Classes as data structures with methods

TEAM INFDEV

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Introduction

Classes as data structures with methods

TEAM INFDEV

Introduction

Problem discussion

General idea

Technical details

A few advanced design topics

Conclusion

Lecture topics

- In this lecture we will "close the circle" of data structures and functions
- We will show how it is possible to form a new, even more powerful abstraction
- We will define **classes** as the joining of functions and data structures

Problem discussion

Problem discussion

Introduction

- Functions on lists always take a list as input
- This list is understood as the main **subject** of the computation
- We would like to create a stronger visual and semantic link between the subject of a computation and the computation itself

Problem discussion

Examples

- `length` of a list
- `sum` of a list
- `find` in a list a given element
- `map` of a list wrt some transformation
- ...

General idea

General idea

Introduction

- To solve the problem, we will create `classes`
- A `class` is the union of a data structures and its characterizing functions
- The functions inside a class are known as `methods`

General idea

Blueprint of a class

- What are the fundamental attributes?
- What properties do the attributes have (relationships, etc.)?
- What are the fundamental methods of the class?
- What properties do the attributes have (relationships, returned values, etc.)?

The blueprint of the list class (**THIS IS NOT CODE!**)


```
Abstraction List =
  the list may be Empty, with
    no attributes
    IsEmpty() method returns True
    Length method returns 0
    Map(f) method returns the empty list
    Filter(p) method returns the empty list
    ...

  the list may be a Node
    head attribute (the value of the element of this node)
    tail attribute (the rest of the list)
    IsEmpty() method returns False
    Length method returns the length of the list
    Map(f) method returns the the transformed list wrt function f
    Filter(p) method returns the list with only elements respecting p
    ...
```

General idea

Design of a class

- The hardest part of building a class is its design
- How do we build a reasonable class?
- What is a bad implementation?

General idea

Design of a class

- The same logic that applies to the design of functions applies to classes
- **Encapsulation**[a] is the central property of both
- A class is well encapsulated if
  - it offers a clear interaction surface by not exposing its internals in a dangerous way
  - it is a cohesive unit that offers a single, clearly defined set of related services

[a] Also called **information hiding**

General idea

Design of a class

- Thanks to **encapsulation**, a program can be built as a series of independent units
- These units are **loosely coupled**, in the sense that a change in the implementation (but not the methods offered) of one unit does not break the others[a]

[a] Think about a faster implementation, for example.

General idea

Design of a class

- A bad example would be a `ListOrPlayer` class which contains a list or a player, with methods and fields such as:
  - `IsList`, `IsPlayer`
  - `Name`, `Score`, `Weapon`, ...
  - `Length`, `Map`, `Filter`, ...
- Why is it a bad example?

General idea

Design of a class

- A bad example would be a `ListOrPlayer` class which contains a list or a player, with methods and fields such as:
  - `IsList`, `IsPlayer`
  - `Name`, `Score`, `Weapon`, ...
  - `Length`, `Map`, `Filter`, ...
- Why is it a bad example?
- Because it is not a clear unit, but two at the same time

General idea

Design of a class

- A bad example would be a `List` which *leaks* parts of implementation
- A "leaky" list would have strange methods that rely on specific external usage patterns like:
  - `ComputeFirstPartOfLength` that computes the length of the first half of the list
  - `ComputeSecondPartOfLength` that computes the length of the second half of the list
  - `GetLengthParts` that returns the lengths of the two half lists
- Why is it a bad example?

General idea

Design of a class

- A bad example would be a `List` which *leaks* parts of implementation
- A “leaky” list would have strange methods that rely on specific external usage patterns like:
  - `ComputeFirstPartOfLength` that computes the length of the first half of the list
  - `ComputeSecondPartOfLength` that computes the length of the second half of the list
  - `GetLengthParts` that returns the lengths of the two half lists
- Why is it a bad example?
- Because the implementation of length only happens if the user of the list calls it properly

Technical details

Technical details

Introduction

- We have already seen how we can define a class in Python, just without methods (beside `__init__`)
- Adding methods is surprisingly simple
- Simply declare a function within the class body, with the usual syntax
  - The function must[a] have a special parameter `self`, which is the first parameter

[a] There are exceptions, we discuss them later.

