Functions in Python Part 3

Prof. Pai H. Chou
National Tsing Hua University

Outline

- Lambda expressions
- Functions that take plug-in functions
- Non-local symbols in lambda vs. function
- Functions in table
- Inner functions

Example revisited: list.sort

```
>>> L = [('apple',20),('oranges',15),('guava',12),('mango',60)]
>>> M = L[:]
>>> M.sort()
[('apple', 20), ('guava', 12), ('mango', 60), ('oranges', 15)]
```

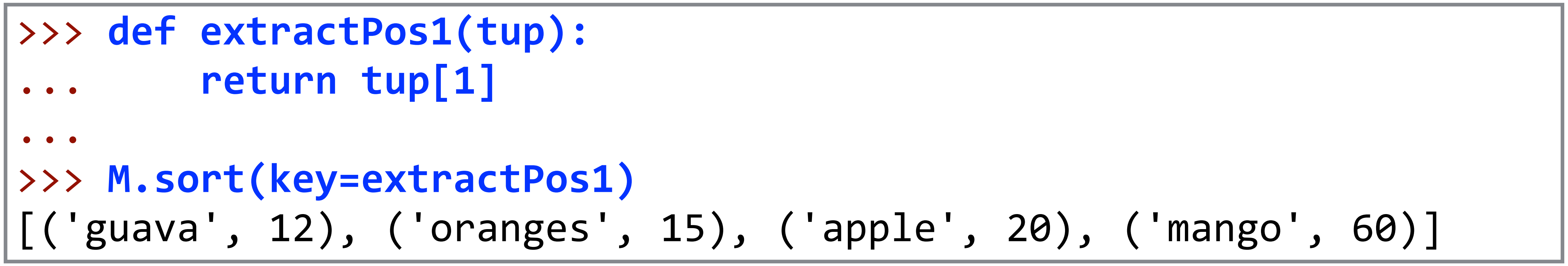
- What if we want to sort by price?

```
>>> help(L.sort)
Help on built-in function sort:

sort(*, key=None, reverse=False) method of builtins.list instance
    Stable sort *IN PLACE*.
```

- What is key?
  - key is a **plug-in function** that you call on each item to obtain the key to sort

Define a function to enable sorting by different keys



```
>>> def extractPos1(tup):
...     return tup[1]
...
>>> M.sort(key=extractPos1)
[('guava', 12), ('oranges', 15), ('apple', 20), ('mango', 60)]
```

  - Now list is sorted by price order (12, 15, 20, 60)

- plug-in function as a parameter to sort
  - sort() calls the plug-in on each element of the list to obtain the key for comparing their order
  - advantage: the element themselves don't have to be comparable, as long as there is a way to map them to keys that can be compared

Lambda expressions

- Lambda = an anonymous function
- Syntax:

  ```
  lambda arg1, arg2, ... : expression
  ```

  - it's like defining function

    ```
    def F(arg1, arg2, ...):
        return expression
    ```

  - but does not define the name 'F' in any scope
- Main purpose: pass it as a "plug-in"

Named function vs. anonymous function (lambda)

- Named function as plug-in

```
>>> def extractPos1(tup):
...     return tup[1]
...
>>> M.sort(key= extractPos1)
```

