Introduction to Programming - Day 2

Ben Evans

kathik

Going Further With Python

- Introducing OO
- OO in Python
  - Classes
  - Design Patterns

Introducing OO

- Object Oriented Programming
- Basic Definitions
- Thinking About Data
- State Models

A Warning

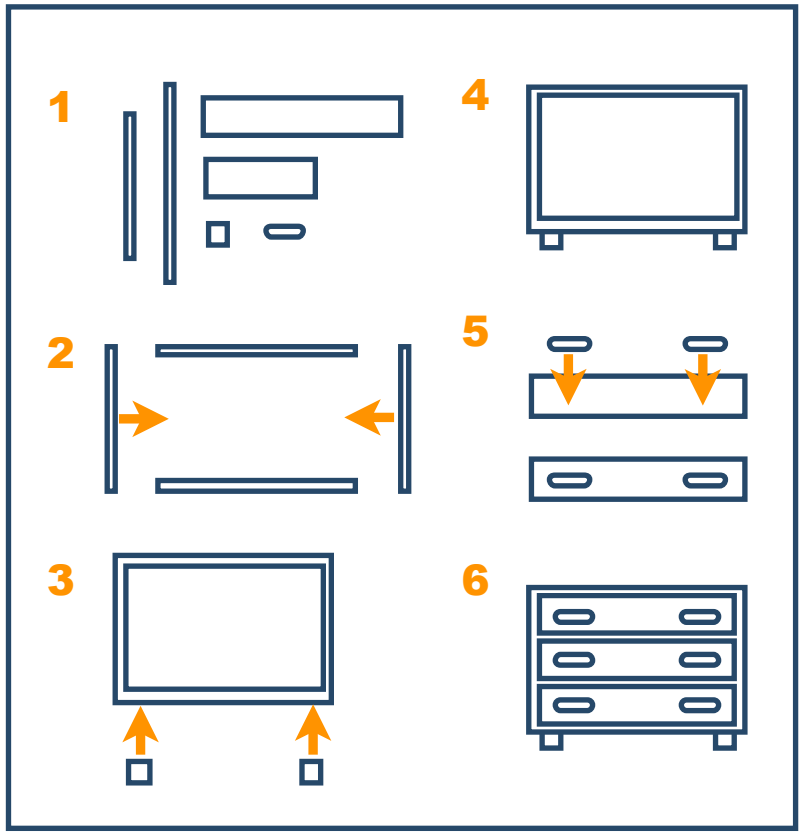
- If you have experience with OO programming, be careful
- "Object-oriented" has different meanings in different languages.
- Don't assume that Python works the same way as other OO languages
  - Particularly true for C++ programmers

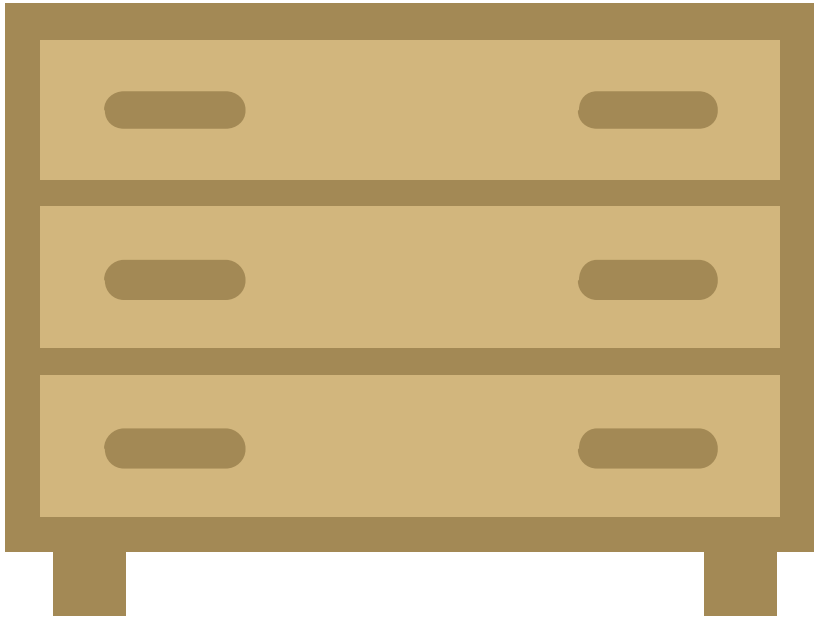
Definitions

- Class
  - A collection of
    - Data fields that hold values
    - Methods that operate on those values

