an object into a string

Pickling Protocol 0

- First Version
- 7-bit ASCII clean
  - Printable (with newlines)
- Can be slow, but very backwards compatible
  - If you need to span very different versions of Python, this may be your only choice

```
>>> import pickle
>>> pickle.dumps(15,protocol=0)
'I15\n.'
```

I15\n.

```
>>> import pickle
>>> pickle.dumps(15,protocol=0)
'I15\n.'
```

I15\n.

I: INT opcode
start of integer

```
>>> import pickle
>>> pickle.dumps(15,protocol=0)
'I15\n.'
```

I15\n.

15: ASCII text of integer

```
>>> import pickle
>>> pickle.dumps(15,protocol=0)
'I15\n.'
```

I15\n.

End of string marker (ASCII)

```
>>> import pickle
>>> pickle.dumps(15,protocol=0)
'I15\n.'
```

I15\n.

STOP opcode

Pickling Protocol 1

- **Available starting with version 1.x**
- **Binary**
  - **Faster than protocol 0**

```
>>> import pickle
>>> pickle.dumps(15,protocol=1)
'K\x0f.' # 3 bytes
```

K\x0f.

```
>>> import pickle
>>> pickle.dumps(15,protocol=1)
'K\x0f.' # 3 bytes
```

K: BININT1
opcode (0-255)

```
>>> import pickle
>>> pickle.dumps(15,protocol=1)
'K\x0f.' # 3 bytes
```

K\x0f.

15 in binary:a byte in memory,shown as hex digit

```
>>> import pickle
>>> pickle.dumps(15,protocol=1)
'K\x0f.' # 3 bytes
```

K\x0f.

STOP opcode

Pickling Protocol 2

- Available starting with Python 2.3
  - PEP 307 “Extensions to pickle protocol”
    - Smaller space to store user-defined classes

  “To date, each release of Python has been able to read pickles written by all previous releases.” - Protocol 2 preserves this
- Binary
  - Protocol 2 is a super-set of protocol 1
    - Backwards compatible, but add new opcodes
  - Voided security warranty (not that there was one ..)
    - FLATLY: Pickling was NOT meant to be secure serialization
    - If you want security, embed pickling in secure mechanism
      - ssh, key-exchanges, etc.

```
>>> import pickle
>>> pickle.dumps(15,protocol=2)
'\x80\x02K\x0f.' # 5 bytes
```

\x80\x02K\x0f.

```
>>> import pickle
>>> pickle.dumps(15,protocol=2)
'\x80\x02K\x0f.' # 5 bytes
```

\x80\x02K\x0f.

PROTO opcode: single byte: 128 (hex value shown)

```
>>> import pickle
>>> pickle.dumps(15,protocol=2)
'\x80\x02K\x0f.' # 5 bytes
```

\x80\x02K\x0f.

Protocol Version:
2 (single byte)
hex shown

```
>>> import pickle
>>> pickle.dumps(15,protocol=2)
'\x80\x02K\x0f.' # 5 bytes
```

\x80\x02K\x0f.

... rest is like protocol 1

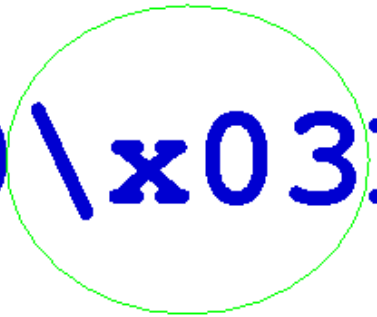
Pickling Protocol 3

- Available starting with Python 3
- Binary
- NOT guaranteed to be backwards compatible with previous versions
  - Strings vs. bytes issues
  - Class instances

```
>>> import pickle
>>> pickle.dumps(15,protocol=3)
b'\x80\x03K\x0f.' # 5 bytes
```

\x80\x03K\x0f.

```
>>> import pickle
>>> pickle.dumps(15,protocol=3)
b'\x80\x03K\x0f.' # 5 bytes
```



\x80\x03K\x0f.

