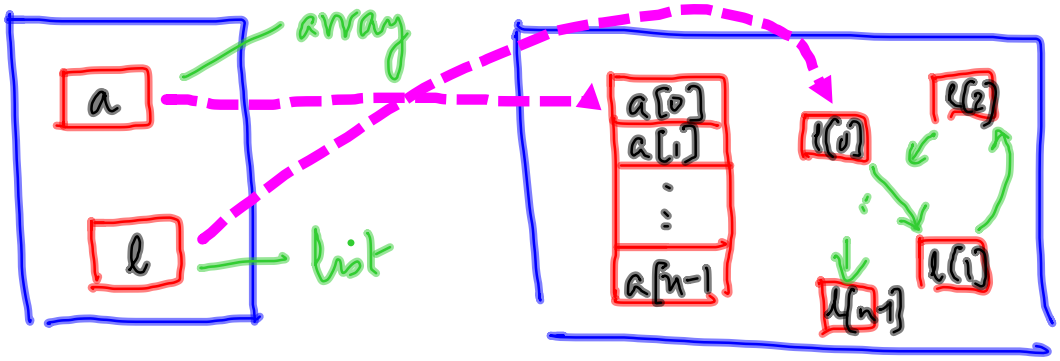
Lists vs Arrays

Collection of values with a single name, indexed by $0, 1, \ldots n-1$

$a[0], a[1], \ldots, a[n-1]$

Lists are "flexible" - insert, append etc

Array is like a list, but it is a contiguous block of memory



Arrays also support indexed naming a[0],...

Location of a[i] can be computed from location of a[0] → "random" access

Cost of locating a[i] is independent of i

In a typical list,

start at a[0] and follow links to a[i]

