Data structures

TEAM INFDEV

Hogeschool Rotterdam
Rotterdam, Netherlands

Introduction

Introduction

Lecture topics

- Mechanism of abstraction
- The need for data structures
- Classes as data structures in Python
- Tuples and records

What is abstraction?

What is abstraction?

Introduction

- The big issue of the whole course is **abstraction** in programming
- Abstraction is a fundamental concept in programming to reduce repetition
- We sit atop a mountain of abstraction, which we make taller at every iteration

What is abstraction?

Grab the student next to you

- Describe what you just did so that someone else can perform the same action

What is abstraction?

Grab the student next to you

- Describe what you just did so that someone else can perform the same action
- Now add specific details about the movements of your arm and phalanges (pieces of fingers)

What is abstraction?

Grab the student next to you

- Describe what you just did so that someone else can perform the same action
- Now add specific details about the movements of your arm and phalanges (pieces of fingers)
- Now realize that there are even more subcomponents: individual muscles, tendons, etc.

What is abstraction?

Grab the student next to you

- Describe what you just did so that someone else can perform the same action
- Now add specific details about the movements of your arm and phalanges (pieces of fingers)
- Now realize that there are even more subcomponents: individual muscles, tendons, etc.
- But then we have also cells that make these up
- ...

What is abstraction?

Human love for abstraction

- Our brain cannot handle so many details
- To cope with this, we are structured in layers
- Our consciousness manipulates only the upper layers with simple instructions
- *Raise arm above head*

What is abstraction?

Human love for abstraction

- The same happens with regular language
- "*Go buy a liter of milk*" is quite a short description
- The underlying operation is very complex

Complexity of simple instructions

```
Go buy a liter of milk =
  Turn game off
  Get up from the couch
  Curse the instruction giver
  Get dressed
  Put money in pocket
  Leave house
  Reach nearest shop
  Enter shop
  Find milk
  Take one liter bottle
  Pay milk
  Go home
  Give milk to instruction giver
```

What is abstraction?

Human love for abstraction

- And clearly something like “*reach nearest shop*” is not a trivial instruction by itself
- Think about all the things you give for granted
  - Crossing roads
  - Traffic lights
  - Pathfinding
  - Road work and obstructions
  - Use of transportation methods
  - ...

Data structures

Data structures

Flying back to Earth

- How is this relevant for programmers?
- We have a similar issue with a modern computer

A single Python instruction runs

```
+-----------------+
| VM instructions |
+----------------------+
| Machine instruction  |
+---------------------------+
| CPU components            |
+--------------------------------+
| Logic gates                    |
+--------------------------------+
...
```

Flying back to Earth

- Moreover, sometimes we have repetition of constructs in our own code
- This means that we would like to extend the pyramid with our own stuff

A single Python program runs


```
+------------+
| Own stuff  |
+-----------------+
| VM instructions |
+----------------------+
| Machine instruction  |
+--------------------------+
| CPU components           |
+------------------------------+
| Logic gates                  |
+------------------------------+
...
```

What kind of "*own stuff*"?

- Any recurring structure, code, etc.
- We do not want to repeat it every time
- We just give it a name, instead of specifying it every time
- The actual goal is to make things simpler
  - Code reuse, maintainability, etc. do not exist
  - It is all just **properly built abstractions that make reasoning about code easier**

Repeated code


```
playerOneName = "P1"
playerOnePositionX = 0.0
playerOnePositionY = 0.0

playerTwoName = "P2"
playerTwoPositionX = 5.0
playerTwoPositionY = 0.0

playerThreeName = "P3"
playerThreePositionX = 10.0
playerThreePositionY = 0.0
```

Repeated code


```
playerOneName = "P1"
playerOnePositionX = 0.0
playerOnePositionY = 0.0

playerTwoName = "P2"
playerTwoPositionX = 5.0
playerTwoPositionY = 0.0

playerThreeName = "P3"
playerThreePositionX = 10.0
playerThreePositionY = 0.0
```

Now let's add a score, an exp level, etc.