Like protocol 2, but diff version (protocol 3)

Stack Machine

- Why does load side not have to specify protocol?
  - Version is embedded in the pickle at front: \x80\x02
    - Early versions of Python won't support
  - All versions of the loader support the same basic protocol using a STACK-BASED machine
- Conceptually two stacks:
  - Value stack: where you place a value right after you've unpickled it
  - Mark stack: to “mark” lengths of lists, dictionaries
    - cpickle uses 2 stacks, pickle uses 1 “combined” stack

Pickle Example with Lists

- Simple list: [1,2]
- Use protocol 1
  - Simplifies explanation
    - Protocol 2 and 3 are “structurally” the same
      - Demonstrating the stack-based nature

```
>>> import pickle
>>> pickle.dumps([1,2],protocol=1)
'](K\x01K\x02e.'
```

```
](K\x01K\x02e.
```

```
>>> import pickle
>>> pickle.dumps([1,2],protocol=1)
'](K\x01K\x02e.'
```

```
](K\x01K\x02e.
```

EMPTY_LIST op:
pushes [] on
values stack

```
>>> import pickle
>>> pickle.dumps([1,2],protocol=1)
']](K\x01K\x02e.'
```

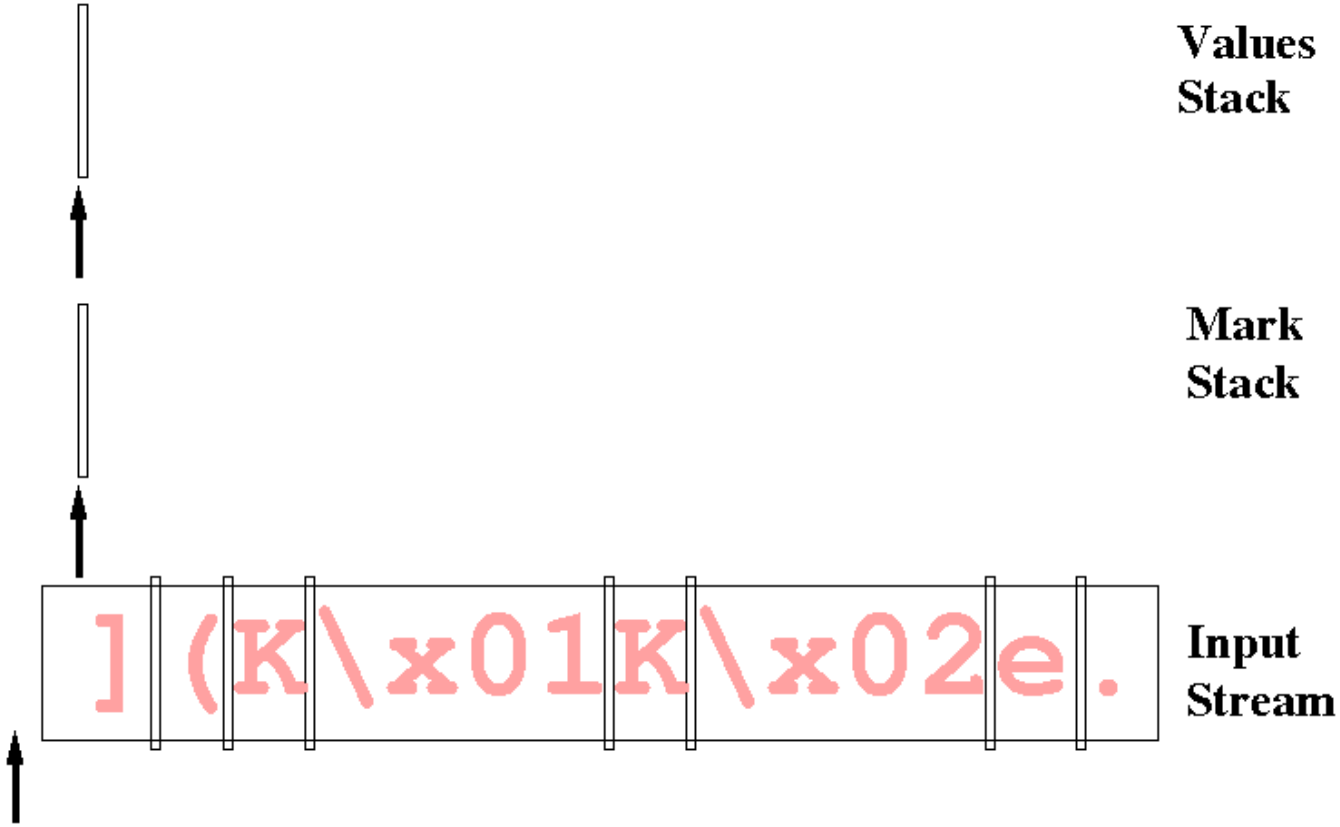
```
](K\x01K\x02e.
```

MARK opcode:
marks the top value (pushes index info on mark stack)