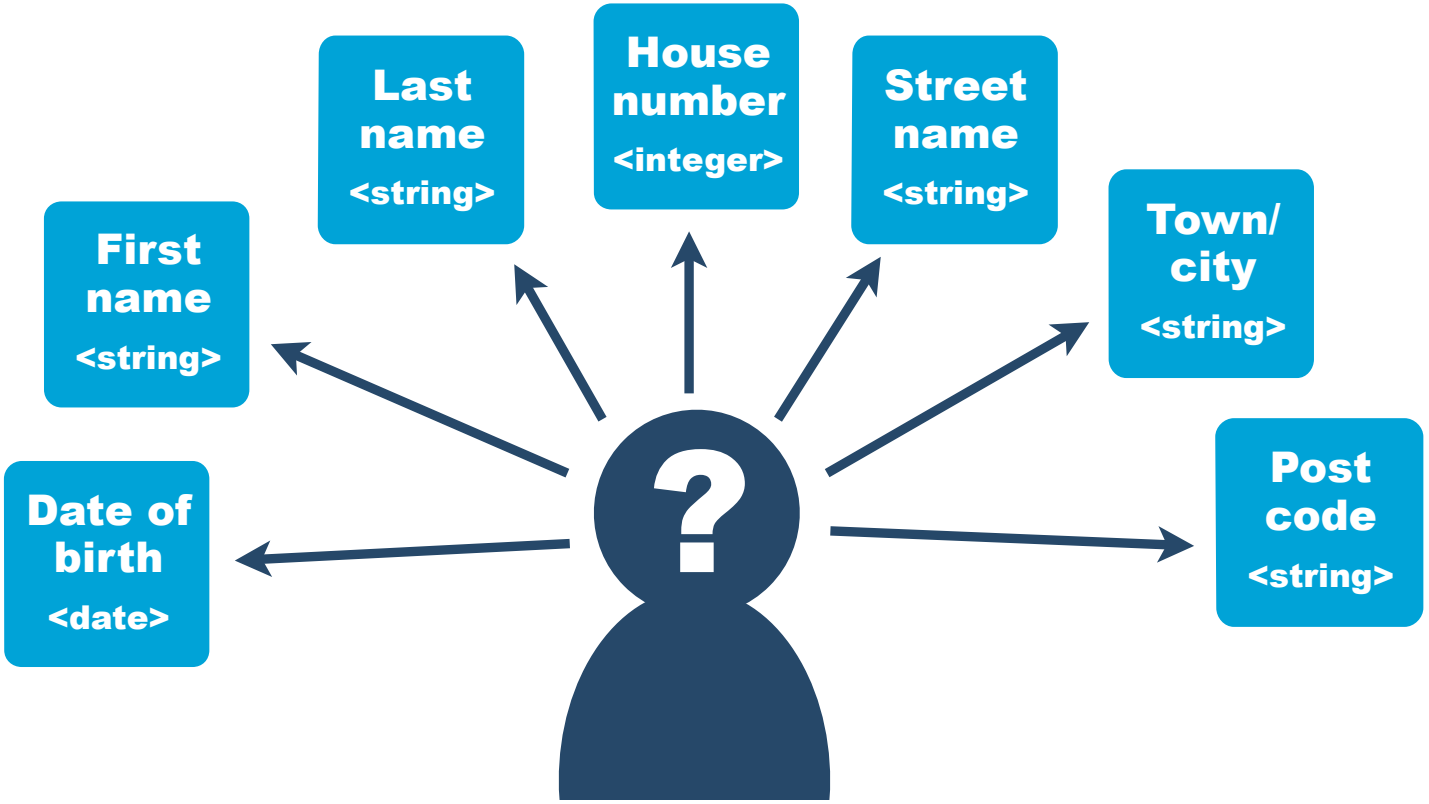
- Object
  - An instance of a class
  - “A bundle of state and methods to act on that state”