Concrete class implementation


```
class Empty:
  def IsEmpty(self):
    return True
  def Length(self):
    return 0
```

Introduction

Concrete class implementation


```
class Empty:
  def IsEmpty(self):
    return True
  def Length(self):
    return 0
```

Introduction

- Notice the `self` parameter of each method

Concrete class implementation


```
class Node:
  def IsEmpty(self): return False
  def __init__(self, x, xs):
    self.head = x
    self.tail = xs
  def Length(self):
    return 1 + self.tail.Length()
```

Introduction

Concrete class implementation


```
class Node:
  def IsEmpty(self): return False
  def __init__(self, x, xs):
    self.head = x
    self.tail = xs
  def Length(self):
    return 1 + self.tail.Length()
```

Introduction

- Notice the `self` parameter of each method

Using classes

- Usage of classes remains quite intuitive
- Instance a class by using its name and giving parameters that will end up in `__init__`
- The returned class can be assigned to a variable
- The methods and attributes of the class can be called with a `.` between the instance of a class and the method/attribute name

Actual class usage


```
l = Node(1, Empty())
print(l.Length())
```

Constructor

- You may now have realized that `__init__` is just another method[a], the **constructor** of a class
- It is called transparently when creating an instance of a class
- We call it with a parameter less (`self`), which is given automatically by Python as a new empty container
- `Node(1,Empty())` invokes the constructor with two parameters, `self` is implicit

[a] A special one, in that it is called automatically by Python, but a regular method nonetheless

Technical details

Methods

- Methods work like the constructor
- We call them with a parameter less (`self`), which is given automatically by Python as the object from which the method was called
- `l.Length()` calls method `Length` with zero parameters, `self` is implicitly passed as `l`

Technical details

Syntax and semantics of methods

- The only new element of semantics is indeed the passing of `self`
- This is described with a code transformation that happens at runtime
- Method call is not really a new feature, but rather a handy way to use existing features

Technical details

Syntax and semantics of methods

- Whenever Python encounters a call such as `x.M(p1, ..., p2` then it *transforms* it into a similar call `C.M(x, p1, ..., pn)` where `C` is the type of `x` in memory
- This way `x` automatically becomes `self` inside `M`

$$\langle x.M(p_1, \dots, p_n), S, H\rangle \rightarrow \langle H[S[x]][C].M(x, p_1, \dots, p_n)\rangle$$

- This presupposes that the heap contains, for each class, an additional attribute `C` that is the class of declaration

...	n	...
...	[...; C $\mapsto$ TypeOfInstance]	...

Technical details

Syntax and semantics of methods

- Methods defined with a `self` parameter are called **instance methods**
- Using them is bound to the calling context (represented by `self`)
- Some methods can be defined without a `self` parameter
- Such methods are called **static methods**, because they are independent of a specific instance

Technical details

Syntax and semantics of methods

- Static methods are used by specifying the name of the class to which the method belongs to `C.M(p1,...,pn)`
- There is nothing special about them, they just behave like any function call
- They are useful to group functionality related to a single class that is not related to the class instances

Technical details

Special method names

- Some special method names are reserved by Python
- These names represent operators that are automatically called by Python
- A typical example is `__str__`, which is automatically invoked whenever `print` or `str` are used

Special method names

- Other special names represent operators
- For example:
  - `__le__` is called whenever `x <= y` was used
  - `__add__` is called whenever `x + y` was used
  - `__lshift__` is called whenever `x << y` was used
  - ...

Technical details

Examples

- Let's begin with a simple example
- A counter that keeps track of how many times some event has happened
- The constructor of the counter starts the count at zero
- A `Tick` method increments the count
- A pretty printer to nicely format instances when they are output

“Counter” example


```
class Counter:
  def __init__(self):
    self.cnt = 0
  def Tick(self, n):
    self.cnt = self.cnt + n
  def __str__(self):
    return "Ticked␣" + str(self.cnt) + "␣times."

c = Counter()
for i in range(0, 20):
  c.Tick(i)
  print(c)
```

What does this program do?