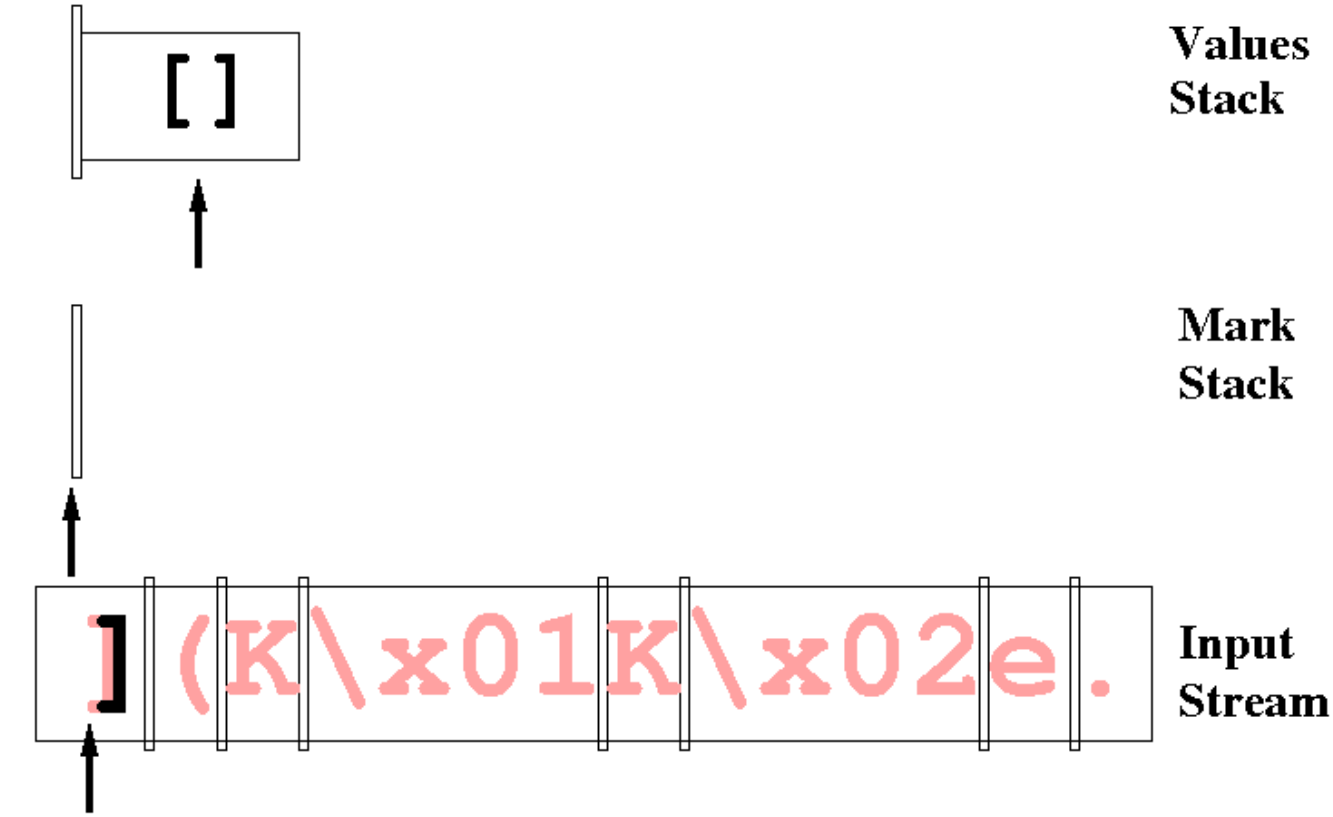
```
>>> import pickle
>>> pickle.dumps([1,2],protocol=1)
'](K\x01K\x02e.'
```

```
](K\x01K\x02e.
```