Class vs Object

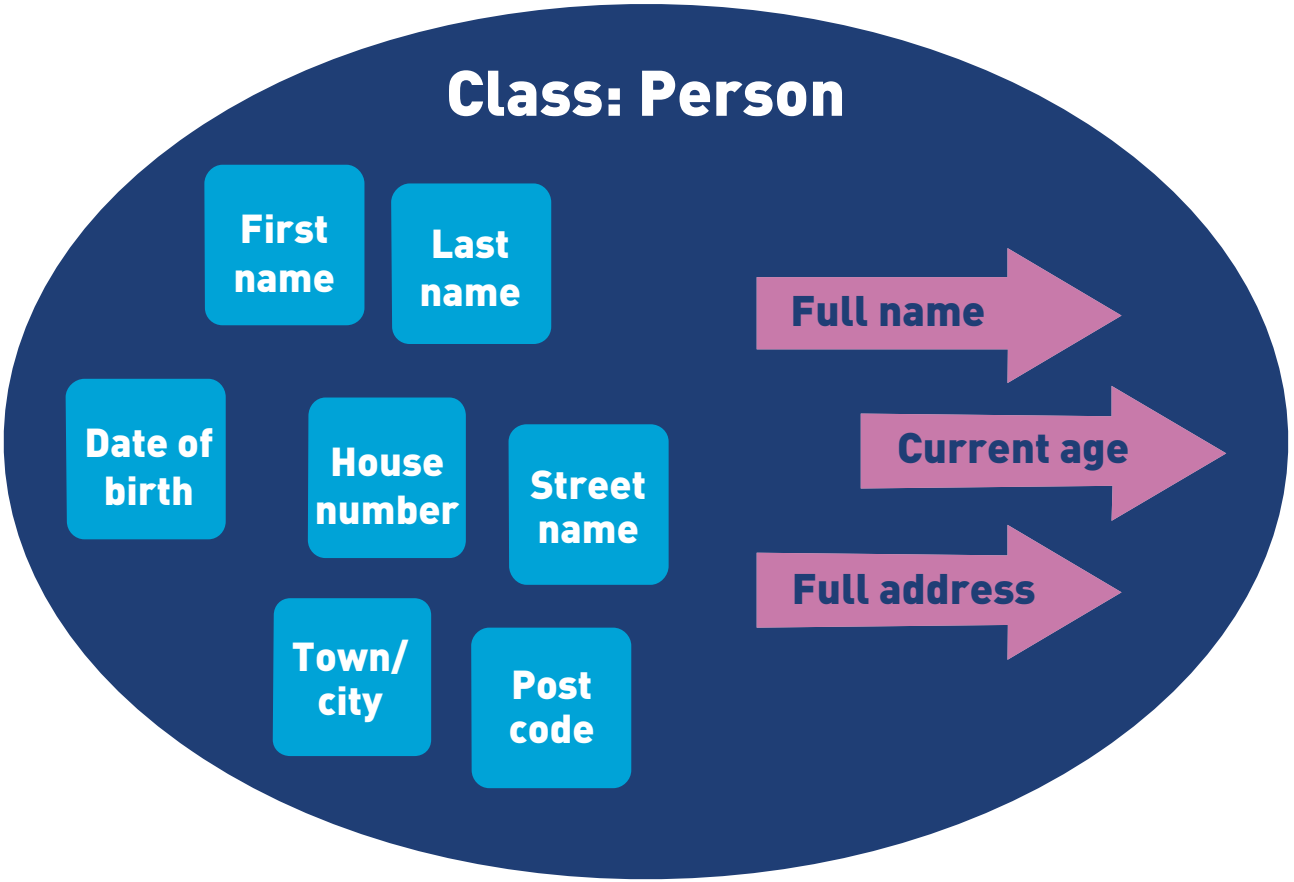




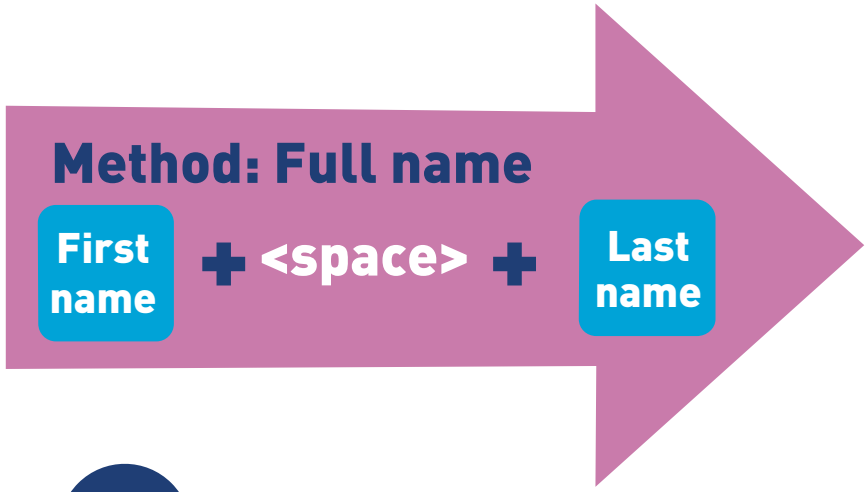
Thinking about data

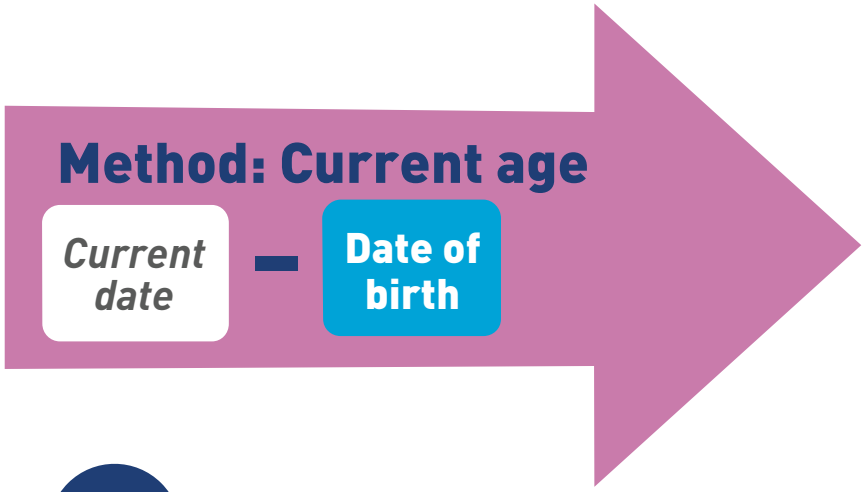


Classes


Class: Person
First name
Last name
Date of birth
House number
Street name
Town/ city
Post code
Full name
Current age
Full address

Methods

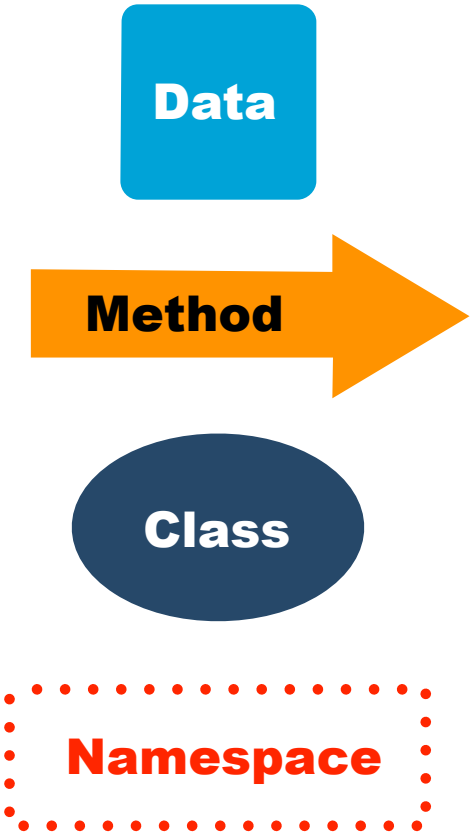




First name: Bob
Last name: Smith
Full name: **Bob Smith**

Current date: 24th May 2015
Date of birth: 21st May 1976
Current age: **38 years old**

Summary: Building blocks



OO in Python

- Classes in Python
- Composition
- Inheritance

OO in Python

- Let's see how to do this in Python
  - __init__ is the constructor

```
>>> class Address:
...     'This class represents an address'
...     def __init__(self, number, street, city, postcode):
...         self.number = number
...         self.street = street
...         self.city = city
...         self.postcode = postcode
...     def address(self):
...         return self.number + ' ' + self.street + ' ' + self.city + ' ' + self.postcode
...
>>> hudson_flat = Address("221b", "Baker St", "London", "NW1 1XX")
>>> hudson_flat.address()
'221b Baker St London NW1 1XX'
```

OO in Python

- Methods are public by default
- Methods are virtual by default
- Explicit ‘self’ parameter (provided via syntactic sugar)
- Classes are themselves objects
- Classes are created at runtime
- Python supports class fields as well as instance

OO in Python

- Extend from addresses to people...

```
>>> class Person:
...     'This class represents a person'
...     def __init__(self, first_name, surname, dob, address):
...         self.first_name = first_name
...         self.surname = surname
...         self.dob = dob
...         self.address = address
...     def full_name(self):
...         return self.first_name +' '+ self.surname
```

