Computational Thinking and Algorithms
159.172
Classes and Objects
An Object Orietned Perspective

Amjed Tahir

a.tahir@massey.ac.nz

Previous contributors: Catherine McCartin and Giovanni Moretti

From Functions to Classes

Up to know you have learned about structural code

Writing separate lines of code or as Functions

```
print ("my program")
  x = 10
  for i in range (x):
     print(i)
```

```
def function(x):
   for i in range (x):
      print(i)

x = 10
function(x)
```

Objects and classes

a class representing a character in a game:

```
class Character():
  name = "Link"
  sex = "Male"
  max_hit_points = 50
  current_hit_points = 50
  max_speed = 10
  armor_amount = 8
```

Objects and classes

class name

```
class Character():
  name = "Link"
  sex = "Male"
  max_hit_points = 50
  current_hit_points = 50
  max_speed = 10
  armor_amount = 8
```

attribute

Objects and classes

Define an address class

```
class Address():
    name = ""
    line1 = ""
    line2 = ""
    city = ""
    state = ""
    zip = ""
```

Objects and classes

Create an object, an instance of the address class

```
# Create an address
homeAddress = Address()

# Set the fields in the address
homeAddress.name = "John Smith"
homeAddress.line1 = "701 N. C Street"
homeAddress.line2 = "Carver Science Building"
homeAddress.city =  "Indianola"
homeAddress.state = "IA"
homeAddress.zip = "50125"
```

Objects and classes

Create another object, another instance of the address class

```
# Create another address
holidayhomeAddress = Address()

# Set the fields in the address
holidayhomeAddress.name = "John Smith"
holidayhomeAddress.line1 = "122 Main Street"
holidayhomeAddress.line2 = ""
holidayhomeAddress.city =  "Miami"
holidayhomeAddress.state = "FL"
holidayhomeAddress.zip = "50125"
```

Objects and classes

```
# Create an address

my_address = Address()

# Alert! This does not set the address's name!
name = "John"

# This doesn't set the name for the address either
Address.name = "John"

# This does work:
my_address.name = "John"
```

You must specify the object to set its attributes

Attributes & Methods

Attributes

- these are stored values
- variables that are local to the class
- something the object "has"

Methods

- functions that modify/use the objects data
- something the object can "do"

→ attributes are like nouns
→ Methods are often like verbs

Methods and Classes

```
class Dog():
    age = 0
    name = ""
    weight = 0

    def bark(self):
        print("Woof")
```

Methods and Classes

```
class Dog():
    age = 0
    name = ""
    weight = 0

    def bark(self):
        print("Woof")
```

first parameter of the method is the object itself

Methods and Classes

```
myDog = Dog()

myDog.name = "Spot”
myDog.weight = 20
myDog.age = 3

myDog.bark()
```

Methods and Classes

```
myDog.bark()
```

first parameter is assumed to be a reference to the dog object itself

behind the scenes, Python makes a call that looks like:

```
# Example, not actually legal
Dog.bark(myDog)
```

Methods and Classes

```
def bark(self):
    print( "Woof says", self.name )
```

Use self to refer to the object itself
and any of its attributes that we need to access

Abstract data types

an Abstract Data Type (ADT) specifies:

a set of operations

+

semantics of the operations (what they do)

does NOT specify the **implementation** of the operations.

want to separate properties of a data type (values and operations) from implementation of that data type.

Abstract data types

client code = code that uses the ADT

provider code = code that implements the ADT

client code interacts with instances of an ADT by invoking one of the operations defined by its **interface**.

set of operations has four categories:

1. constructors : create and initialize new instances of the ADT
2. accessors : return data contained in ADT instance without modifying it
3. mutators : modify the contents of an ADT instance
4. iterators : process data components of ADT instance sequentially.