Repeated code


```
playerOneName = "P1"
playerOnePositionX = 0.0
playerOnePositionY = 0.0

playerTwoName = "P2"
playerTwoPositionX = 5.0
playerTwoPositionY = 0.0

playerThreeName = "P3"
playerThreePositionX = 10.0
playerThreePositionY = 0.0
```

Now let's add a score, an exp level, etc.

Does it scale well?

Data structures

Make some examples

- **Everyone make an example of repeated structures of data.**
- **Some of you will present theirs**

General idea

Introduction

- A possible solution to this problem is capturing the repetition of data structures
- With a name, and a specification of what is common about them

General idea

Fundamental ingredients of the solution

- Brains of the programmer, always active
- Abstraction requires awareness and experience
- It is as much technique as it is art

Repeated code

```
playerOneName = "P1"
playerOnePositionX = 0.0
playerOnePositionY = 0.0

playerTwoName = "P2"
playerTwoPositionX = 5.0
playerTwoPositionY = 0.0

playerThreeName = "P3"
playerThreePositionX = 10.0
playerThreePositionY = 0.0
```

General idea

Fundamental ingredients of the solution

- We observe that there is an underlying pattern, which we will call **abstraction**
- The pattern, or abstraction, comes repeated in several **concrete instances** in our program

General idea

Fundamental ingredients of the solution

- We observe that there is an underlying pattern, which we will call **abstraction**
- The pattern, or abstraction, comes repeated in several **concrete instances** in our program
- In the program above this is fairly obvious, in real life not always really :)

General idea

Fundamental ingredients of the solution

- A proper name for the abstraction
- **For example?**

General idea

Fundamental ingredients of the solution

- A proper name for the abstraction
- **For example?** `Player`

General idea

Fundamental ingredients of the solution

- A set of common attributes
- All characterizing aspects of the abstraction that are common to all its instances
- **For example?**

General idea

Fundamental ingredients of the solution

- A set of common attributes
- All characterizing aspects of the abstraction that are common to all its instances
- **For example?** `Name`, `PositionX`, `PositionY`

The blueprint (**THIS IS NOT CODE!**)

```
Abstraction Player =
  Name, which is a string
  PositionX, which is a number
  PositionY, which is a number
```

The abstraction above is called a **data structure**.

It is not valid Python code, but it is a blueprint specifying a recurrent set of attributes that often go together to identify a player.

Technical details

Technical details

How is this done in Python?

- Python offers a facility called `class`
- It is used to capture a data structure.

Syntax of Python classes

```
class <<Name>>:
  def __init__(self, <<v1>>, <<v2>>, ..., <<vN>):
    self.<<A1>> = <<v1>>
    self.<<A2>> = <<v2>>
    ...
    self.<<AN>> = <<vN>>
```

The class has thus: `name`, `initial values` v_1 through v_N, and `attributes` A_1 through A_N initialized with `__init__`.

`self` is a reference to the `concrete instance` that is being set up.

Usage of Python classes


```
x = <<Name>>(<<v1>>, <<v2>>, ..., <<vN>)
```

Sets up a `concrete instance` of <<Name>> with some `initial values`.

Usage of Python classes


```
print(x.<<A2>>)
```

Prints the value of the second `attribute` of the `concrete instance` called `x` of class `<<Name>>`.

Usage of Python classes


```
x.<<A3>> = y
```

Assigns y as the new value of the third attribute of the concrete instance called x of class <<Name>>.

Technical details

Semantics of Python classes

- The semantics of Python classes require a more sophisticated model of memory
- Memory is now divided in two

STACK The state that we used so far, for primitive values (`int`, `string`, etc.)