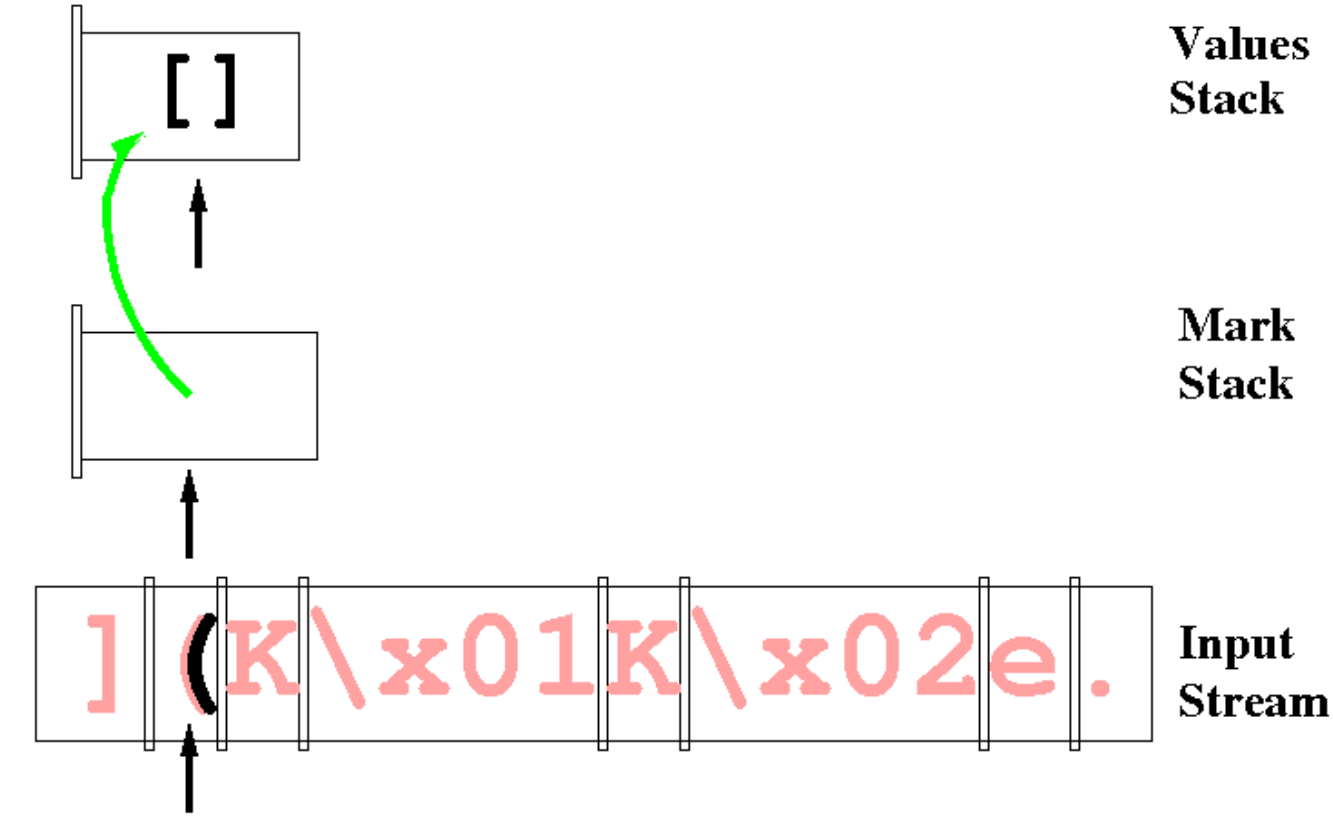
APPENDS opcode:
finds last mark
& appends values
from MARK-APPENDS
into empty list