Polymorphism

same operation works on objects from **different** classes

polymorphic = having many forms

```
>>> 2 + 3
5
>>> 'my’ + 'string'’
mystring’
>>> [1, 2, 3] + ['a', 'b', 'c']
[1, 2, 3, 'a', 'b', 'c']
```

arguments can be anything that supports addition

Polymorphism

same operation works on objects from **different** classes
polymorphic = having many forms

```
def length_message(x):
       print("The length of", repr(x), "is", len(x))

length_message('string') prints
The length ???? is ????

length_message([1,2,3,4]) prints
The length of ???? is ????

length_message([(1,2), (5,6), (8,9,10)]) prints
The length of ????  is ????
```

Polymorphism

same operation works on objects from **different** classes

polymorphic = having many forms

```
def length_message(x):
       print("The length of", repr(x), "is", len(x))

length_message('string') prints
The length of 'string' is 6

length_message([1,2,3,4]) prints
The length of [1, 2, 3, 4] is 4

length_message([(1,2), (5,6), (8,9,10)]) prints
The length of [(1, 2), (5, 6), (8, 9, 10)] is 3
```

Encapsulation

objects may hide (**encapsulate**) their internal state.
Let you use an object without knowing how it's constructed

```
class Person():
      __surname = 'Allen'

      def setName(self, name):
            self.name = name

      def getName(self):
            return self.name

      def __secretmessage(self):
            print('I can’t tell my name, it is ' + self.name + ' ' + self.__surname)

      def _semi_secret(self):
           print 'I have not told you my name.'

      def public_message(self):
          print ('The secret message is: ')
          self.__secretmessage()
```

Encapsulation

Objects may hide (**encapsulate**) their internal state.

- enables you to use an object without knowing how it's constructed

```
class Person():
    __surname = 'Allen'

    def setName(self, name):
        self.name = name

    def getName(self):
        return self.name

    def __secretmessage(self):
        print('I can’t tell my name, it is ' + self.name + ' ' +
self.__surname)
```

Encapsulation

```
class Person():
    __surname = 'Allen'

    def setName(self, name):
        self.name = name

    def getName(self):
        return self.name

    def __secretmessage(self):
        print('I can’t tell my name, it is ' + self.name + ' ' + self.__surname)
```

Private class variables should only be used by methods from that class

If a name starts with **two underscores,** that method or attribute is **private**

- it can only (easily) be accessed by methods in that class

Names starting with single underscore work normally but are **regarded as private**

Encapsulation

objects may hide (**encapsulate**) their internal state.
enables you to use an object without knowing how it's constructed

```
class Person():

      __surname = 'Allen'

      def _semi_secret(self):
           print 'I have not told you my name.'

      def public_message(self):
          print ('The secret message is: ')
          self.__secretmessage()
```

in Python, to make method or attribute private, start its name with two underscores
a single underscore to start the name means it should be *regarded* as private

Encapsulation

objects may hide (**encapsulate**) their internal state.
enables you to use an object without knowing how it's constructed

```
class Person():
      __surname = 'Allen'

      def _semi_secret(self):
           print('I have not told you my name.')

      def public_message(self):
          print('The secret message is: ' )
          self.__secretmessage()

x = Person()
x.setName('John')

x.public_message() prints

The secret message is:
I can’t tell my name, it is John Allen
```

Encapsulation

objects may hide (**encapsulate**) their internal state.
enables you to use an object without knowing how it's constructed

```
class Person():
      __surname = 'Allen'

      def getName(self):
         return self.name

      def getSurname(self):
         return self.__surname

>>> print(x.getName())
John
>>> print(x.getSurname())
Allen
>>> print(x.__surname())
AttributeError
```

Inheritance

inheritance allows us to build classes that are "specialisations" of other classes
a **subclass** inherits functionality from its **super** class

```
class Filter():
   def __init__(self):
       self.blocked = []

   def filter(self, sequence):
       return [x for x in sequence if x not in self.blocked]


class SPAMFilter(Filter): # SPAMFilter is a subclass of Filter
   def __init__(self):
       self.blocked = ['SPAM']
```