  - seems wasteful to define a whole function just for a simple selection

- Anonymous function as plug-in

```
>>> M.sort(key= lambda s: s[1])
```

  - concise, like an in-line expression passed as actual parameter

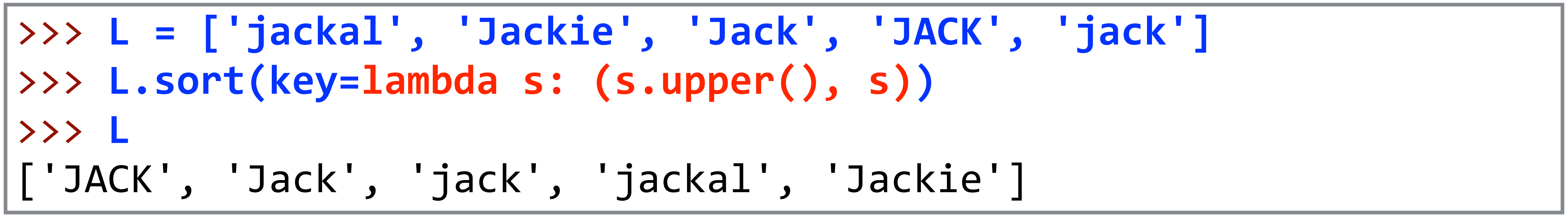
More examples of lambda in sort

- sorting in "true dictionary order"
  - case insensitive normally, but in case of tie, capitalized is ordered first
- Example
  - "JACK" < "Jack" < "jack" < "jackal" < "Jackie"

all ordered before "jackal" or "Jackie";
but "JACK" < "Jack" because "A < a",
"Jack" < "jack" because "J" < "j"

Solution to "true dictionary order"

- two-level comparison
  - Primary order: convert all words to all upper case
    - so "JACK", "Jack", "jack" => all "JACK"
  - Secondary order: use original capitalization as tie-breaker
- `lambda s: (s.upper(), s)`



```
>>> L = ['jackal', 'Jackie', 'Jack', 'JACK', 'jack']
>>> L.sort(key=lambda s: (s.upper(), s))
>>> L
['JACK', 'Jack', 'jack', 'jackal', 'Jackie']
```

Another use of lambda: sorting day-of-week strings

- Want days of week to be sorted by
  - ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
  - problem: string sort is alphabetical, not by day of week
- Solution: map `str` to an `int` as key
  - `L.sort(key=lambda s: ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'].index(s))`
  - `L.sort(key=lambda s: {'Sun':0, 'Mon':1, 'Tue':2, 'Wed':3, 'Thu':4, 'Fri':5, 'Sat':6}[s])`
- Issue: efficiency
  - `list.index` is linear search, may be less efficient than `dict` for large table

Two built-in functions that take plug-in functions: map(), filter()

- `map(`*f*, *seq*`)`
  - like `[`*f*(*x*) **`for`** *x* **`in`** *seq*`]` but as an iterator
- `map(`*f*, *A*`,` *B*`,` …`)` # general form
  - like `[`*f*(*A*`[0]`,*B*`[0]`,…)`,` *f*(*A*`[1]`,*B*`[1]`,…)`, ...]` but as iterator
- `filter(`*f*, *seq*`)`
  - like `[`*x* **`for`** *x* **`in`** *seq* **`if`** *f*(*x*)`]` but as iterator

map(): convert between str and ASCII in a sequence

- Example: want to get list of ASCII codes for characters in a string

```
>>> # like [ord('h'), ord('e'), ord('l'), ord('l'), ord('o')]
>>> # or you can say [ord(x) for x in 'hello']
>>> list(map(ord, "hello"))
[104, 101, 108, 108, 111]
```

- Convert ASCII code back to string

```
>>> # map does chr(104), chr(101), chr(108), chr(108), chr(111).
>>> # same as 'h', 'e', 'l', 'l' 'o' in a sequence, then join
>>> ''.join(map(chr, [104, 101, 108, 108, 111]))
'hello'
```

