北京航空航天大學
BEIHANG UNIVERSITY

Introduction to Computer Science and Programming

Lecture 5

Sebastian Wandelt

Beihang University

Outline

- Recap
- Compound datatypes
- Object-Oriented Programming

Recap

One task

Task 5 ***Recursion – Full permutation*** *(15 minutes)*

Write a function which recieves a list and print all permutation of the elements, i.e., all possible orders of elements.

```
# sample input:
permutation([1,2,3])
# sample output:
[1, 3, 2]
[1, 2, 3]
[3, 1, 2]
[3, 2, 1]
[2, 1, 3]
[2, 3, 1]
```

Yet another task

- Given a set with n elements, e.g. S=list([2,5,6,7,8]), print all possible x-tuples (with repetitions)!
- Expected output (for x =2):
  - 2,2
  - 2,5
  - 2,6
  - 2,7
  - 2,8
  - 5,2
  - 5,5
  - 5,6
  - 5,7
  - …
  - 8,8

First without recursion!

Yet another task

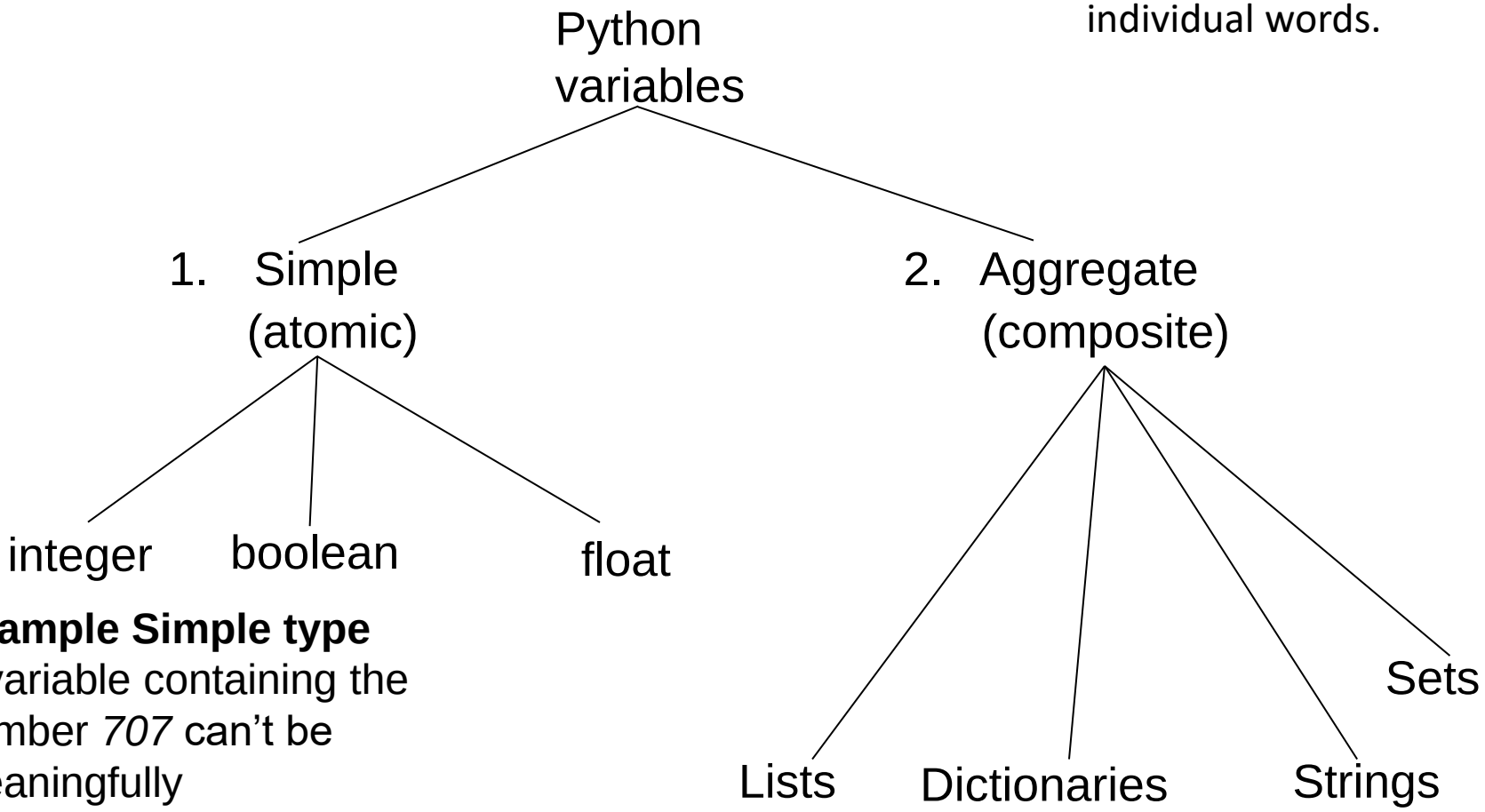
- Given a set with n elements, e.g. S=list([2,5,6,7,8]), print all possible x-tuples (with repetitions)!
- Expected output (for x =2):
  - 2,2
  - 2,5
  - 2,6
  - 2,7
  - 2,8
  - 5,2
  - 5,5
  - 5,6
  - 5,7
  - …
  - 8,8

Now with recursion!