Inheritance

inheritance allows us to build classes that are "specialisations" of other classes

a **subclass** inherits functionality from its **super** class

```
>>> f = Filter()
>>> f.filter([1, 2, 3])
[1, 2, 3]
```

```
>>> s = SPAMFilter()
>>> s.filter( ['SPAM','SPAM','eggs','bacon', 'SPAM','SPAM']
['eggs', 'bacon']
```

Constructors

There's a special method called **the constructor**

- defines and initializes the data to be contained in the object
- it's automatically called when an object is created

```
class Point():
    def __init__(self, x, y):
        self.xCoord = x
        self.yCoord = y

>>> pointA = Point(5,7)
>>> pointB = Point(0,0)
```

Constructors

overriding a constructor - creating a constructor for a subclass

```
class Bird():
   def __init__(self):
       self.hungry = True

   def eat(self):
       if self.hungry:
          print('Yippee – food!')
       else:
         print('No thanks!')
```

Constructors

```
class SongBird(Bird):    # subclass (inherited class) of Bird
    def __init__(self):
        self.sound = 'Sqwark!'

    def sing(self):
        print(self.sound)

>>> sb = SongBird()

>>> sb.sing()
Squwark!

>>> sb.eat()   ??????
```

Constructors

```
class SongBird(Bird):    # subclass (inherited class) of Bird
    def __init__(self):
        self.sound = 'Sqwark!'

    def sing(self):
        print(self.sound)

>>> sb = SongBird()

>>> sb.sing()
Squwark!

>>> sb.eat()   ??????     THIS WON'T WORK!
```

Constructors

The parent class (Bird) hasn't been initialised
-- three ways to solve this problem.

```
class SongBird(Bird):
    def __init__(self):
        Bird.__init__(self)                     # Either this
        self.sound = 'Squwark!'

class SongBird(Bird):
    def __init__(self):
        super(SongBird, self).__init__()    # or this
        self.sound = 'Squwark!'

class SongBird(Bird):
    def __init__(self):
        super().__init__()    # <<< SIMPLEST ALTERNATIVE – RECOMMENDED!
        self.sound = 'Squwark!'
```

Special variables and Iterators

Python reserves some double underscore names as special

__init__ used to initialise objects
__len__ should return the length of something, if this makes sense
__repr__ should return a string representation of the object
Exactly what is up to you!

Iterators – classes that can be used in *for-loops* and with *in*

__iter__() Called to start an iteration – should return an Object that contains a __next__() method

__next__() Called to get the next item,
should raise **StopIteration** if no items remain

Iterator example – the Bag ADT

```
class Bag():
      def __init__(self, items = []):
            self. items = items

      def __iter__(self):
            self.currentItem = 0
            return self

      def __next__(self):
            if self.currentItem < len(self.items):
                  result = self.items[self.currentItem]
                  self.currentItem += 1
                  return result
            else:
                  raise StopIteration
```

Iterator example

an implementation of the Bag ADT

```
class Bag():
    def __init__(self):
        self.theitems = []

    def add(self, item):
        self.theitems.append(item)

    def remove(self, item):
        position = self.theitems.index(item)
        del self.theitems[position]

    def __iter__(self):
        return Bagiterator(self.theitems)
```

The Bag Iterator class

```
class Bagiterator:
    def __init__(self, thelist):     # GIVE IT ITEM LIST
        self.bagitems = thelist
        self.current = 0

    def __iter__(self):
        return self

    def next(self):
        if self.current < len(self.bagitems):
           item = self.bagitems[self.current]
           self.current += 1
           return item
        else:
           raise StopIteration
```

Subclassing and Inheritance

The idea: create an object BASED ON existing one

The original (the parent) has attributes & methods

- the child **inherits** all the parent's attributes & methods
- it can **add** new attributes and methods
- it can **override** attributes and methods of the parent

Why subclass?