"Counter" example


```
class Counter:
  def __init__(self):
    self.cnt = 0
  def Tick(self, n):
    self.cnt = self.cnt + n
  def __str__(self):
    return "Ticked␣" + str(self.cnt) + "␣times."

c = Counter()
for i in range(0, 20):
  c.Tick(i)
  print(c)
```

What does this program do?

Prints the "ticked" message for `0, 1, 3, 6, 10`

Technical details

Examples

- Let's move to an almost full implementation of lists
- Lists come in two flavours: `Empty`, and `Node`
- We want at least the following methods:
  - `IsEmpty`
  - String conversion
  - << to create lists more easily
  - `Sum` to add all elements of the list
  - `Length` to find the list length
  - `Map` to transform all elements of the list
  - `Filter` to remove some elements of the list
  - `Fold` to collapse the list

Let's start with the empty list


```
class Empty:
  def IsEmpty(self): return True
  def __str__(self):
    return "[]"
  def __rlshift__(self, v):
    return Node(v, self)
  def Sum(self):
    return 0
  def Length(self):
    return 0
  def Map(self,f):
    return self
  def Filter(self,p):
    return self
  def Fold(self,f,z):
    return z
Empty = Empty()
```

The non-empty list


```
class Node:
  def IsEmpty(self): return False
  def Head(self): return self.Head
  def Tail(self): return self.Tail
  def __init__(self, x, xs):
    self.head = x
    self.tail = xs
  def __rlshift__(self, v):
    return Node(v, self)
  def __str__(self):
    return str(self.head) + "<<" + str(self.tail)
  def Sum(self):
    return self.head + self.tail.Sum()
  def Length(self):
    return 1 + self.tail.Length()
  def Map(self,f):
    return Node(f(self.head), self.tail.Map(f))
  def Filter(self,p):
    xs = self.tail.Filter(p)
    if p(self.head):
      return Node(self.head, xs)
    else:
      return xs
  def Fold(self,f,z):
    return f(self.head, self.tail.Fold(f,z))
```

Technical details

Assignments

- Find the implementation (or the slides) online and open them to reference `Node` and `Empty`
- What does `l = 1 << (2 << (3 << (4 << Empty)))` do?

Technical details

Assignments

- Find the implementation (or the slides) online and open them to reference `Node` and `Empty`
- What does `l = 1 << (2 << (3 << (4 << Empty)))` do?
  - It creates list `Node(1, Node(2, Node(3, Node(4, Empty))))`
- Why did we write `Empty` instead of `Empty()`?

Technical details

Assignments

- Find the implementation (or the slides) online and open them to reference `Node` and `Empty`
- What does `l = 1 << (2 << (3 << (4 << Empty)))` do?
  - It creates list `Node(1, Node(2, Node(3, Node(4, Empty))))`
- Why did we write `Empty` instead of `Empty()`?
  - Aesthetics: we define a single instance of `Empty = Empty()` to avoid recalling the constructor every time

Using lists


```
l = 1 << (2 << (3 << (4 << Empty)))
print("length(" + str(l) + ")␣=␣" + str(l.Length()))
print("incr(" + str(l) + ")␣=␣" + str(l.Map(lambda x: x + 1)))
print("even(" + str(l) + ")␣=␣" + str(l.Filter(lambda x: x % 2 == 0)))
print("sum(" + str(l) + ")␣=␣" + str(l.Sum()))
print("mul(" + str(l) + ")␣=␣" + str(l.Fold(lambda x,y: x * y, 1)))
```

What does this program print?

Using lists


```
l = 1 << (2 << (3 << (4 << Empty)))
print("length(" + str(l) + ") = " + str(l.Length()))
print("incr(" + str(l) + ") = " + str(l.Map(lambda x: x + 1)))
print("even(" + str(l) + ") = " + str(l.Filter(lambda x: x % 2 == 0)))
print("sum(" + str(l) + ") = " + str(l.Sum()))
print("mul(" + str(l) + ") = " + str(l.Fold(lambda x,y: x * y, 1)))
```

What does this program print?