Compound datatypes

Types Of Variables

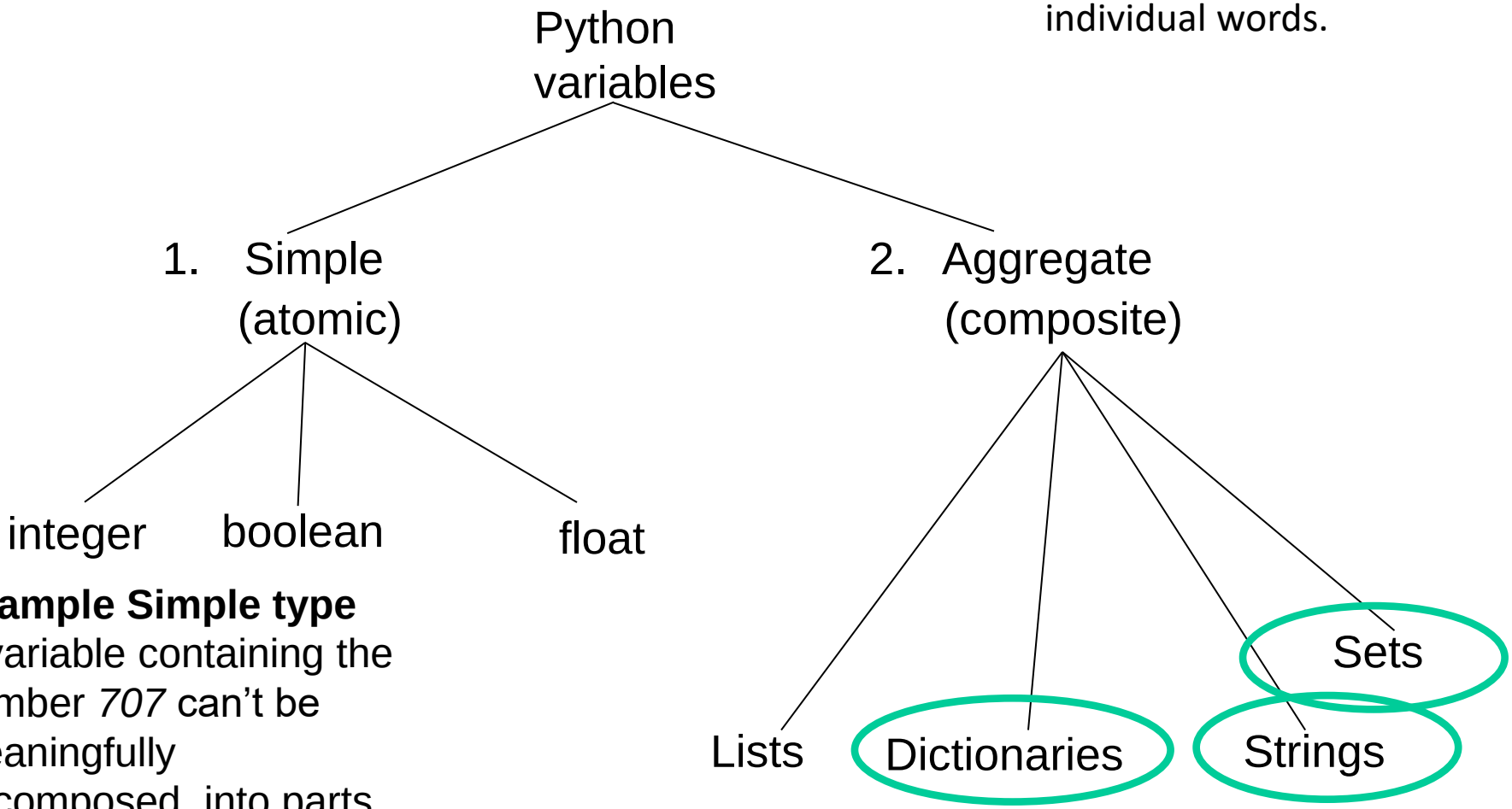
Example composite
A string (sequence of characters) can be decomposed into individual words.



Example Simple type
A variable containing the number *707* can't be meaningfully decomposed into parts

Types Of Variables

Example composite
A string (sequence of characters) can be decomposed into individual words.



Example Simple type
A variable containing the number *707* can't be meaningfully decomposed into parts

Compound datatypes
Strings

What are strings?

- Strings are a list of characters
- Almost all operations on lists also work on strings!
  - Iteration
  - Length
  - Index/Slicing
  - …

Compound datatypes
Dictionaries

What is a dictionary?