- avoid duplicating code that already work
- subclasses can be added without modifying class source code

Subclassing Bag

The **Bag** Class can create a bag, display and iterate over it

Let's create a subclass that

- adds a **save()** method to write the bag to a file
- modifies the constructor to print the bag
- modifies **display()** so title is REQUIRED

Original **Bag** class (in *iterators.py*)

```
class Bag():
    def __init__(self, itemList):
        self.items = itemList

    def display(self, title = ""): # Title is optional
        print(title, self.items)

    def __iter__(self):     # support an iterator
        self.index = 0
        return self

    def __next__(self):
        if self.index < len(self.items):
            result = self.items[self.index]
            self.index +=1
            return result
        else:
            raise StopIteration
```

Extending **Bag()** via subclassing

```
import iterators

class Saveable_Bag(iterators.Bag):          #  Saveable_Bas is s subclass of Bag

    def __init__(self, contents):
        print('New Bag contains', contents)
        #  If subclass has a constructor, MUST call parent constructor
        super().__init__(contents)              # CALL BAG.__init__() to setup items

    def save(self, filename):
        with open(filename, 'w') as f:
            for item in self.items:
                f.write(item + '\n')

    def display(self, title):    # make title required
        super().display(title) # call display() from parent class
```

These work too including with Python 2.7

super(Saveable_bag, self).__init__(contents)

iterators.Bag.__init__(self, contents)

Fibonacci Sequence via Iterators

you can iterate over any object whose class definition implements the __iter__ method which returns an iterator

```
class Fibs():                    # returns 1, 2, 3, 5, 8, 11 ...
    def __init__(self):
        self.a = 0
        self.b = 1

    def __next__(self):
        self.a, self.b = self.b, self.a+self.b
        return self.a

    def __iter__(self):
        return self
```

Iterators

iterator for the rows of Pascal's triangle

```
class Pascal():
    def __init__(self):
        self.lastRow = []
        self.nextRow = [1]

    def next(self):
        self.lastRow = self.nextRow
        self.nextRow = [(a+b) for a,b in zip
                        ([0]+self.lastRow,self.lastRow+[0])]!
        return self.lastRow

    def __iter__(self):
        return self
```

ADTs

An **abstract data type (ADT)** “consists of” data together with functions that operate on the data.

“Abstract” in the sense that how the data is represented and how the functions are implemented is not specified.

Only the **behaviour** of the functions is specified, via an **interface**.

Stacks

- linear sequence of data items
- **insertions and deletions made at only one end** - stack “top”
- last-in, first-out (LIFO) data structure

Stack interface:

`__init__()`	initialize a new empty stack.
`push(new_item)`	add a new item to the stack.
`pop()`	remove and return an item (always the last one added)
`isEmpty()`	check whether the stack is empty.

Stacks

stack interface:

`__init__()`	initialize a new empty stack.
`push(new_item)`	add a new item to the stack.
`pop()`	remove and return an item (always the last one added)
`isEmpty()`	check whether the stack is empty.

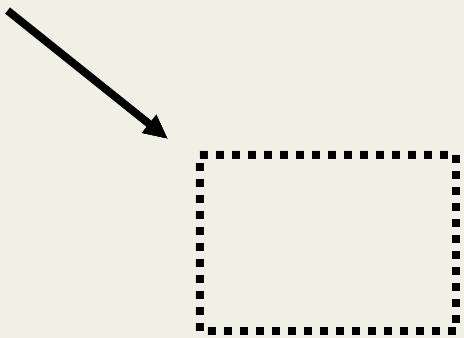
specifications define a stack but do not tell how to implement it