HEAP A storage for complex values such as classes

Technical details

Semantics of Python classes

- An instruction `I` will now transform the initial heap and stack `H,S` into the resulting (possibly changed) heap and stack `H',S'` [a]

$$< PC, H, S > \xrightarrow{I} < PC', H', S' >$$

[a] in addition to the program counter `PC`, which always behaves in the same way

Technical details

Semantics of creation

- Consider creation of a Python class: `x = <<Name>>(...)` (shortened to `xName`)
- This affects both memories

HEAP We create and initialize a new instance of class `<<Name>>`

STACK We add an entry `x` to the stack, which references to the newly created instance

Semantics of creation

- Given that:
- $|H|$ is the size of the heap at creation, which we call the **address** of the new instance
- $\langle\langle Name \rangle\rangle(...)$ is a new instance of the class, which contains a map from the attribute names to their values

$$< PC, H, S > \stackrel{x Name}{\rightarrow} < PC + 1, H[|H| \mapsto \langle\langle Name \rangle\rangle(\dots)], S[x \mapsto |H|] >$$

- x is, unsurprisingly, called a **reference**
  - it does not contain the value of the class instance
  - it merely tells us where to find it

Technical details

Attribute lookup

- Consider reading an attribute (also called lookup)
- `x.<<A>>` (shortened to `xA`)
- Where is it in memory?

STACK We find an entry `x`, which tells us where the corresponding instance of the class is found

HEAP We find the actual attribute in the map of attributes

$$< PC, H, S > \stackrel{xA}{\to} < PC + 1, H[S[x]][\langle\!\langle A \rangle\!\rangle], S >$$

Technical details

Attribute update

- Consider assigning to an attribute
- `x.<<A>> = v` (shortened to `xAv`)
- Where is it in memory?

STACK We find an entry `x`, which tells us where the corresponding instance of the class is found

HEAP We reassign the actual attribute in the map of attributes

$$< PC, H, S > \overset{xAv}{\rightarrow} < PC + 1, H[S[x] \mapsto S[x][A \mapsto v]]$$

Technical details

Examples

- We can now implement our player data type
- We will use a Python class to do so
- We will then create concrete instances of it, and use them

The blueprint to implement

```
Abstraction Player =
  Name, which is a string
  PositionX, which is a number
  PositionY, which is a number
```

The implemented class


```
class Player:
  def __init__(self, name, posX, posY):
    self.Name = name
    self.PositionX = posX
    self.PositionY = posY
```

Creating concrete instances


```
playerOneName = "P1"
playerOnePositionX = 0.0
playerOnePositionY = 0.0

playerTwoName = "P2"
playerTwoPositionX = 5.0
playerTwoPositionY = 0.0

playerThreeName = "P3"
playerThreePositionX = 10.0
playerThreePositionY = 0.0
```

Becomes:

```
playerOne   = Player("P1", 0.0, 0.0)
playerTwo   = Player("P2", 5.0, 0.0)
playerThree = Player("P3", 10.0, 0.0)
```

Creating concrete instances

S

PC
1

H


```
playerOne   = Player("P1", 0.0, 0.0)
playerTwo   = Player("P2", 5.0, 0.0)
playerThree = Player("P3", 10.0, 0.0)
```

Creating concrete instances

S

PC
1

H


```
playerOne   = Player("P1", 0.0, 0.0)
playerTwo   = Player("P2", 5.0, 0.0)
playerThree = Player("P3", 10.0, 0.0)
```

S

PC	playerOne
2	ref(0)

H

0
[N ↦ "P1"; PX ↦ 0.0; PY ↦ 0.0]

Creating concrete instances

S

PC	playerOne
2	ref(0)

H

0
[N $\mapsto$ "P1"; PX $\mapsto$ 0.0; PY $\mapsto$ 0.0]

```
playerOne   = Player("P1", 0.0, 0.0)
playerTwo   = Player("P2", 5.0, 0.0)
playerThree = Player("P3", 10.0, 0.0)
```

Creating concrete instances

S

PC	playerOne
2	ref(0)