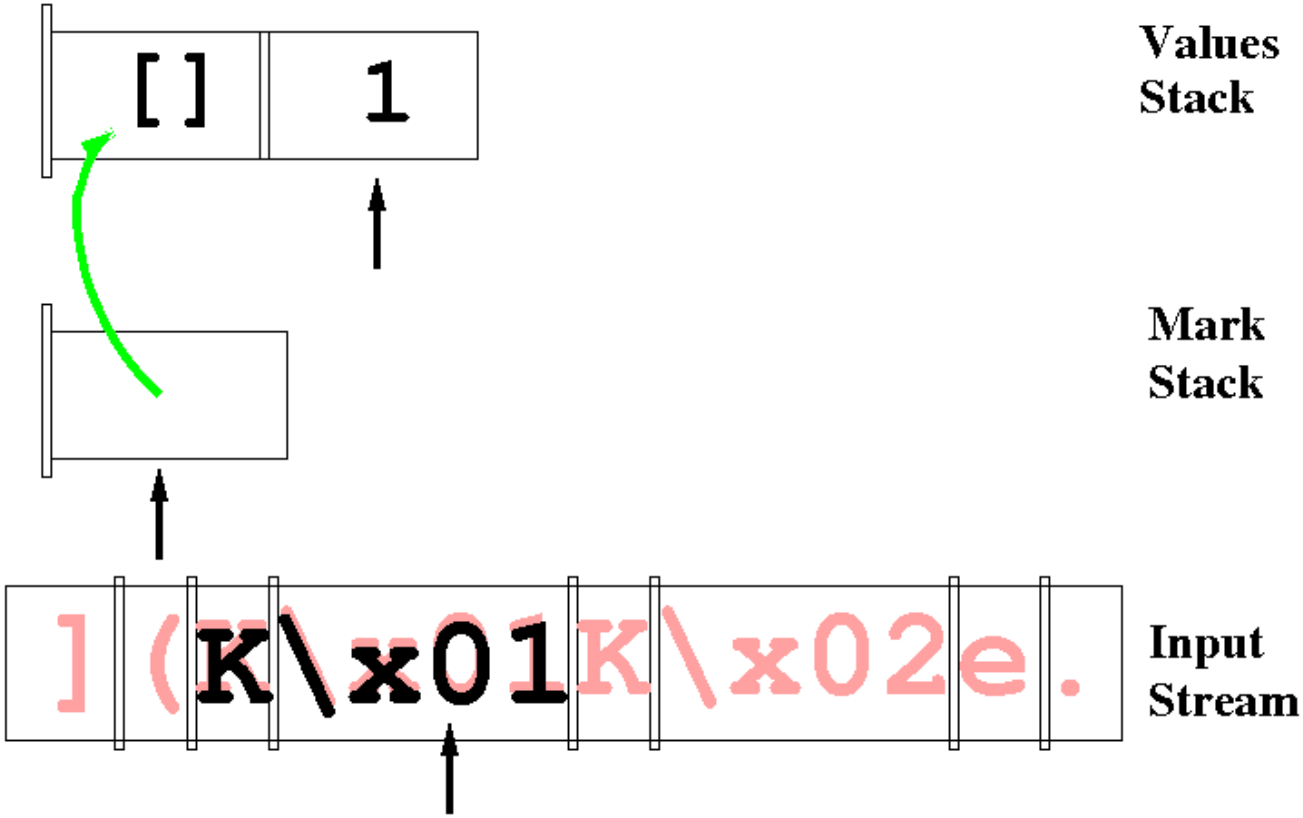
Comment: Initial state: start of input stream
Values stack empty
Mark stack empty



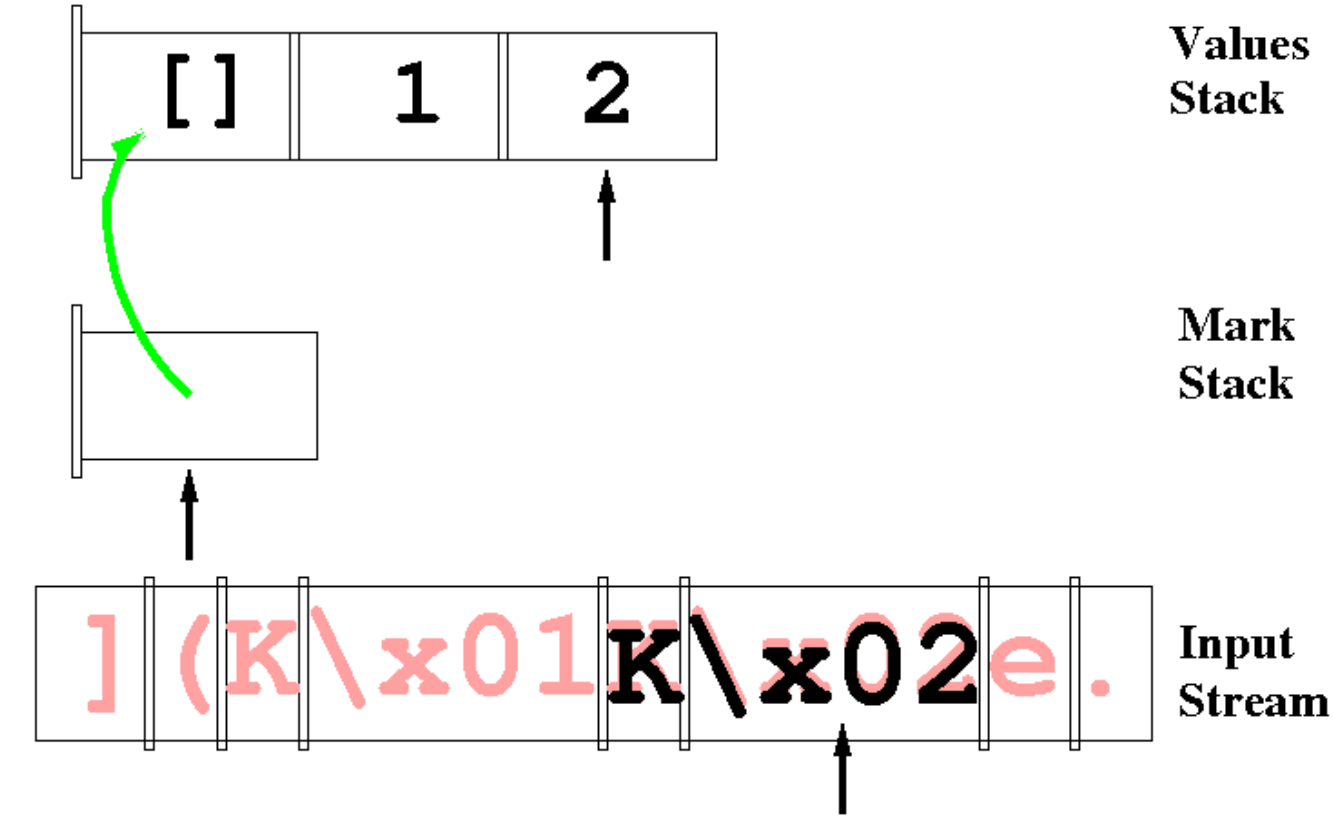
Comment: Empty list pushed on top of values stack