map() with lambda

- Example: want to negate a list of numbers

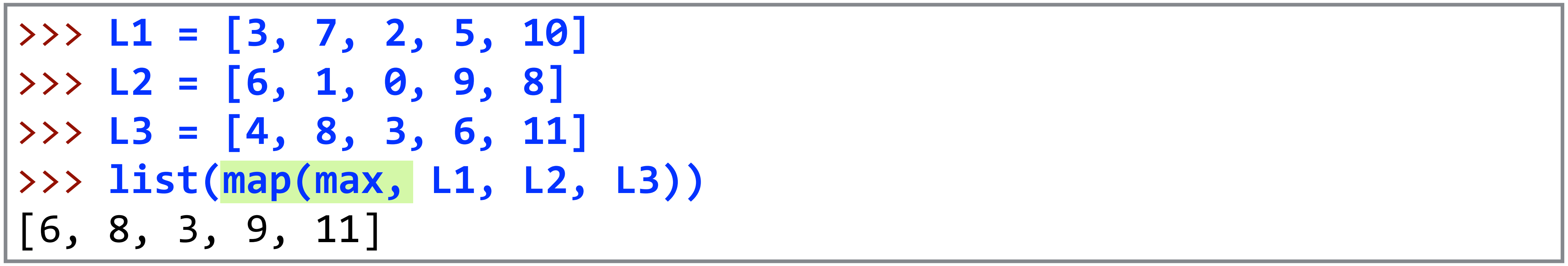
```
>>> L = [2, 5, -3, -1, 4]
>>> [-x for x in L]  # first way: list comprehension
[-2, -5, 3, 1, -4]
>>> list(map(lambda x: -x, L)) # second way: map lambda x: -x
[-2, -5, 3, 1, -4]
>>> from operator import neg
>>> list(map(neg, L)) # third way: map operator.neg
[-2, -5, 3, 1, -4]
```

- Example: double each number

```
>>> [x*2 for x in L]  # first way: list comprehension
[4, 10, -6, -2, 8]
>>> list(map(lambda x: x*2, L)) # second way: map lambda x: x*2
[4, 10, -6, -2, 8]
```

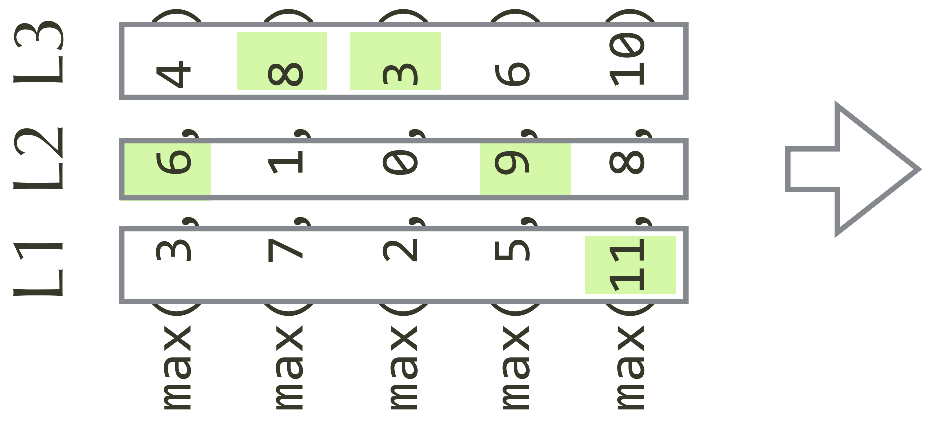
map() with multiple sequences

- map(f, A, B, C..)
  - is like [f(A[0], B[0], C[0], ..), f(A[1], B[1], C[1], ..), ... f(A[n-1], B[n-1], C[n-1], ...)]
  - ex: max of corresponding elements in three lists



```
>>> L1 = [3, 7, 2, 5, 10]
>>> L2 = [6, 1, 0, 9, 8]
>>> L3 = [4, 8, 3, 6, 11]
>>> list(map(max, L1, L2, L3))
[6, 8, 3, 9, 11]
```

map



[6, 8, 3, 9, 11]

List comprehension with multiple sequences: use zip()