- You can think of it as a list of pairs, where the first element of the pair, the ***key***, is used to retrieve the second element, the ***value***.
- Thus we ***map a key to a value***
- Example:
  - Telephone book

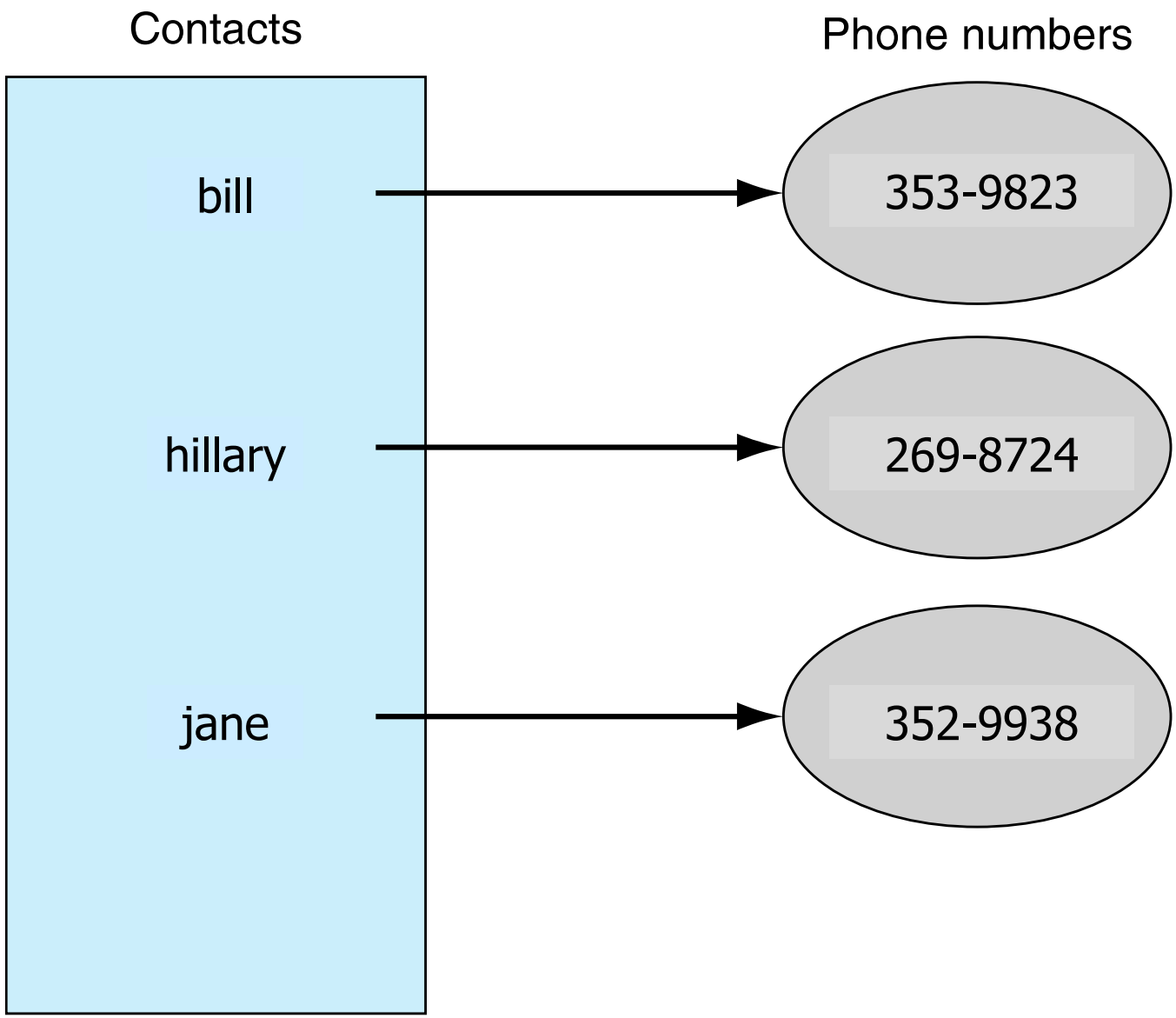
Key-Value pairs

- The key acts as an index to find the associated value.
- Just like a dictionary, you look up a word by its spelling to find the associated definition
- A dictionary can be searched to locate the value associated with a key

Python Dictionary

- Use the { } marker to create a dictionary
- Use the : marker to indicate key:value pairs

```
contacts= {'bill':'353-9823',
           'hillary':'269-8724',
          'jane':'352-9938'}
```


Contacts
Phone numbers
bill
353-9823
hillary
269-8724
jane
352-9938

Keys and values

- Key must be immutable
  - strings, integers, tuples are fine
  - lists are NOT
- Value can be anything

Collections but not a sequence

- Dictionaries are collections but they are not sequences such as lists, strings or tuples
  - There is no order to the elements of a dictionary
  - In fact, the order (for example, when printed) might change as elements are added or deleted.
- So how to access dictionary elements?

Access dictionary elements

Access requires `[ ]`, but the *key* is the index!