H

0
[N ↦ "P1"; PX ↦ 0.0; PY ↦ 0.0]

```
playerOne   = Player("P1", 0.0, 0.0)
playerTwo   = Player("P2", 5.0, 0.0)
playerThree = Player("P3", 10.0, 0.0)
```

S

PC	playerOne	playerTwo
3	ref(0)	ref(1)

H

0	1
...	[N ↦ "P2"; PX ↦ 5.0; PY ↦ 0.0]

Creating concrete instances

S

PC	playerOne	playerTwo
3	ref(0)	ref(1)

H

0	1
...	[N $\mapsto$ "P2"; PX $\mapsto$ 5.0; PY $\mapsto$ 0.0]

```
playerOne   = Player("P1", 0.0, 0.0)
playerTwo   = Player("P2", 5.0, 0.0)
playerThree = Player("P3", 10.0, 0.0)
```

Creating concrete instances

S

PC	playerOne	playerTwo
3	ref(0)	ref(1)

H

0	1
...	[N ↦ "P2"; PX ↦ 5.0; PY ↦ 0.0]

```
playerOne   = Player("P1", 0.0, 0.0)
playerTwo   = Player("P2", 5.0, 0.0)
playerThree = Player("P3", 10.0, 0.0)
```

S

PC	playerOne	playerTwo	playerThree
4	ref(0)	ref(1)	ref(2)

H

0	1	2
...	...	[N ↦ "P3"; PX ↦ 10.0; PY ↦ 0.0]

Using the concrete instances

Suppose we wish to access `playerOne.PositionX`

S

PC	playerOne	playerTwo	playerThree
4	ref(0)	ref(1)	ref(2)

H

0	1	2
[N $\mapsto$ "P1"; PX $\mapsto$ 0.0; PY $\mapsto$ 0.0]	...	...

Using the concrete instances

Suppose we wish to access `playerOne.PositionX`

S

PC	playerOne	playerTwo	playerThree
4	ref(0)	ref(1)	ref(2)

H

0	1	2
[N $\mapsto$ "P1"; PX $\mapsto$ 0.0; PY $\mapsto$ 0.0]	...	...

First we look in the stack:

S

PC	playerOne	playerTwo	playerThree
5	ref(0)	ref(1)	ref(2)

H

0	1	2
[N $\mapsto$ "P1"; PX $\mapsto$ 0.0; PY $\mapsto$ 0.0]	...	...

Using the concrete instances

Suppose we wish to access `playerOne.PositionX`

S

PC	playerOne	playerTwo	playerThree
5	ref(0)	ref(1)	ref(2)

H

0	1	2
[N $\mapsto$ "P1"; PX $\mapsto$ 0.0; PY $\mapsto$ 0.0]	...	...

Using the concrete instances

Suppose we wish to access `playerOne.PositionX`

S

PC	playerOne	playerTwo	playerThree
5	ref(0)	ref(1)	ref(2)

H

0	1	2
[N $\mapsto$ "P1"; PX $\mapsto$ 0.0; PY $\mapsto$ 0.0]	...	...

Then we look in the heap:

S

PC	playerOne	playerTwo	playerThree
5	ref(0)	ref(1)	ref(2)

H

0	1	2
[N $\mapsto$ "P1"; PX $\mapsto$ 0.0; PY $\mapsto$ 0.0]	...	...

Using the concrete instances

Suppose we wish to access `playerOne.PositionX`

S

PC	playerOne	playerTwo	playerThree
5	ref(0)	ref(1)	ref(2)

H

0	1	2
[N $\mapsto$ "P1"; PX $\mapsto$ 0.0; PY $\mapsto$ 0.0]	...	...

Using the concrete instances

Suppose we wish to access `playerOne.PositionX`

S

PC	playerOne	playerTwo	playerThree
5	ref(0)	ref(1)	ref(2)

H

0	1	2
[N ↦ "P1"; PX ↦ 0.0; PY ↦ 0.0]	...	...