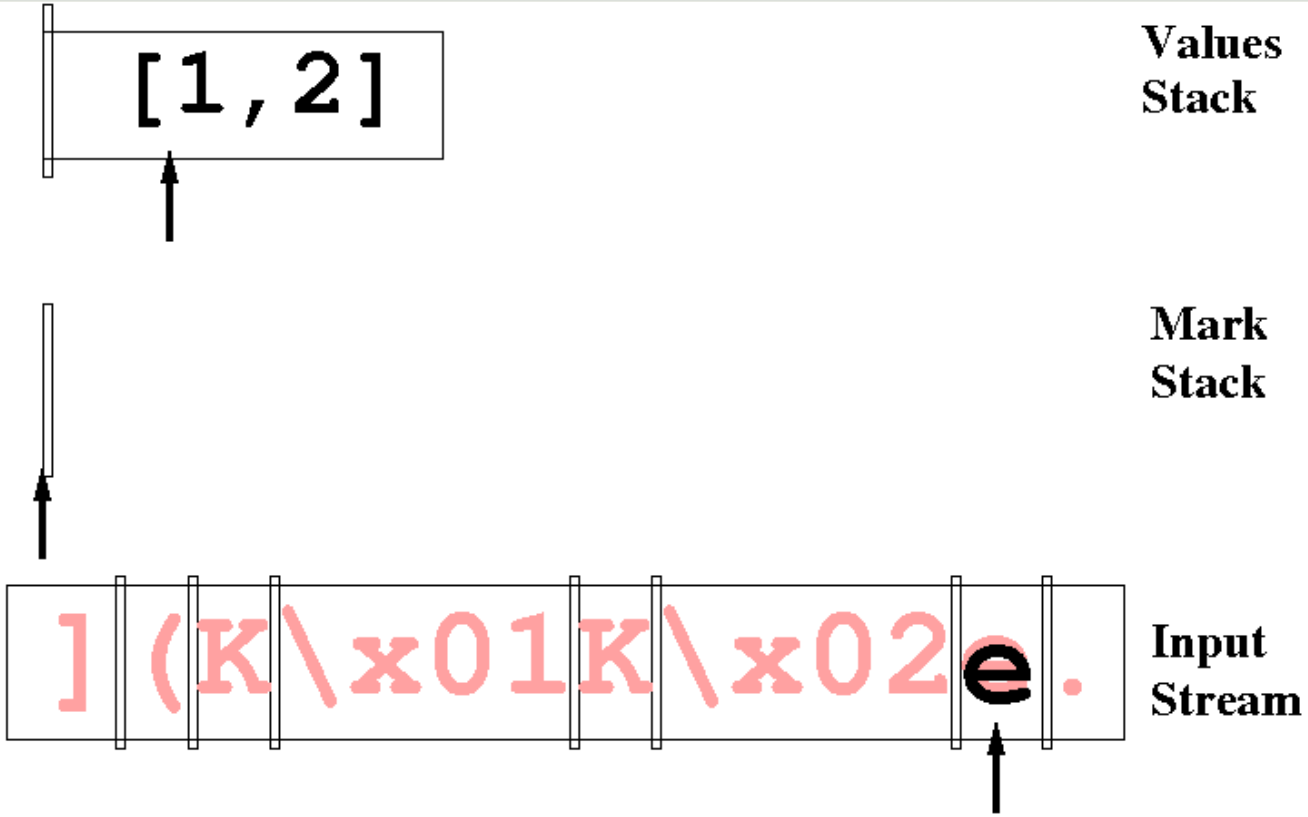
Comment: Mark where current top of values stack is