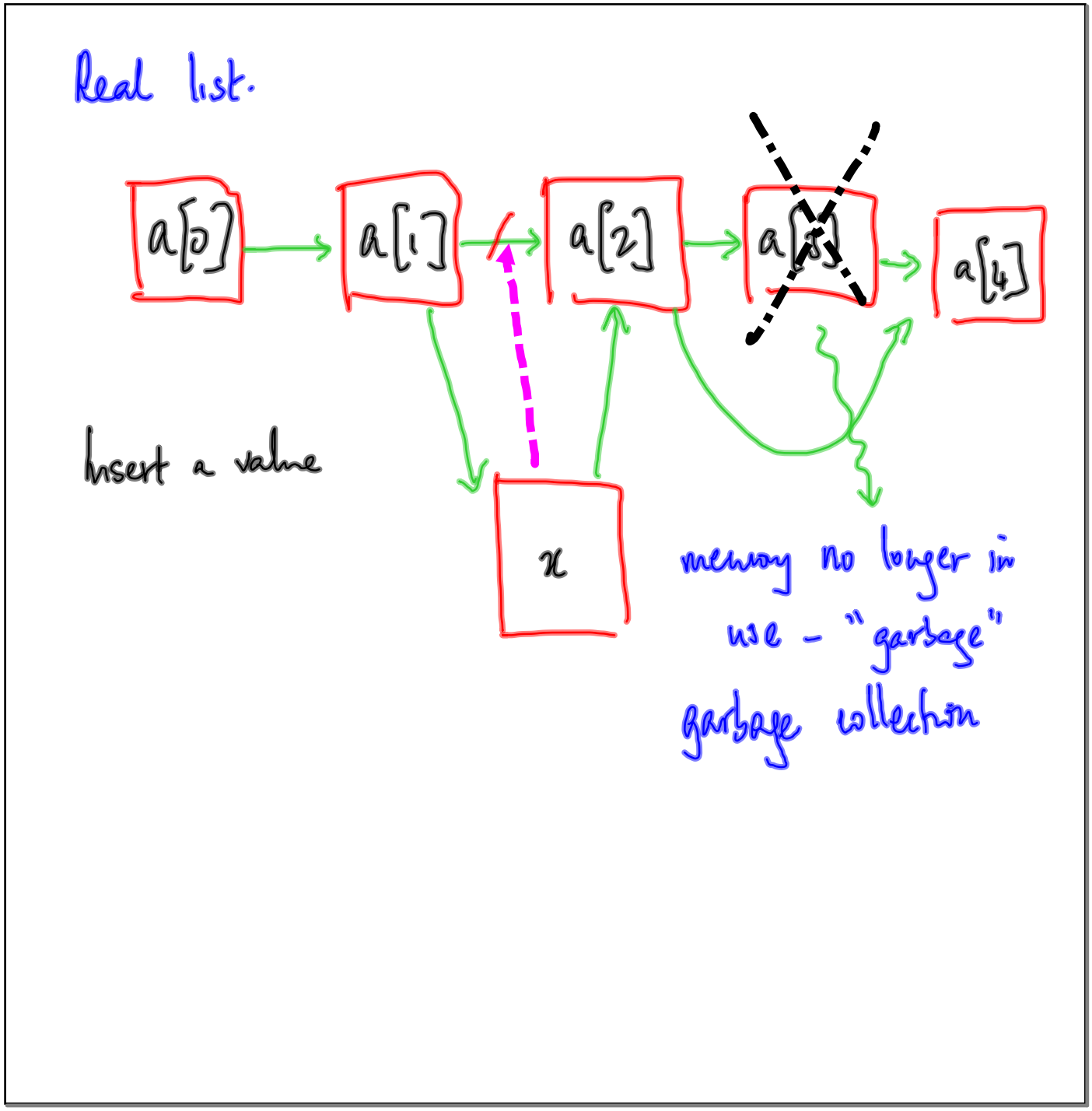
Tradeoff

Random access vs Flexibility

Arrays:

Size is fixed

Insertion/deletion requires shifting a block


Real list.
a[0]
a[1]
a[2]
a[3]
a[4]
Insert a value
x
memory no longer in use - "garbage"
garbage collection

Garbage collection vs Explicit memory management

Implicit

C

ask for space

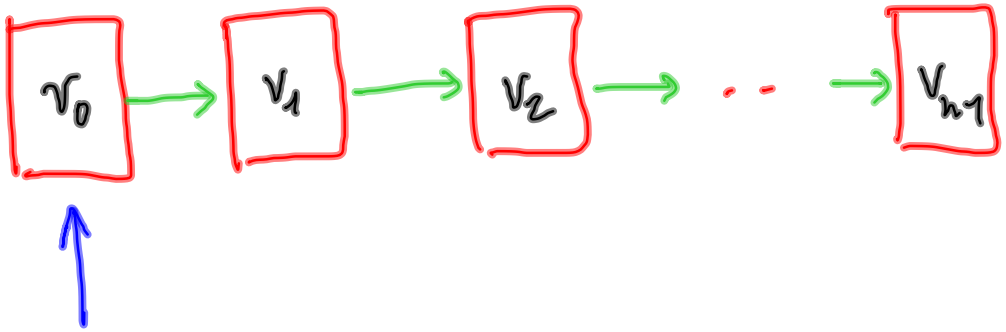
return it when no longer needed

If you forget, memory usage increases with time

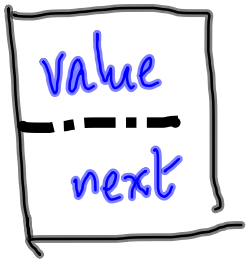
"Memory Leak"