`my_dict={}`

- an empty dictionary

`my_dict['bill']=25`

- added the pair 'bill':25

`print(my_dict['bill'])`

- prints `25`

Dictionaries are mutable

- Like lists, dictionaries are a mutable data structure
  - you can change the object via various operations, such as index assignment

```
my_dict = {'bill':3,'hillary':10}
print(my_dict['bill’])
my_dict['bill'] = 100
print(my_dict['bill'])
```

Common operators with Lists

Like others, dictionaries respond to these

- `len(my_dict)`
  - number of key:value **pairs** in the dictionary
- `element in my_dict`
  - boolean, is `element` a **key** in the dictionary
- `for key in my_dict:`
  - iterates through the **keys** of a dictionary

Dictionary content methods

- `my_dict.items()` – List of all the key/value pairs
- `my_dict.keys()` – List of all the keys
- `my_dict.values()` – List of all the values

How to print all values in a dictionary, each value in a separate line?

Example: Word frequencies

We are given a list of words

word_list=[‘she’,’dog’, ’cat’,’hello’,’he’, ’she’,’good’,’dog’]

and want to compute the word frequencies using a dictionary

Example: Word frequencies

```
count_dict = {}
for word in word_list:
    if word in count_dict:
        count_dict[word] += 1
    else:
        count_dict[word] = 1
```

Objects and classes

Why should you use functions in your code?

Why should you use functions in your code?

- Functions avoid repetitive code
- Functions provide a fixed interface
  - User does not need to know implementation
  - Better reusability
- Functions can be grouped into context
  - Packages in Python: math, numpy, …
  - Leads to much cleaner code!

What is the problem with functions?

What is the problem with functions?

- Functions need to get all information via parameters
  - Difficult/Hard to manage for complex objects
  - (We do not consider the use of global variables here)
- The parameters depend on the implementation
  - Changing the function implementation will also change the parameters, at least for complex code
- If the user has access to the parameters/implementation details, he can mess up (=break) the whole implementation

Large pieces of reusable software need new software-design techniques in order to be used in complex environments!

Object-oriented programming (OOP)

Object-Oriented Programming

- You have learned structured programming
  - Breaking tasks into subtasks
  - Writing re-usable methods to handle tasks
- We will now study Objects and Classes
  - To build larger and more complex programs
  - To model objects we use in the world

A class describes objects with the same behavior. For example, a Car class describes all passenger vehicles that have a certain capacity and shape.

Objects and Programs

- You have already experienced this programming style when you used strings and lists. Each of these objects has a set of methods.
- For example, you can use the `append()` or `len()` methods to operate on list objects
  - L=[1,2,3,4]
  - L.append(5)

What is going on there? You will learn that today!

Public Interfaces

- The set of all methods provided by a class, together with a description of their behavior, is called the public interface of the class.
- When you work with an object of a class, you do not know how the object stores its data, or how the methods are implemented.
  - **You do not need to know how a `list` stores its elements.**
- All you need to know is the public interface—which methods you can apply, and what these methods do.

Objects and Programs

5.1. More on Lists

- The list data type has some more methods. Here are all of the methods of list objects:

`list.`**`append`**`(x)`
Add an item to the end of the list; equivalent to `a[len(a):] = [x]`.

`list.`**`extend`**`(L)`
- Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`.

`list.`**`insert`**`(i, x)`
Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.`**`remove`**`(x)`
Remove the first item from the list whose value is *x*. It is an error if there is no such item.

`list.`**`pop`**`([i])`
Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

`list.`**`index`**`(x)`
Return the index in the list of the first item whose value is *x*. It is an error if there is no such item.

`list.`**`count`**`(x)`
Return the number of times *x* appears in the list.

https://docs.python.org/3/tutorial/datastructures.html

Public Interfaces

- The process of providing a public interface, while hiding the implementation details, is called **encapsulation**.
- If you work on a program that is being developed over a long period of time, it is common for implementation details to change, usually to make objects more efficient or more capable.
- When the implementation is hidden, the improvements do not affect the programmers who use the objects.

OOP by example I

How would you implement a counter in Python?

- Tally Counter: A mechanical device that is used to count people
  - For example, to find out how many people attend a concert or board a bus
- Inteface: What should it do?
  - Increment the tally
  - Get the current total

Tally Counter with functions I

```
def increment():
    global value
    value=value+1

def show():
    global value
    print(value)

value=0
increment()
increment()
increment()
show()
increment()
increment()
show()
```

Global variables are BAD.
Try to NEVER use them!

Tally Counter with functions II

```
def increment(value):
    return value+1

def show(value):
    print(value)

value=0
value=increment(value)
value=increment(value)
value=increment(value)
show(value)
value=increment(value)
value=increment(value)
show(value)
```

We have to pass the data around! Not good.

Tally Counter with functions ...