Comment: Push integer 1 onto values stack



Comment: Push integer 2 onto values stack

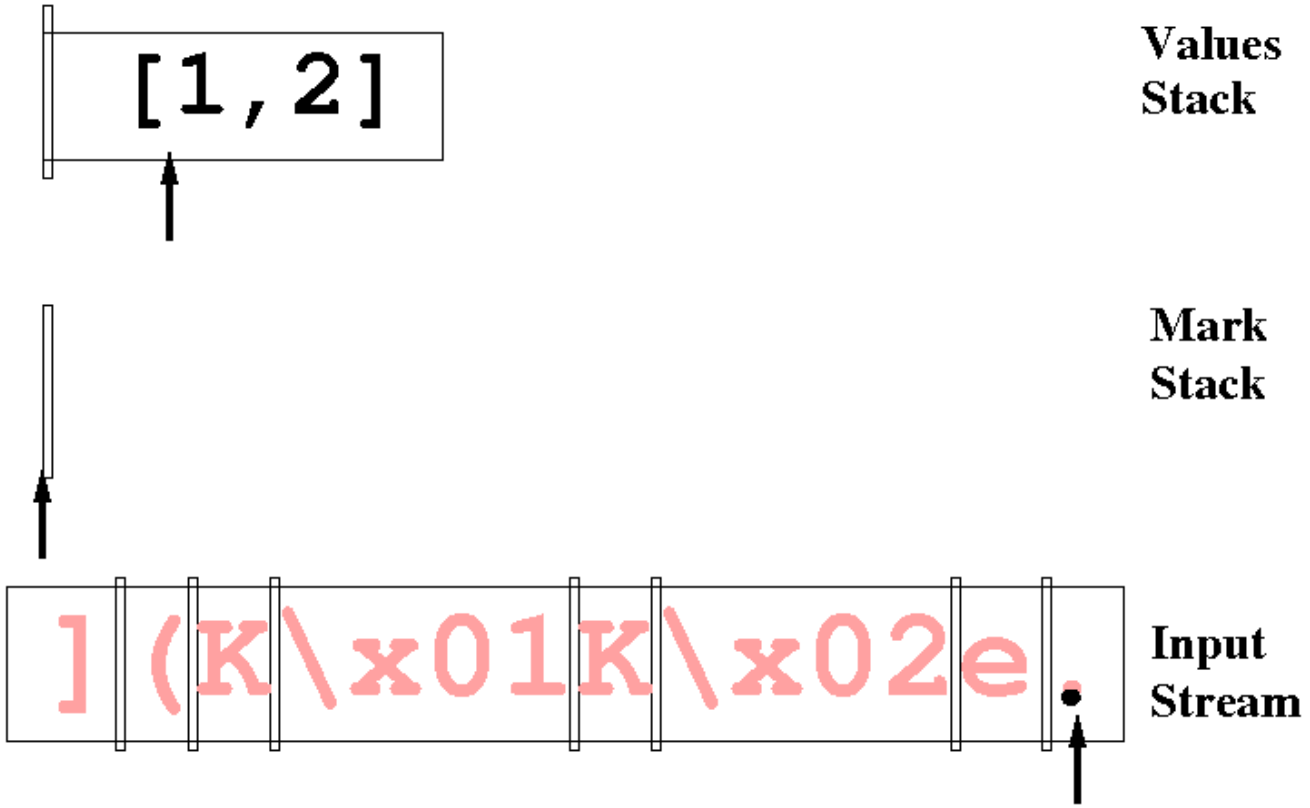


Comment: Encountered APPENDS:

(a) pop mark stack: tells range of APPENDS

(b) APPEND values into marked list

– this pops the values stack as APPEND



Comment: STOP opcode: all done!
The top of the values stack is what we return! [1,2]

How Does Protocols Handle Sharing?