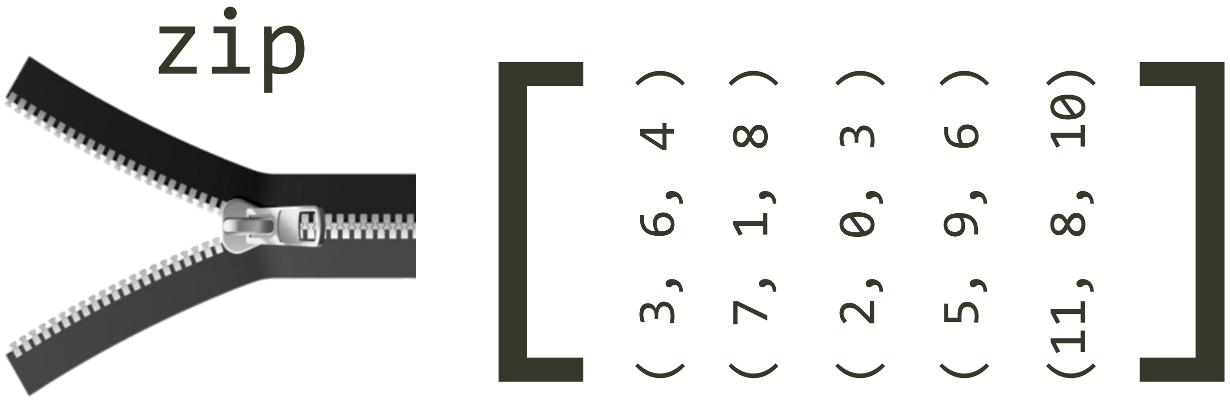
- zip(A, B, C,..)
  - like [(A[0], B[0], C[0],..), (A[1], B[1], C[1], ..), ...]
  - example: list comprehension version of map(max

```
>>> L1 = [3, 7, 2, 5, 10]
>>> L2 = [6, 1, 0, 9, 8]
>>> L3 = [4, 8, 3, 6, 11]
>>> [max(*t) for t in zip(L1, L2, L3)]
[6, 8, 3, 9, 11]
```

```
L1   = [3, 7, 2, 5, 10]

L2   = [6, 1, 0, 9, 8]

L3   = [4, 8, 3, 6, 11]
```



```
[ ( 3, 6, 4 ) ( 7, 1, 8 ) ( 2, 0, 3 ) ( 5, 9, 6 ) (11, 8, 10) ]
```

filter() on sequence

- like [*x* **for** *x* **in** *seq* **if** *f*(*x*)] but as iterator
- Example: filter out negative numbers

```
>>> L1 = [10, -3, 0, 5, -9, 8, 7, -2]
>>> i2 = filter(lambda n: n >= 0, L1)
>>> list(i2)
[10, 0, 5, 8, 7]
```

  - `lambda n: n >= 0` evaluates to `False` if its actual argument < 0

Function in data structures

- Functions can be put in a table
  - list, dict, etc.
- Use indexing or key lookup to pick a function to execute
- Why? potentially cleaner organization
  - often useful when dispatching a command

Stack interpreter example rewritten with "function table"