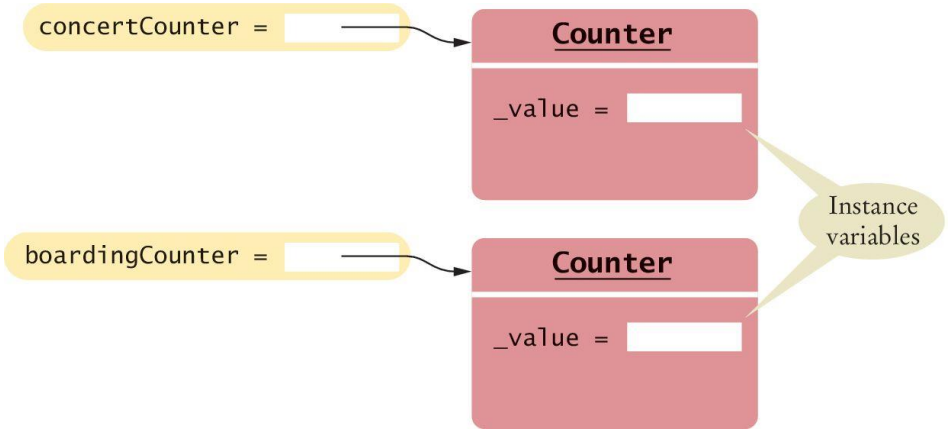
- We want to have a kind of data structure which allows us to interact with the counter in a predefined way
  - Without global variables
  - Without passing data around
  - Actually, without caring what the instance data is like at all

Tally Counter in OOP style

```
class Counter:
    def __init__(self):
        self.value=0

    def increment(self):
        self.value=self.value+1

    def show(self):
        print(self.value)

c=Counter()
c.increment()
c.increment()
c.increment()
c.show()
c.increment()
c.increment()
c.show()
```

Instance Variables

- An object stores its data in **instance variables**.
- An *instance* of a class is an object of the class.
- In our example, each `Counter` object has a single instance variable named `value`.
  - For example, if `concertCounter` and `boardingCounter` are two objects of the `Counter` class, then each object has its own `value` variable



Instance Variables

- Instance variables are part of the implementation details that should be hidden from the user of the class.

Class Methods

- The methods provided by the class are defined in the class body.
- The `increment()` method advances the `value` instance variable by 1.

```
def increment(self) :
    self.value = self.value + 1
```

- A method definition is very similar to a function with these exceptions:
  - A method is defined as part of a class definition.
  - The first parameter variable of a method is called `self`.

Class Methods and Attributes

- Note how the `increment()` method increments the instance variable `value`.
- *Which* instance variable? The one belonging to the object on which the method is invoked.
  - In the example below the call to `increment()` advances the `value variable` of the `concertCounter` object.
  - No argument was provided when the `increment()` method was called *even though the definition includes the `self` parameter*

```
concertCounter.increment()
```

  - The `self` parameter variable refers to the object on which the method was invoked, `concertCounter` in this example.