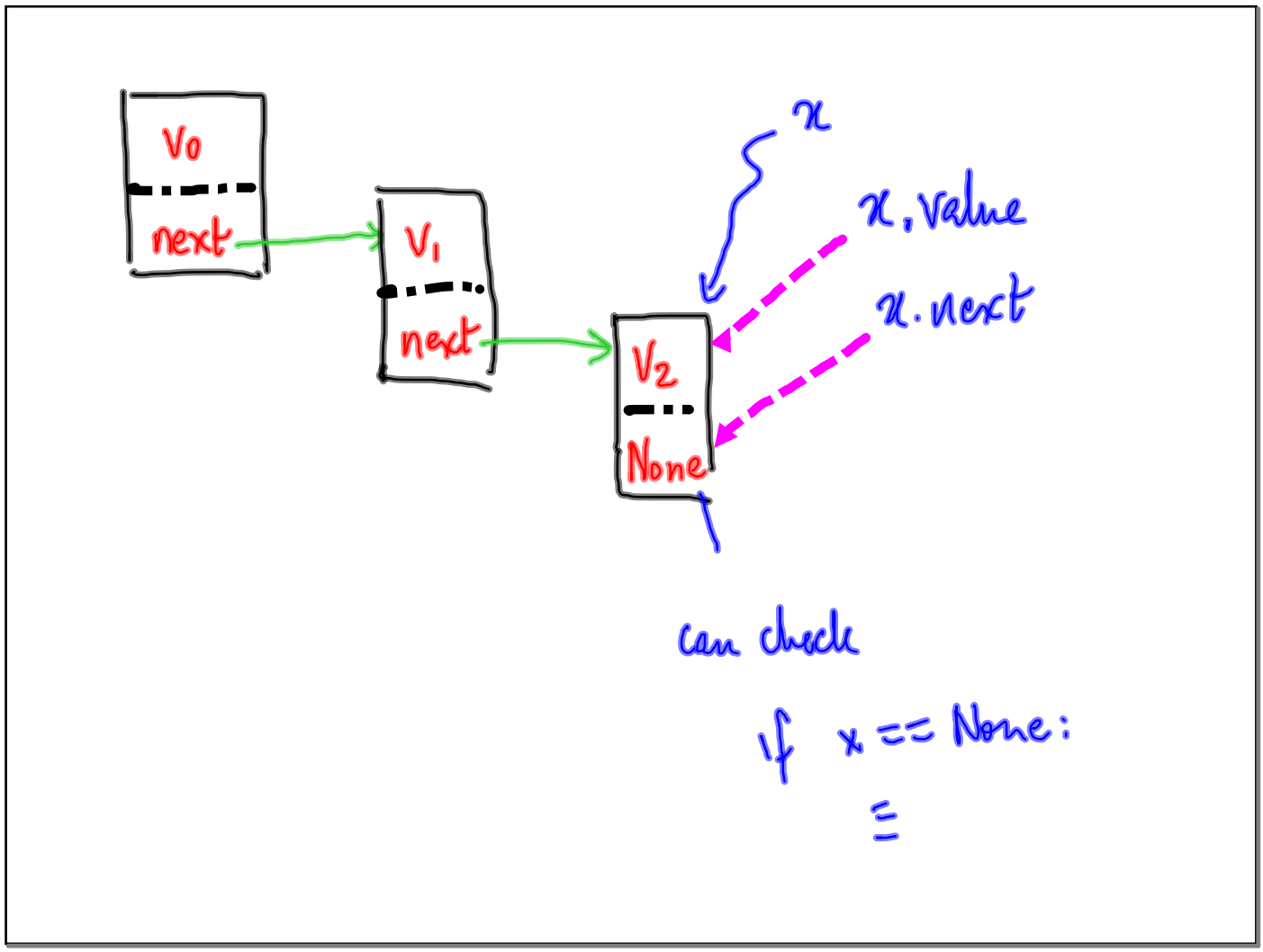
How to program lists

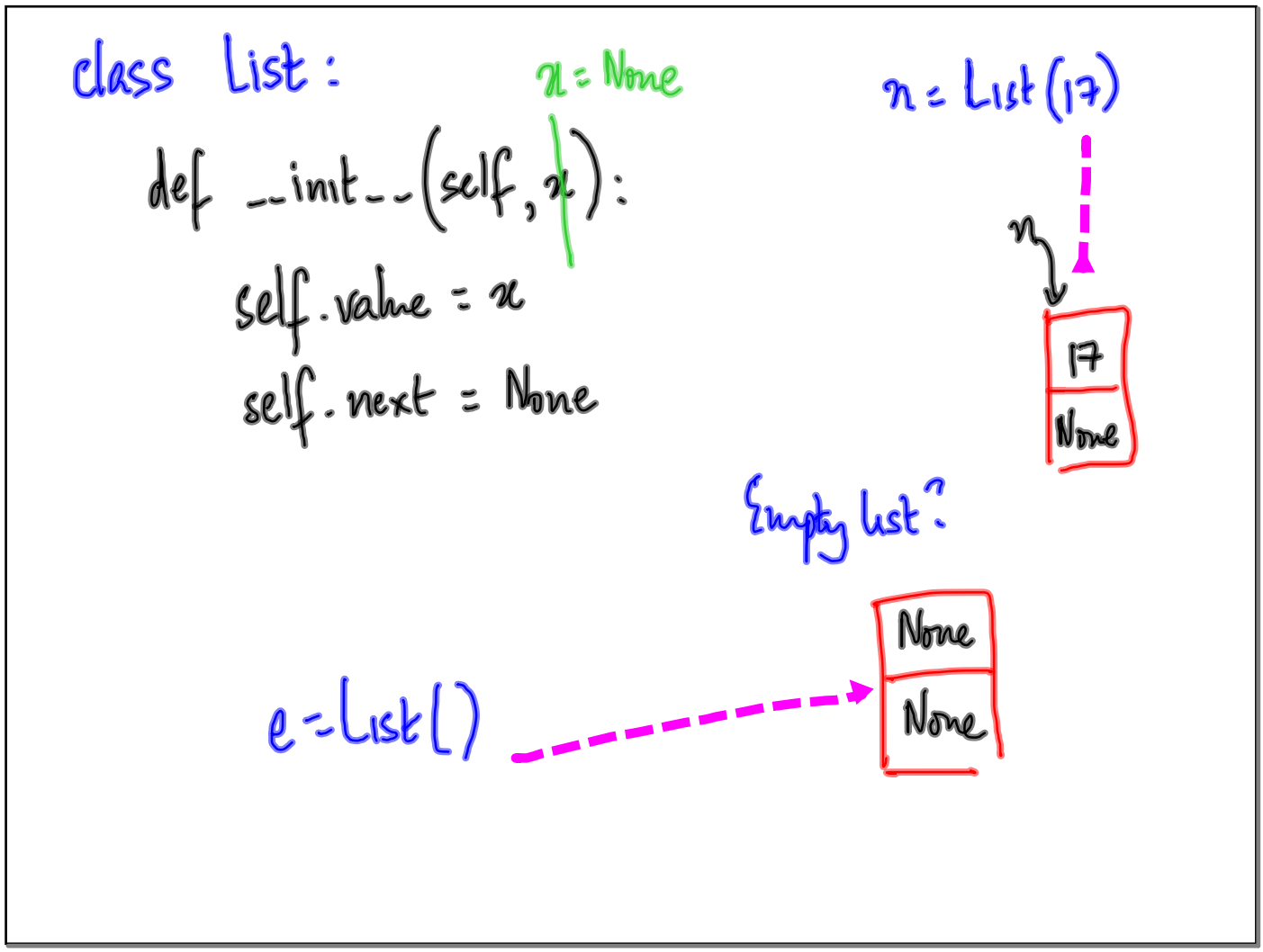
An appropriate Python class



fundamental unit




v0
next
v1
next
v2
None
x
x.value
x.next
can check
if x == None:


class List:
def __init__(self, x):
x = None
self.value = x
self.next = None
n = List(17)
n
17
None
Empty list:
e = List()
None
None

Growing a list: append

```
class List:
    def __init__(self, x=None):
        self.value = x
        self.next = None

    def append(self, x):
        if empty
            set value to x
        else
            find last node, create new node with value x,
                connect last node to new node
```

```
def append(self,x):
    # if self.value == None
    if self.isempty():
        self.value = x
        return
    #Find end of list
    tmp = self
    while tmp.next != None:
        tmp = tmp.next
    # Create new node and make tmp point to it
    newnode = list(x) ; tmp.next = newnode ; return
```