original if-elif version

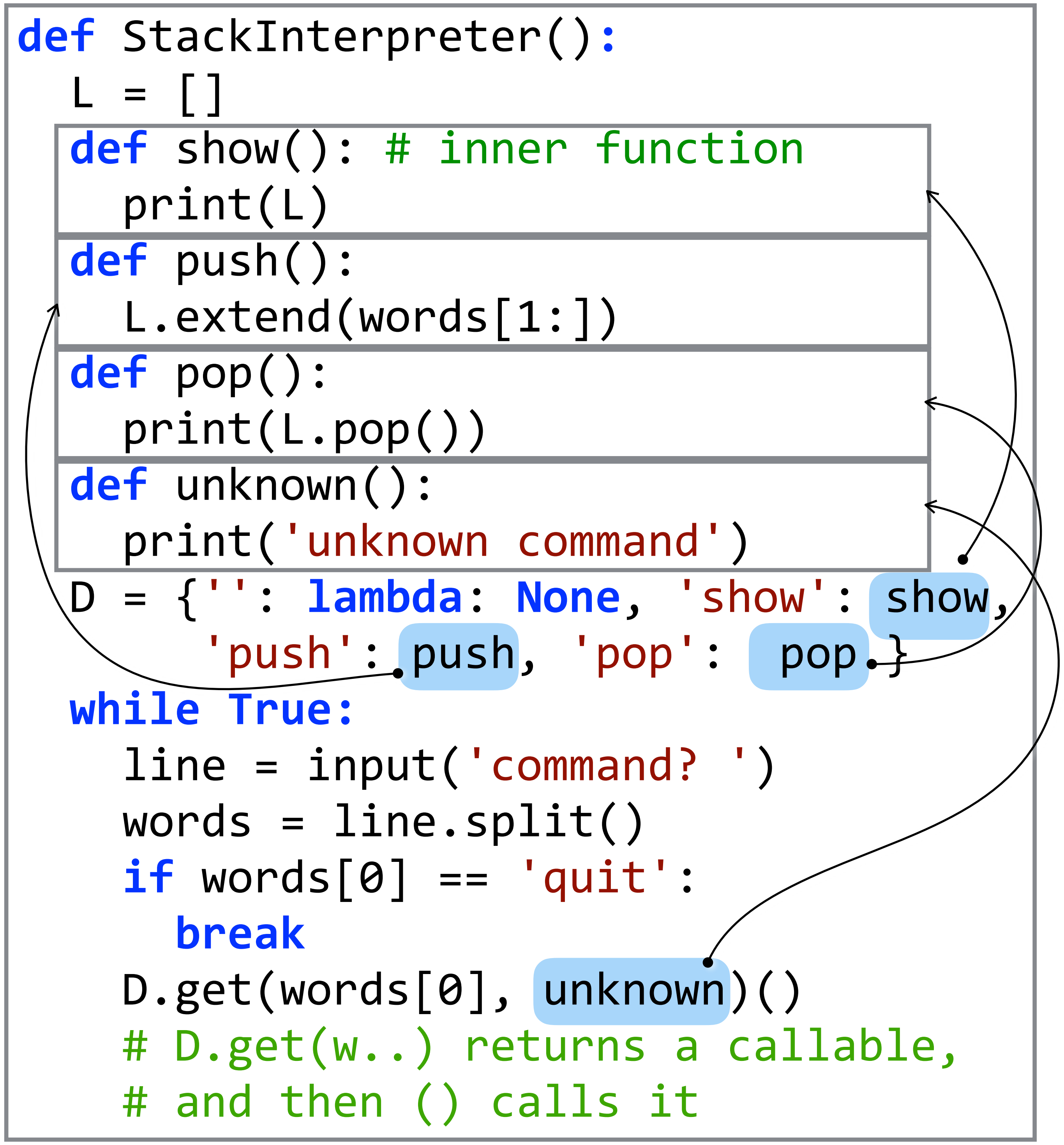
```
def StackInterpreter():
  L = []
  while True:
    line = input('command? ')
    words = line.split()
    if len(words) == 0:
      pass
    elif words[0] == 'show':
      print(L)
    elif words[0] == 'push':
      L.extend(words[1:])
    elif words[0] == 'pop':
      print(L.pop())
    elif words[0] == 'quit':
      break
    else:
      print('unknown command')
```

function-table version

```
def StackInterpreter():
  L = []
  D = {'':     lambda: None,
       'show': lambda: print(L),
       'push': lambda: L.extend(words[1:]),
       'pop':  lambda: print(L.pop()) }
  while True:
    line = input('command? ')
    words = line.split()
    if words[0] == 'quit':
      break
    D.get(words[0],\
        lambda:print('unknown command'))()
```

Alternative to lambda in function table: inner functions

- Inner function
  - function defined within another function
- Advantages
  - named, instead of exposing detail like lambda
  - has same access to parent function's local variables
  - does not pollute name space



```
def StackInterpreter():
  L = []
  def show(): # inner function
    print(L)
  def push():
    L.extend(words[1:])
  def pop():
    print(L.pop())
  def unknown():
    print('unknown command')
  D = {'': lambda: None, 'show': show,
       'push': push, 'pop':  pop }
  while True:
    line = input('command? ')
    words = line.split()
    if words[0] == 'quit':
      break
    D.get(words[0], unknown)()
    # D.get(w..) returns a callable,
    # and then () calls it
```