As long as the operations have the properties specified, the ADT is a stack

Stacks

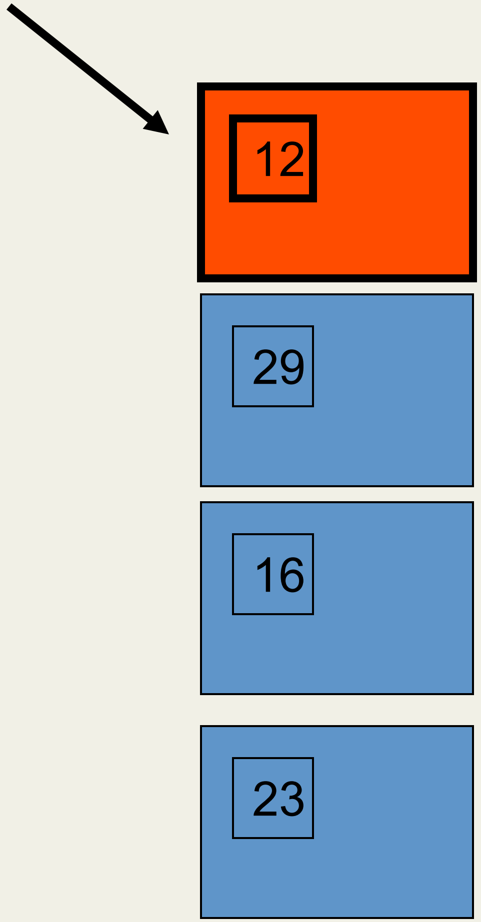
executing method **`s.__init__()`** makes **`s`** an empty stack
constructor called automatically

```
>>> s = Stack()
```