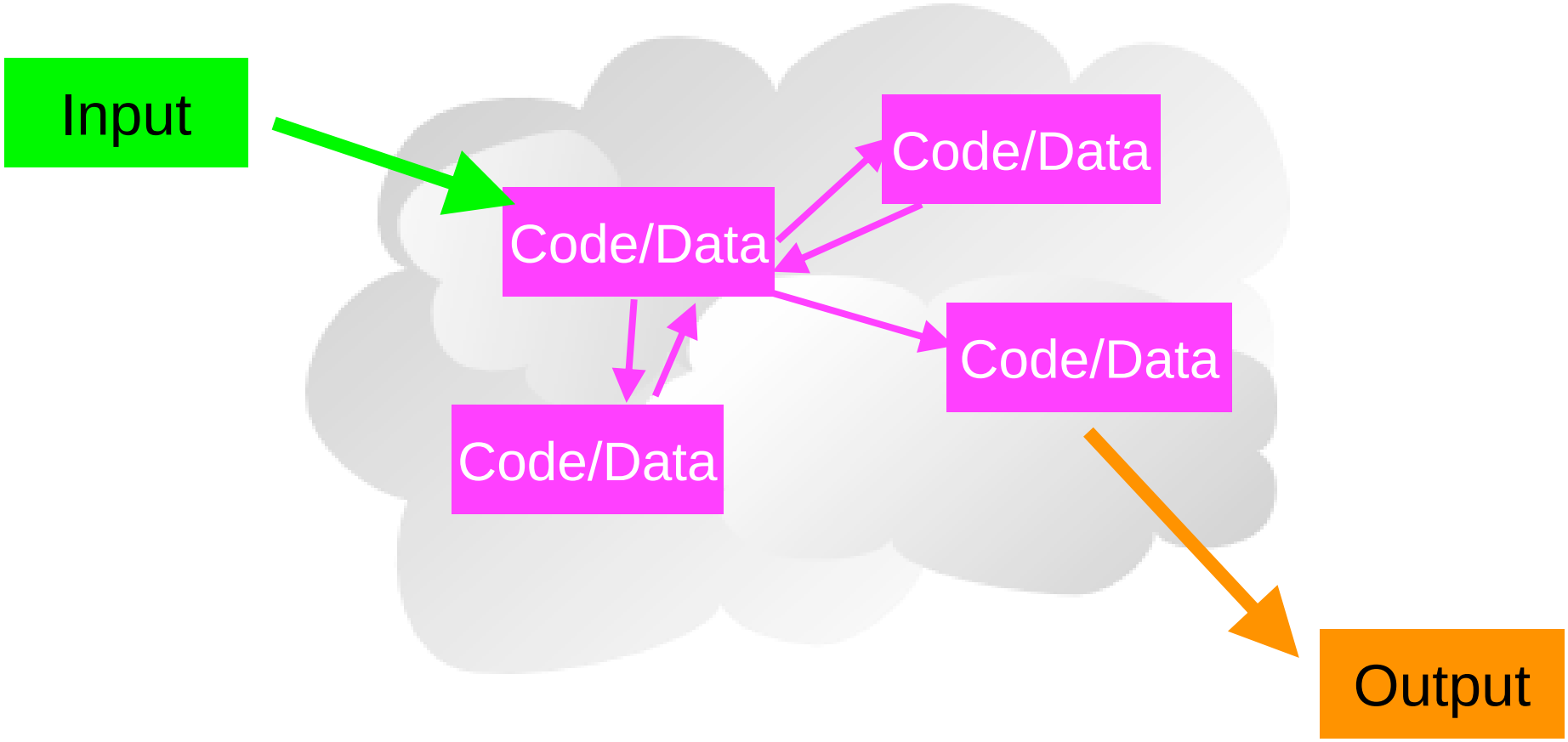
Example of Encapsulation

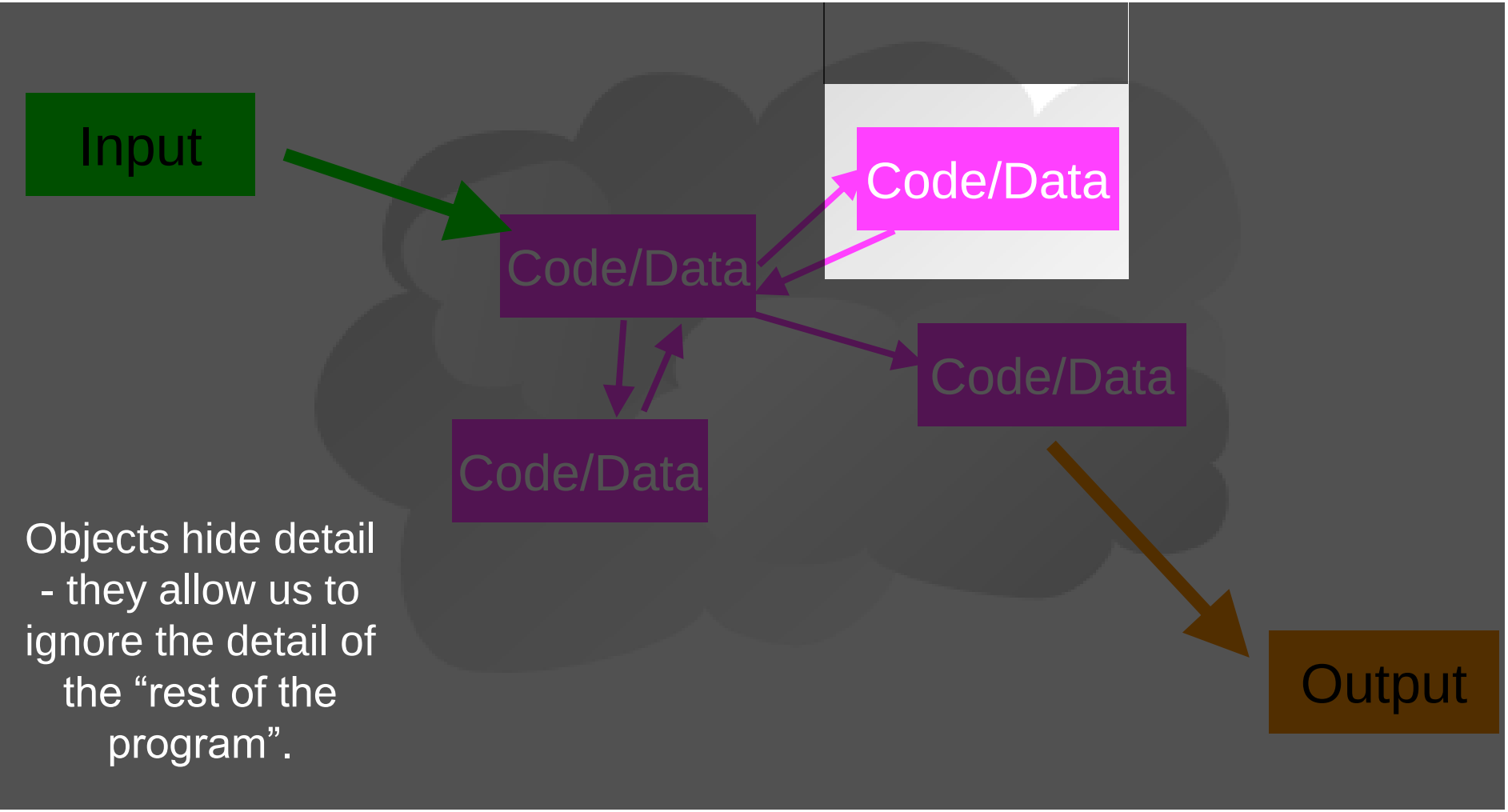
- The `show()` method prints the current `value`:

```
def show(self) :
    return self.value
```