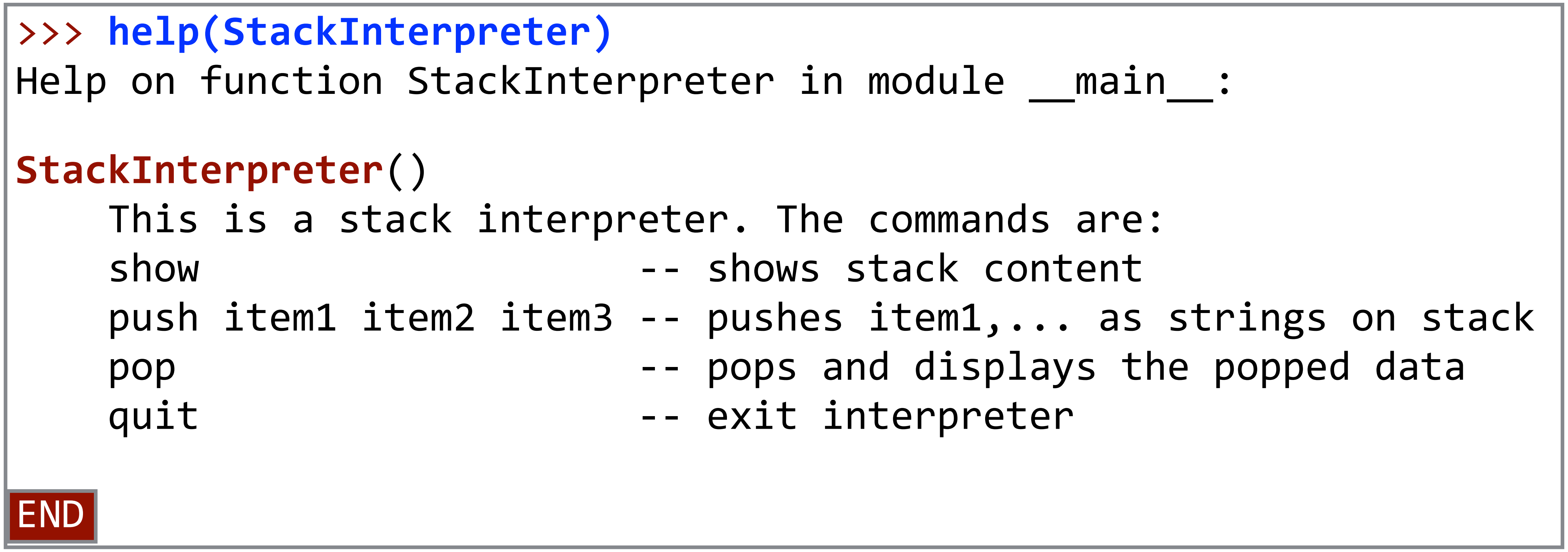
DocString: documentation string

- DocString
  - a triple-quoted string immediately after **def** line
  - purpose: string to display when `help(`*function*`)`
- Example

```
def StackInterpreter():
    """This is a stack interpreter. The commands are:
    show                   -- shows stack content
    push item1 item2 item3 -- pushes item1,... as strings on stack
    pop                    -- pops and displays the popped data
    quit                   -- exit interprets
    """
    L = []
    D = {'':     lambda: None,
    ...
```

Use of DocString

- Interactive help



```
>>> help(StackInterpreter)
Help on function StackInterpreter in module __main__:

StackInterpreter()
    This is a stack interpreter. The commands are:
    show                     -- shows stack content
    push item1 item2 item3 -- pushes item1,... as strings on stack
    pop                      -- pops and displays the popped data
    quit                     -- exit interpreter

END
```

- Do this on all public functions (i.e., you want other people to call)
- May want to include explanation of parameters, return values, assumptions

Python Style Guide

- https://www.python.org/dev/peps/pep-0008/
- Naming convention:
  - use ***snake_case*** instead of ***CamelCase*** (capitalized words) for function names and (most) variable names!
- Example

✋ CamelCase: `def StackInterpreter():`

  - Used for legacy code compatibility

✓ snake_case: `def stack_interpreter():`

  - Used for new code