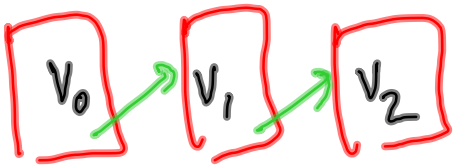
```
def isempty(self):
    if self.value == None
        return(True)
    else:
        return(False)
```

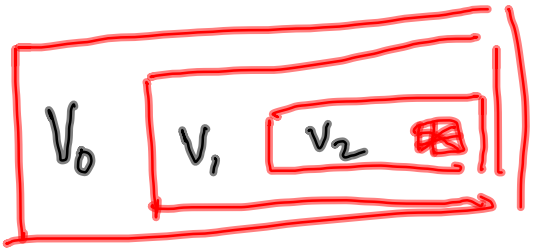
```
def __str__(self):
    mylist = []
    if self.isempty():
        return (str(mylist))

    tmp = self
    mylist.append(tmp.value)
    while tmp.next != None:
        tmp = tmp.next
        mylist.append(tmp.value)
    return (str(mylist))
```

Our version of append explicitly walks down the list

List is an example of a recursive data structure

List → empty
↳ value + List



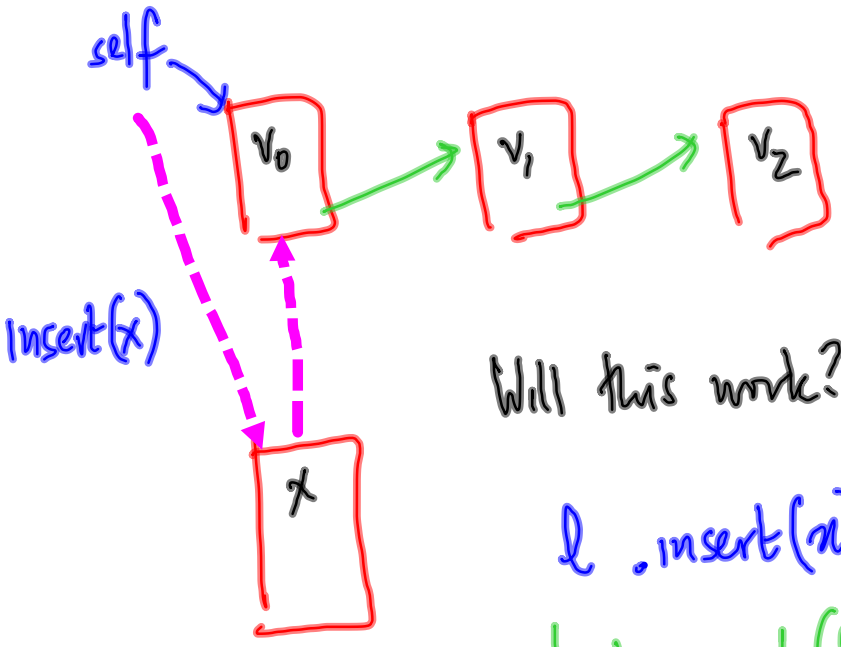


Sometimes more elegant to exploit this recursive structure

```
append
    if empty
        set value = x

    if this is the last node
        add a new node here
    else
        append into my inner list
```

```
def appendr(self, x):
    if self.isempty():
        self.value = x
        return

    if self.next == None:
        n = List(x)
        self.next = n
    else:
        self.next.append(x)
    return
```

How about inserting a value at the beginning?