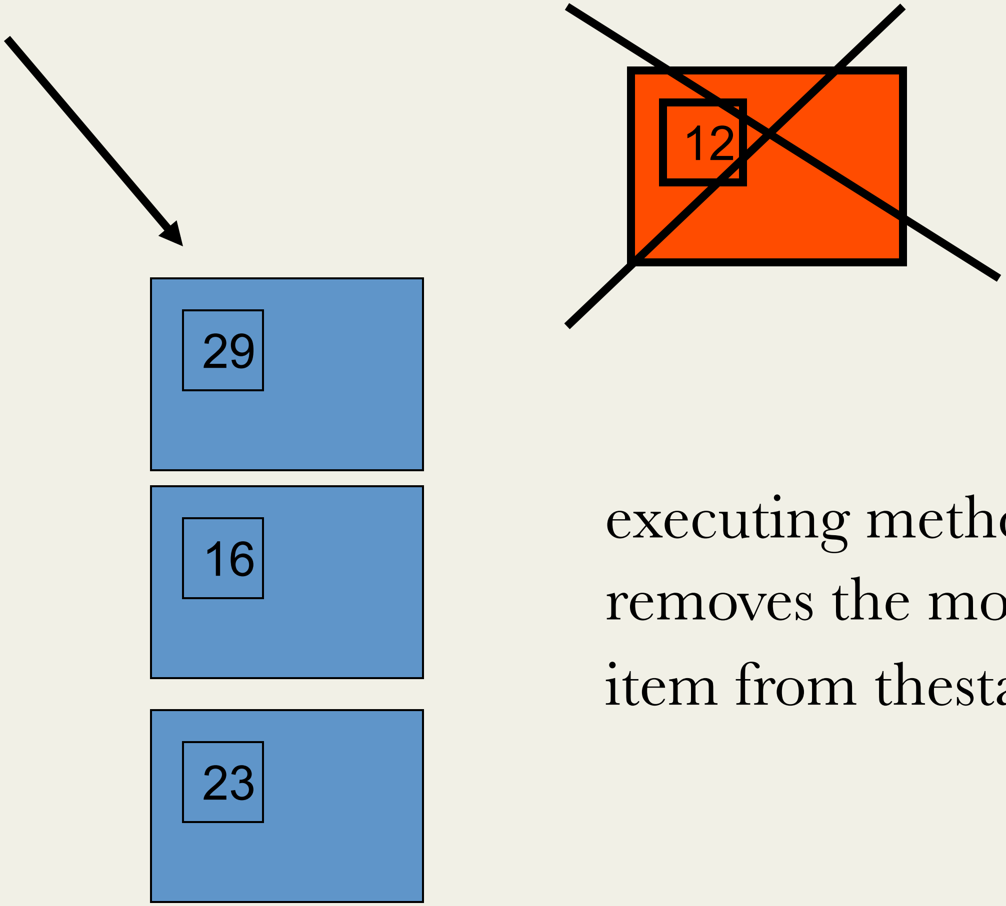
`s.isempty()` is true

Stacks



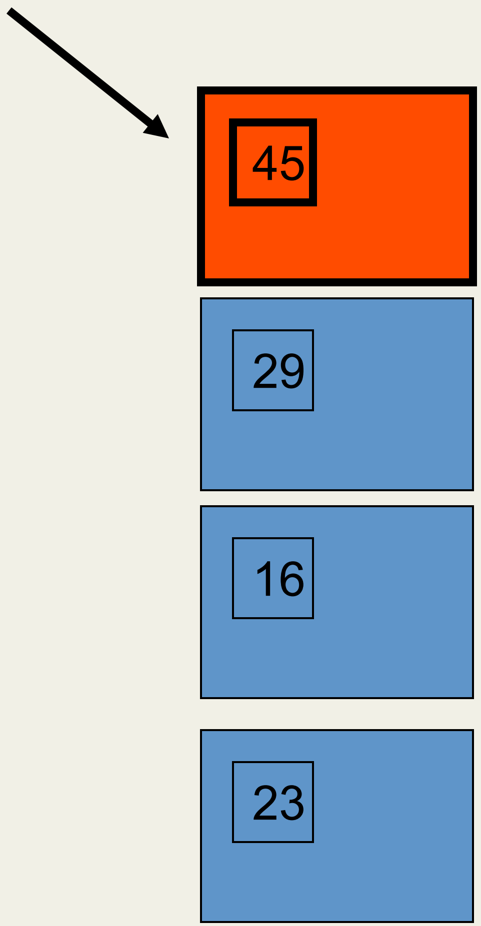
executing method **`s.push(12)`** adds 12 to the stack

Stacks



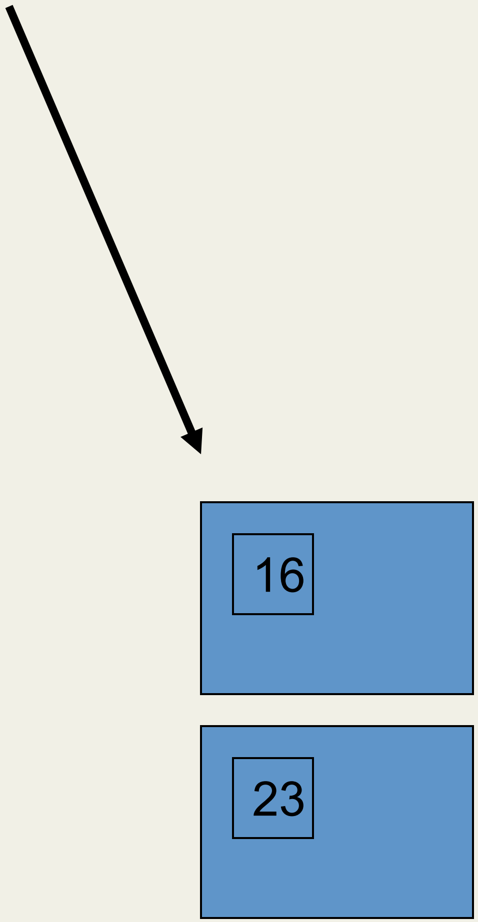
executing method **`s.pop()`** removes the most recently added item from thestack