- '4', '2<<3<<4<<5', '2<<4', '10', '24']

Technical details

Let's see build a 2D vector

- We want things to move over the screen
- We define 2D vectors to store the position of objects
- 2D vectors contain only `X`, `Y` attributes
- Methods are:
  - Vector addition, subtraction, negation, multiplication by scalar
  - Length of the vector (according to Pythagoras' theorem)
  - Some static methods to get some special vectors (null, (0,1), (1,0)

Implementation of vector 2D


```
class Vector2:
  def __init__(self, x, y):
    self.X = x
    self.Y = y
  def Length(self):
    return math.sqrt(self.X * self.X + self.Y * self.Y)
  def __neg__(self):
    return Vector2(-self.X, -self.Y)
  def __add__(self, other):
    return Vector2(self.X + other.X, self.Y + other.Y)
  def __sub__(self, other):
    return self + (-other);
  def __mul__(self, k):
    return Vector2(self.X * k, self.Y * k)
  def __str__(self):
    return "(" + str(self.X) + "," + str(self.Y) + ")"
  def Zero():
    return Vector2(0.0, 0.0)
  def UnitX():
    return Vector2(1.0, 0.0)
  def UnitY():
    return Vector2(0.0, 1.0)
```

Let's build a 2D vector


```
v = Vector2.UnitX() * 10.0 - Vector2.UnitY() * 2.0
print(str(v))
```

What does this program print?

Let's build a 2D vector

```
v = Vector2.UnitX() * 10.0 - Vector2.UnitY() * 2.0
print(str(v))
```

What does this program print?

- (10.0, -2.0)]

Technical details

Let's make a car

- A car has a `Position` and a `Velocity`, both `Vector2`'s
- A car also has `Gas` in the tank
- A car can `Travel`: `Position` moves forward, `Gas` is burned

Implementation of car

```
class Car:
  def __init__(self, p, v, g):
    self.Position = p
    self.Velocity = v
    self.Gas = g
  def Travel(self, dt):
    if self.Gas > 0.0:
      self.Position = self.Position + self.Velocity * dt
      self.Gas = self.Gas - 1.0 * dt
  def __str__(self):
    return "A car at " + str(self.Position) + " with a tank of " + str(self.Gas) + " liters"
```

Let's build a car!


```
c = Car(Vector2.Zero(), Vector2.UnitX(), 10.0)
while(c.Gas > 0.0):
  c.Travel(2.0)
  print(c)
```

What does this program print?

Let's build a car!

```
c = Car(Vector2.Zero(), Vector2.UnitX(), 10.0)
while(c.Gas > 0.0):
  c.Travel(2.0)
  print(c)
```

What does this program print?

- A car at (2.0,0.0) with a tank of 8.0 liters
- A car at (4.0,0.0) with a tank of 6.0 liters
- A car at (6.0,0.0) with a tank of 4.0 liters
- A car at (8.0,0.0) with a tank of 2.0 liters
- A car at (10.0,0.0) with a tank of 0.0 liters

A few advanced design topics

A few advanced design topics

General idea

- There are countless ways to design classes
- Each of these has advantages and disadvantages
- It is a design discipline, and as such "artsier" than strictly needed
- Experience will show the way
- For the moment, we just illustrate one such possibility

A few advanced design topics

Immutable and mutable classes

- Not all classes modify the values of their attributes as methods are called
- Some classes can be designed to be **immutable**
- An instance never changes values after creation
- Methods **return a new instance with the new attribute values** instead of changing the attribute values of the starting instance

A few advanced design topics

Immutable and mutable classes

- Never changing attributes is a powerful technique
- Sharing of instances is safer, because unexpected changes will never happen
- **Can you think of examples when this might be useful?**

A few advanced design topics

Immutable and mutable classes

- Never changing attributes is a powerful technique
- Sharing of instances is safer, because unexpected changes will never happen
- **Can you think of examples when this might be useful?**
- **Breaking another class which assumes that the given instance will always remain as expected**
- **Multiple threads**
- **Rollback/checkpoints/transactional systems**
- ...