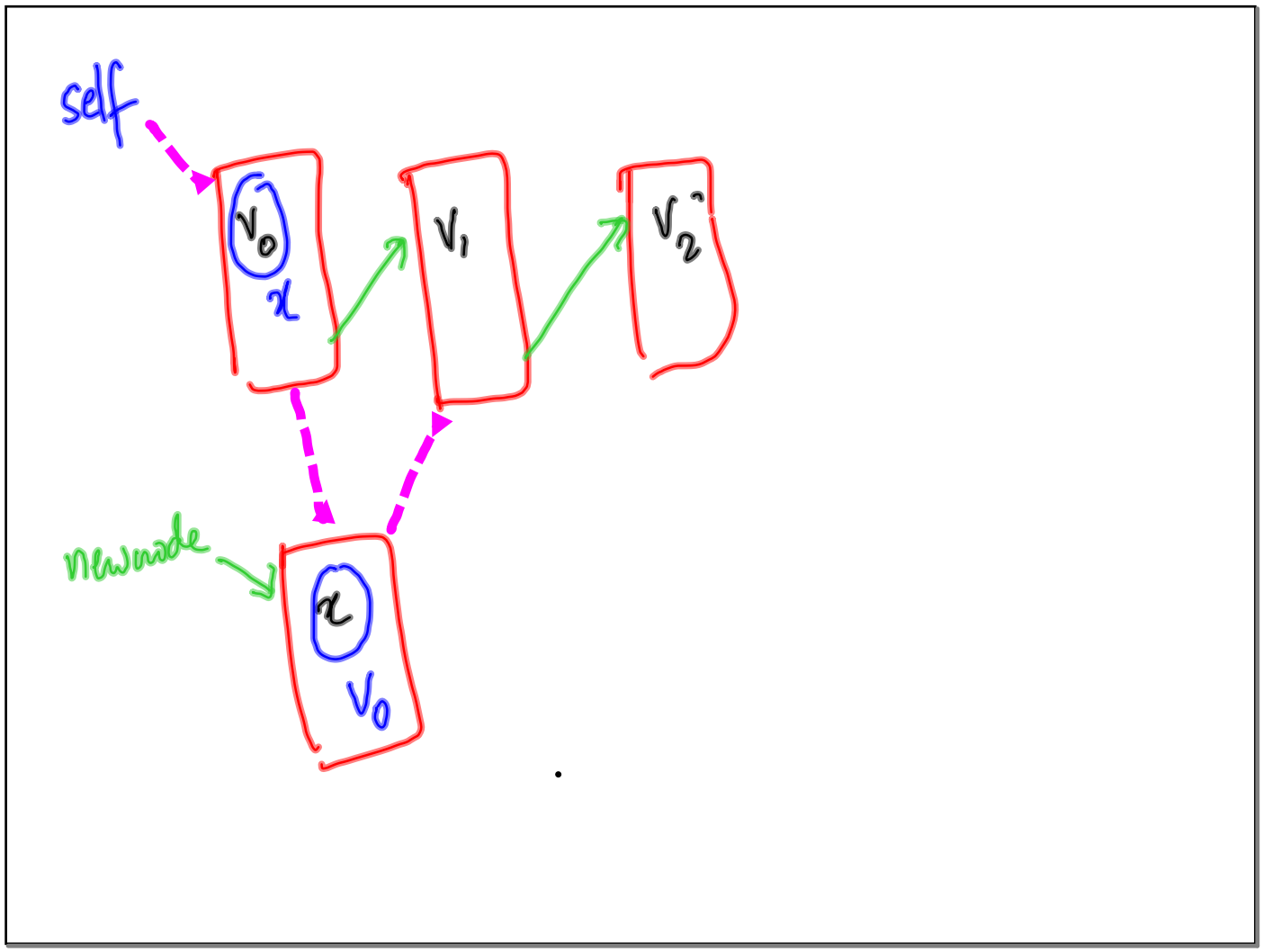


Will this work? No!

l.insert(x) → is self inside

List.insert(l, x)

Cannot make l point to a new object


self
v_0
x
v_1
v_2
newnode
x
v_0

```
def insert(self,x):
    if self.isempty():
        self.value = x
        return
    newnode = list(x)
    (self.value, newnode.value) = (newnode.value, self.value)
    (self.next, newnode.next) = (newnode, self.next)
```