Stacks



`s.push(45)` adds 45 to the stack

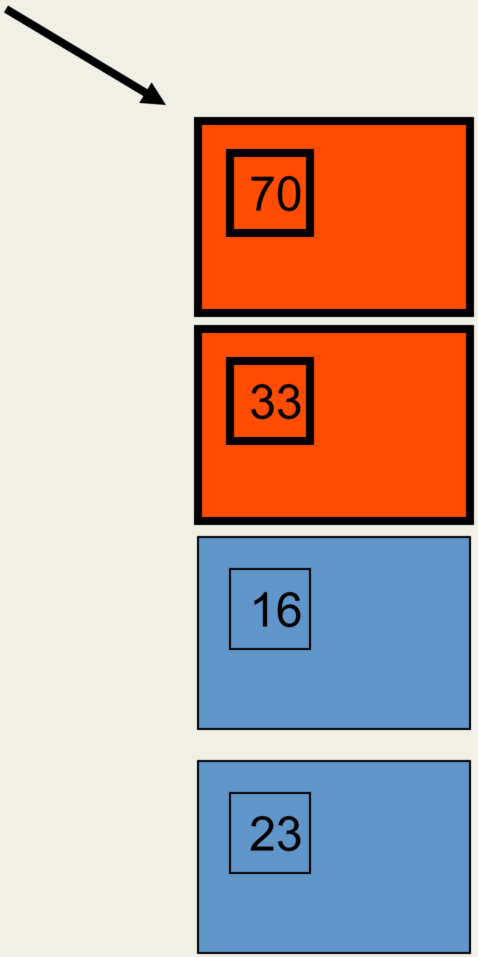
Stacks




```
>>> s.pop()
>>> s.pop()
```

removes last two most recently added items

Stacks




```
>>> s.push(33)
>>> s.push(70)
```

adds 33, then 70 to the stack

Stacks

Implementing the stack interface using a Python list:

```
class Stack :
  def __init__(self) :
      self.items = []

  def push(self, item) :
      self.items.append(item)

  def pop(self) :
      return self.items.pop()

  def isEmpty(self) :
      return (self.items == [])
```

Stacks

What could we add to the stack interface?

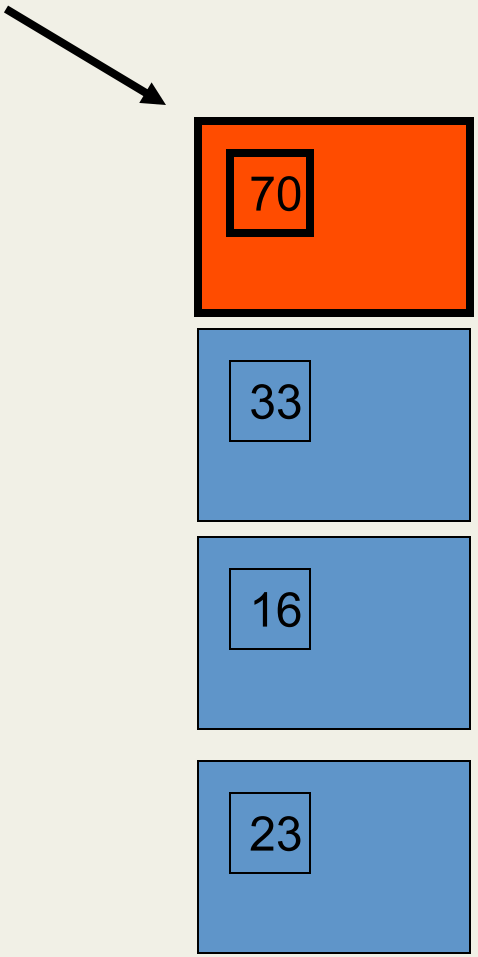
`top()` return item on top of stack without removing it.

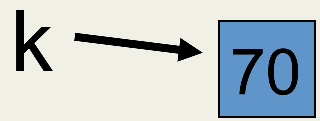
`peek()` return item directly below top of stack without removing it.

`pop_many(n)` remove and return an item top n items (as a tuple)

`push_many(seq)` push objects in seq onto stack (from left to right)

Stacks






```
>>> k = s.top()
```

does not remove 70 from the stack

Stacks

Adding to the stack interface?

```
class Stack:
  def top(self):
      return self.items[-1]

  def peek(self):
      return self.items[-2]

  def pop_many(self, n):
      ???

   def push_many(self, seq):
      ???

   def display(self):
      ???
```

Stacks and recursion

Compare these two functions. Try them out on a couple of strings.
The recursive function is implemented by storing activation records on a **run-time stack**.

```
def printbackwards(myString):
     if myString == '':
        return
    else:
        printbackwards(myString[1:])
        print(myString[0])

def stackdisplay(myString):
   s = Stack()
   for i in myString:
       s.push(i)
   while not s.isEmpty():
       print(s.pop())
```