- This method is provided so that users of the `Counter` class can find out how many times a particular counter has been clicked.
- A class user should **not** directly access any instance variables. Restricting access to instance variables is an essential part of encapsulation.

A new view on “Algorithms”





Objects hide detail - they allow us to ignore the detail of the “rest of the program”.

OOP by example II

A Cash register

- A customer buys some items
- We want to know how much he needs to pay!

Public Interface of a Class

- When you design a class, start by specifying the public interface of the new class
  - What tasks will this class perform?
  - What methods will you need?
  - What parameters will the methods need to receive?

- Example: A Cash Register Class

Task	Method
Add the price of an item	`addItem(price)`
Get the total amount owed	`getTotal()`
Get the count of items purchased	`getCount()`
Clear the cash register for a new sale	`clear()`

  - Since the '`self`' parameter is required for all methods it was excluded for simplicity.

Writing the Public Interface

```
## A simulated cash register that tracks the item count and the total amount
due.
#
class CashRegister :
  ## Adds an item to this cash register.
  # @param price: the price of this item
  #
  def addItem(self, price) :
    # Method body

  ##  Gets the price of all items in the current sale.
  #  @return the total price
  #
  def getTotal(self):  ...
```

Class comments document the class and the behavior of each method

The method declarations make up the *public interface* of the class

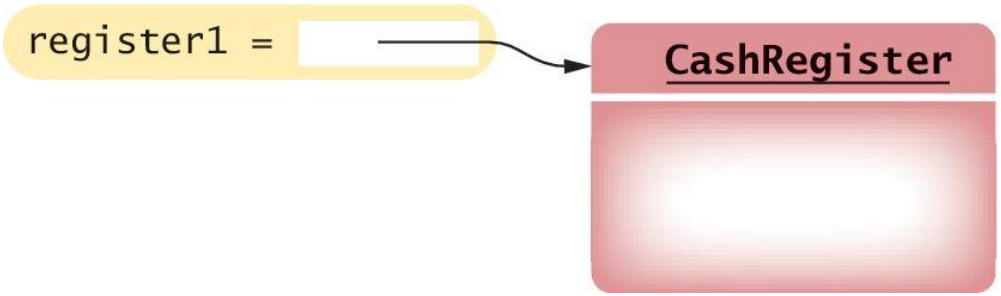
The data and method bodies make up the *private implementation* of the class

Using the Class

- After defining the class we can now construct an object:

```
register1 = CashRegister()
   # Constructs a CashRegister object
```

- This statement defines the `register1` variable and initializes it with a reference to a new `CashRegister` object



Using Methods

- Now that an object has been constructed, we are ready to invoke a method:

```
register1.addItem(1.95) # Invokes a method.
```

Accessor and Mutator Methods

- Many methods fall into two categories:
  1) Accessor Methods: '**get**' methods
     - Asks the object for information without changing it
     - Normally returns the current value of an attribute