- somelist = [1,2]
- share = [somelist, somelist]
- Text:
  - >>> print share
  - [[1,2], [1,2]] # Looks like two separate lists
- Binary:
  - Use special opcodes to indicate we are sharing
  - PUT opcode points to what could be shared
  - GET opcode shares

```
>>> import pickle
>>> inner = []
>>> outer = [inner, inner]
>>> outer[0] is outer[1]
True
>>> copies = [[],[]]
>>> copies[0] is copies[1]
False
```

```
>>> import pickle
>>> inner = []
>>> outer = [inner, inner]
>>> pickle.dumps(outer, protocol=1)
']q\x00(]q\x01h\x01e.'
```

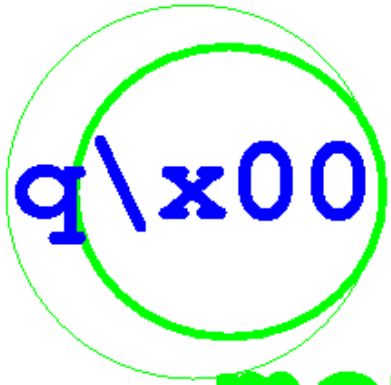
]q\x00(]q\x01h\x01e.

```
>>> import pickle
>>> inner = []
>>> outer = [inner, inner]
>>> pickle.dumps(outer, protocol=1)
']q\x00(]q\x01h\x01e.'
```

```
]q\x00(]q\x01h\x01e.
```

PUT opcode:
remember object on top of stack

```
>>> import pickle
>>> inner = []
>>> outer = [inner, inner]
>>> pickle.dumps(outer, protocol=1)
']q\x00(]q\x01h\x01e.'
```



```
]q\x00(]q\x01h\x01e.
```

...memo number for top of stack is 0 (outer list)

```
>>> import pickle
>>> inner = []
>>> outer = [inner, inner]
>>> pickle.dumps(outer, protocol=1)
']q\x00(]q\x01h\x01e.'
```

]q\x00(]q\x01h\x01e.

another PUT with memo 1 for the inner list

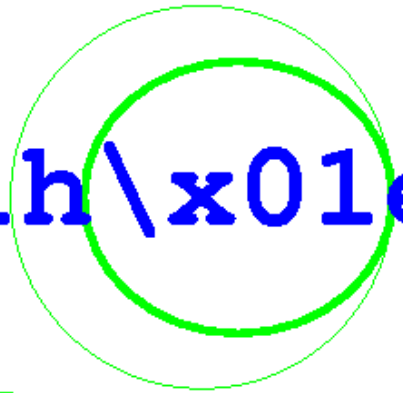
```
>>> import pickle
>>> inner = []
>>> outer = [inner, inner]
>>> pickle.dumps(outer, protocol=1)
']q\x00(]q\x01h\x01e.'
```

```
]q\x00(]q\x01h\x01e.
```

GET opcode!

reuses previous PUT

```
>>> import pickle
>>> inner = []
>>> outer = [inner, inner]
>>> pickle.dumps(outer, protocol=1)
']q\x00(]q\x01h\x01e.'
```



```
]q\x00(]q\x01h\x01e.
```

GET memo 1
(the inner list)

Not deep copy!

Noticed "Philosophies" of the Pickling Modules

- What I've noticed (not necessarily explicit)
  - Do not deserialize directly into the list, dict, etc.
    - Stack-based model
    - Python object model (moving pointers) easy to move into final destination very quickly
    - Could implement as direct insert... might be faster?
  - Have a special "marker" to indicate when things done
    - \n on protocol 0 ("I100\n")
    - MARK opcode until reaches APPENDS, SETITEMS, etc.
      - In contrast to count in the stream (would take up a full integer)

Experience and Recommendations

- Use Python Dictionaries to keep your state
  - uses first-class object of the language
  - easy to manipulate, change, read, write
- Store Python Dictionaries as plain “pretty printed” text for disk files
  - { 'savegame': 12, 'doors':12, 'startroom':1 }
  - Can easily manipulate with standard text tools
- Use Python Dictionaries as pickle protocol 2 for across sockets (3 if embraced Python 3x)
  - much faster

Further Information

- From the Python Documentation
  - http://docs.python.org/library/pickle.html
- In the source code for Python
  - Modules/cPickle.c # C implementation, fast
  - Libs/pickle.py # Python impl, easy to read
  - Libs/pickletools.py
- From the PicklingTools modules
  - C++/pickleloader.h # C++ loader for different object model (so can pickle in C++)
  - C++/m2pythonpickler.h,cc # C++ dump for different object model (so can load from C++)

Questions?