Finally we search the right attribute (`PositionX`):

S

PC	playerOne	playerTwo	playerThree
5	ref(0)	ref(1)	ref(2)

H

0	1	2
[N ↦ "P1"; PX ↦ 0.0 ; PY ↦ 0.0]	...	...

Designing data structures

Designing data structures

Are we there yet?

- We can keep extending our knowledge about the problem
- For example, we might notice that `PositionX` and `PositionY` might happen in other places of the program
- **What could we do?**

Are we there yet?

- We can keep extending our knowledge about the problem
- For example, we might notice that `PositionX` and `PositionY` might happen in other places of the program
- **What could we do?**
- We could define a `Point2D` (or `Vector2D`) data structure!

Refined data structures

```
class Vector2:
  def __init__(self, x, y):
    self.X = x
    self.Y = y

class PlayerRefined:
  def __init__(self, name, posX, posY):
    self.Name = name
    self.Position = Vector2(posX,posY)
```

Designing data structures

Refined data structures

- Creation is precisely identical to the previous sample
- The `__init__` of the `PlayerRefined` has the same inputs
- Where we had `playerOne = Player("P1", 0.0, 0.0)`
- Now we have `playerOne = PlayerRefined("P1", 0.0, 0.0)`

Designing data structures

Refined data structures

- Usage of the new player definition is almost identical to the previous
- Only changes are lookups like: `playerOne.PositionY`
- **What do they become now?**
- `playerOne.Position.Y`

Refined data structures

- What does memory look like now with a player that contains a vector?
- Stack is similar to previous instance
- Heap contains a reference to a vector!

Creating concrete instances

S	PC
	1
H	

```
playerOne    = PlayerRefined("P1", 0.0, 0.0)
```

Creating concrete instances

S

PC
1

H


```
playerOne    = PlayerRefined("P1", 0.0, 0.0)
```

S

PC	playerOne
2	ref(0)

H

0	1
[N $\mapsto$ "P1"; P $\mapsto$ ref(1);]	[X $\mapsto$ 0.0; Y $\mapsto$ 0.0]

Designing data structures

What characterizes a good design of data structures?

- **Reuse** of code in places where otherwise repetition would happen
- **Encapsulation** of the semantics of the data structure
- **Loose coupling** between the data structure and the rest of the program

Designing data structures

Reuse of code

- Repetition is dangerous
- A small change in one place but not in the others can lead to unexpected consequences
- More code to read means more mental overhead
- Actual work of the program is hidden under lots of noise and thus less visible

Designing data structures

Encapsulation

- A data structure has a single, clear, well-defined goal
- Its name clearly explains what it contains and does
- There is no multiple functionality mix

Designing data structures

Encapsulation

- A data structure has a single, clear, well-defined goal
- Its name clearly explains what it contains and does
- There is no multiple functionality mix
- It's a cold beer, not a cocktail

Designing data structures

Loose coupling

- A data structure is a closed and complete unit
- To use it, you just need to declare it and initialize it
- The rest of the program integrates a well-designed data structure with minimal modification

Designing data structures

How do we verify all this?!?

- Takes experience and good taste
- It is an old story
- Remember: you have the power to make your own life a living Hell...

Designing data structures

How do we verify all this?!?

- Takes experience and good taste
- It is an old story
- Remember: you have the power to make your own life a living Hell...
- ...unless you reason first and write code after

Assignment

Assignment

Build, in class, a series of data structures

- Tyre
- Wheel
- Engine
- Seat
- Light
- Person (driver and passenger)
- Car

Conclusion

Conclusion

Lecture topics

- Abstraction is the fundamental mechanism that allows us to group concepts together and refer to them as if they were a single concept
- For example, a name and two numbers became a `player`
- We then use the new concept (the `player`) without having to explicitly mention all of its components every time
- This makes it leaner for us to manipulate complex programs, as less concepts ("actors") make an appearance

This is it!

The best of luck, and thanks for the attention!