A few advanced design topics

Let's make an immutable car

- A car has a `Position` and a `Velocity`, both `Vector2`'s
- A car also has `Gas` in the tank
- A car can `Travel`: **a new car is returned where** `Position` is moved forward, `Gas` is burned

Implementation of an immutable car

```
class ImmutableCar:
  def __init__(self, p, v, g):
    self.Position = p
    self.Velocity = v
    self.Gas = g
  def Travel(self, dt):
    if self.Gas > 0.0:
      return Car(self.Position + self.Velocity * dt, self.Velocity, self.Gas
            - 1.0 * dt)
    else:
      return self
  def __str__(self):
    return "A␣car␣at␣" + str(self.Position) + "␣with␣a␣tank␣of␣" + str(self.
          Gas) + "␣liters"
```

Let's build an immutable car!

Classes as data structures with methods

TEAM INFDEV

Introduction

Problem discussion

General idea

Technical details

A few advanced design topics

Conclusion

```
c = ImmutableCar(Vector2.Zero(), Vector2.UnitX(), 10.0)
while(c.Gas > 0.0):
  c = c.Travel(2.0)
  print(c)
```

What does this program print?

Let's build an immutable car!

```
c = ImmutableCar(Vector2.Zero(), Vector2.UnitX(), 10.0)
while(c.Gas > 0.0):
  c = c.Travel(2.0)
  print(c)
```

What does this program print?

- A car at (2.0,0.0) with a tank of 8.0 liters
- A car at (4.0,0.0) with a tank of 6.0 liters
- A car at (6.0,0.0) with a tank of 4.0 liters
- A car at (8.0,0.0) with a tank of 2.0 liters
- A car at (10.0,0.0) with a tank of 0.0 liters

Rolling back time

- Suppose we stored the initial value of the car `c0 = ImmutableCar(Vector2.Zero(), Vector2.UnitX(), 10.0)` before the loop
- Then we just run the loop
- **Can we "roll back time" by going back to the initial value of the car?**

Rolling back time

- Suppose we stored the initial value of the car `c0 = ImmutableCar(Vector2.Zero(), Vector2.UnitX(), 10.0)` before the loop
- Then we just run the loop
- **Can we "roll back time" by going back to the initial value of the car?**
- Yes!
- **Was this possible with the first implementation?**

A few advanced design topics

Rolling back time

- Suppose we stored the initial value of the car `c0 = ImmutableCar(Vector2.Zero(), Vector2.UnitX(), 10.0)` before the loop
- Then we just run the loop
- **Can we "roll back time" by going back to the initial value of the car?**
- Yes!
- **Was this possible with the first implementation?**
- No!

Rolling back time


```
c = ImmutableCar(Vector2.Zero(), Vector2.UnitX(), 10.0)
c0 = c
while(c.Gas > 0.0):
  c = c.Travel(2.0)
  print(c)
print(c0)
```

What is stored in `c0`?

Rolling back time


```
c = ImmutableCar(Vector2.Zero(), Vector2.UnitX(), 10.0)
c0 = c
while(c.Gas > 0.0):
  c = c.Travel(2.0)
  print(c)
print(c0)
```

What is stored in `c0`? The initial value of the car, unchanged!

Choices, choices

- Why would you use one design instead of the other?
- Mutable classes have
  - the advantage that they are simple to build and intuitive
  - the disadvantage that they are "destructive", in the sense that all modification destroys previous information that might still be in use
- Immutable classes have
  - the advantage that they are not destructive
  - the disadvantage that they are more complex to build and structure
- Mutable classes for current state of an object
- Immutable classes for things that are not supposed to change (vectors, numbers) or that might need to be rolled back

Conclusion

Conclusion

Lecture topics

- In this lecture we have “closed the circle” of data structures and functions
- We have defined **classes** as the joining of functions and data structures
- We have seen a series of class implementation examples in action
- We have discussed different design balances that might come into consideration when choosing the structure of a class

This is it!

The best of luck, and thanks for the attention!