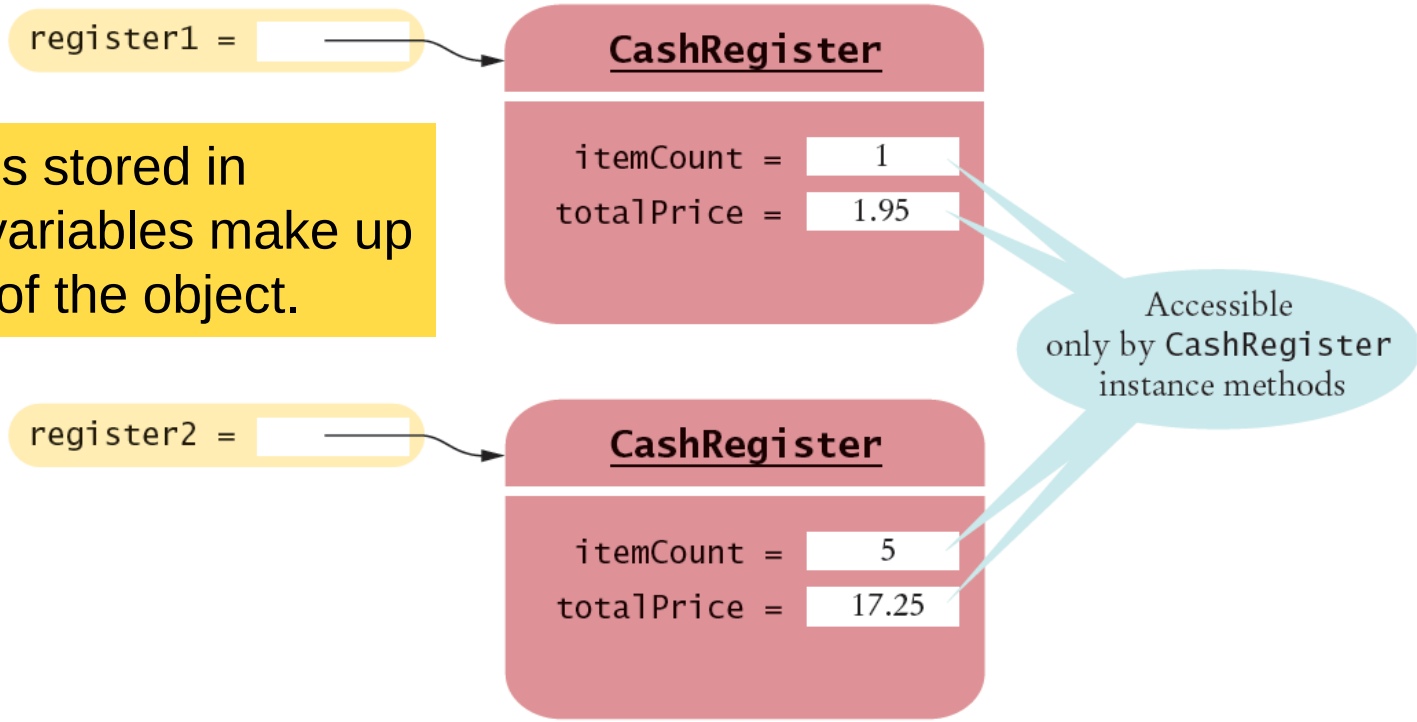
```
def getTotal(self):
def getCount(self):
```

  2) Mutator Methods: '**set**' methods
     - Changes values in the object
     - Usually take a parameter that will change an instance variable

```
def addItem(self, price):
def clear(self):
```

Instance Variables of Objects

- Each object of a class has a separate set of instance variables.



The values stored in instance variables make up the **state** of the object.

The code

```
class CashRegister:
    def __init__(self):
        self.itemCount=0
        self.totalPrice=0

    def addItem(self, price):
        self.itemCount=self.itemCount+1
        self.totalPrice=self.totalPrice+price

    def getTotal(self):
        return self.totalPrice

    def getCount(self):
        return self.itemCount

    def clear(self):
        self.itemCount=0
        self.totalPrice=0

register1=CashRegister()
register1.addItem(5.4)
register1.addItem(15.4)
print(register1.getTotal())
```

The code II (different implementation, same interface)

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

    def addItem(self, price):
        self.items.append(price)

    def getTotal(self):
        return sum(self.items)

    def getCount(self):
        return len(self.items)

    def clear(self):
        self.items=[]

register1=CashRegister()
register1.addItem(5.4)
register1.addItem(15.4)
print(register1.getTotal())
```

Comparison

```
class CashRegister:
    def __init__(self):
        self.itemCount=0
        self.totalPrice=0

    def addItem(self, price):
        self.itemCount=self.itemCount+1
        self.totalPrice=self.totalPrice+price

    def getTotal(self):
        return self.totalPrice

    def getCount(self):
        return self.itemCount

    def clear(self):
        self.itemCount=0
        self.totalPrice=0

register1=CashRegister()
register1.addItem(5.4)
register1.addItem(15.4)
print(register1.getTotal())
```

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

    def addItem(self, price):
        self.items.append(price)

    def getTotal(self):
        return sum(self.items)

    def getCount(self):
        return len(self.items)

    def clear(self):
        self.items=[]

register1=CashRegister()
register1.addItem(5.4)
register1.addItem(15.4)
print(register1.getTotal())
```

Did you understand OOP?

What is the output?


```
class Animal:
    def setSpecies(self, species):
        self.spec = species

    def setSound(self, language):
        self.lang = language

    def makeSound(self):
        print("I am a",self.spec,"and I",self.lang)


snoopy=Animal()
snoopy.setSpecies('dog')
snoopy.setSound('bark')
snoopy.makeSound()

garfield=Animal()
garfield.setSpecies('cat')
garfield.setSound('meow')
garfield.makeSound()
```

How about this one?


```
class Animal:
    def setSpecies(self, species):
        self.spec = species

    def setSound(self, language):
        self.lang = language

    def makeSound(self):
        print("I am a",self.spec,"and I",self.lang)


snoopy=Animal()
garfield=Animal()
snoopy.setSpecies('dog')
garfield.setSound('meow')
garfield.setSpecies('cat')
snoopy.setSound('bark')
snoopy.makeSound()
garfield.makeSound()
```

Thank you very much!

If you have any questions,
please get in touch with me:
wandelt@buaa.edu.cn