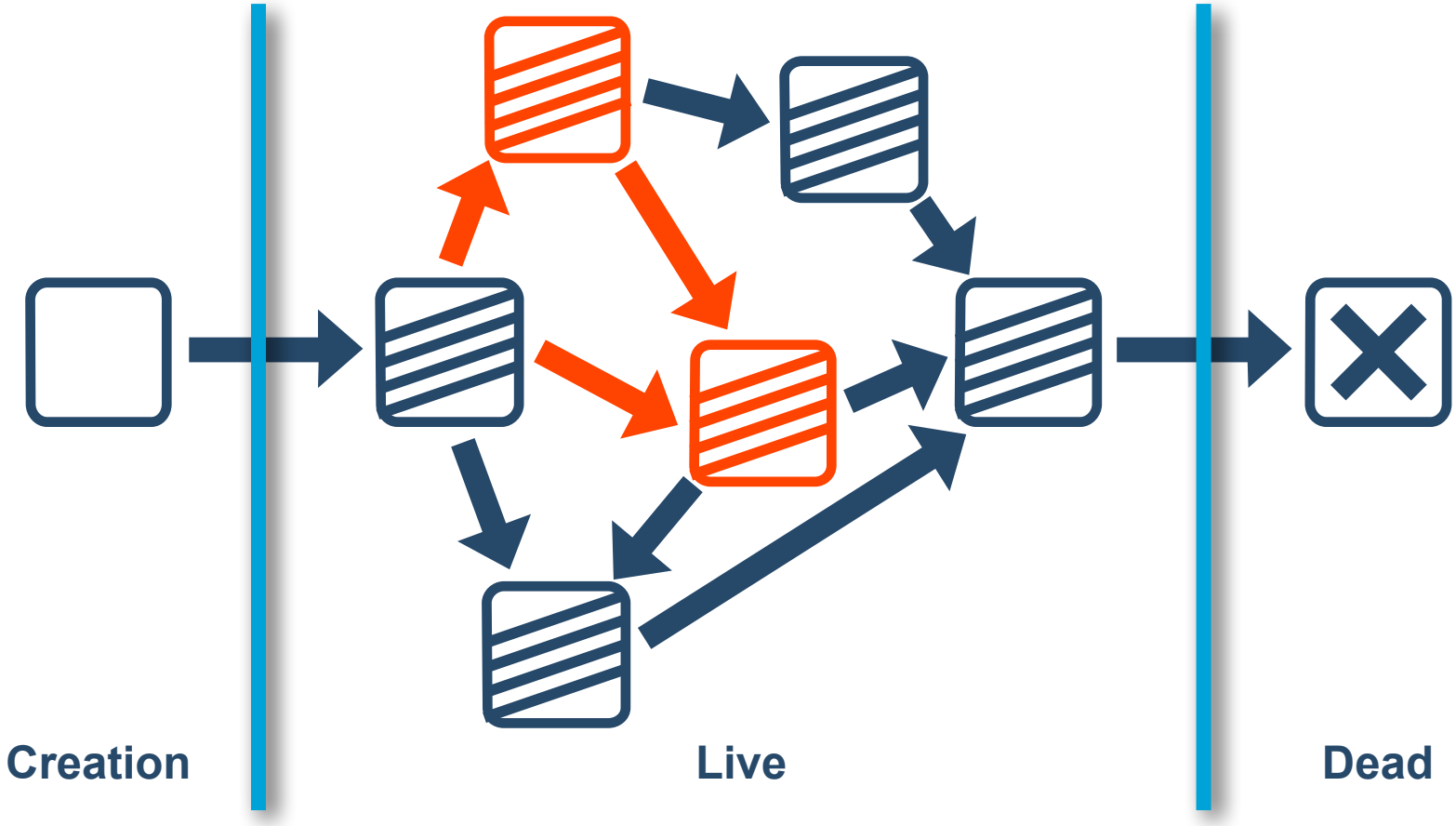
OO in Python

- Let's do some inheritance...
  - Note the explicit super call

```
class Detective(Person):
    def __init__(self, first_name, surname, dob, address, assistant):
        Person.__init__(self, first_name, surname, dob, address)
        self.assistant = assistant

>>> watson = Person("John", "Watson", "1974-03-22", hudson_flat)
>>> holmes = Detective("Sherlock", "Holmes", "1976-07-19", hudson_flat, watson)
```

OO State Transition Model



Group Exercise A

- Design an object system to represent iTunes listens
- Group Discussion: 20-30 minutes

Patterns & AntiPatterns

- Pattern
  - A small-scale best practice in code
  - E.g. Iterator, Adapter, Singleton, Write-behind journal
- Antipattern
  - A larger-scale dysfunction of a project or team
  - E.g. Not Invented Here

Patterns: Recurring Themes

- How can we learn how to do better?
- Pattern Languages
  - A way of talking about software design
  - Higher level of abstraction
  - The essence of the software
  - Not tied to specific projects
  - Developers love war stories

Adapter

kathik

Adapter

Simplicity?

Londoner (proud of the Tube system, to friends from the country). "THERE'S THE WHOLE THING, YOU SEE! *ABSOLUTELY SIMPLE!*"

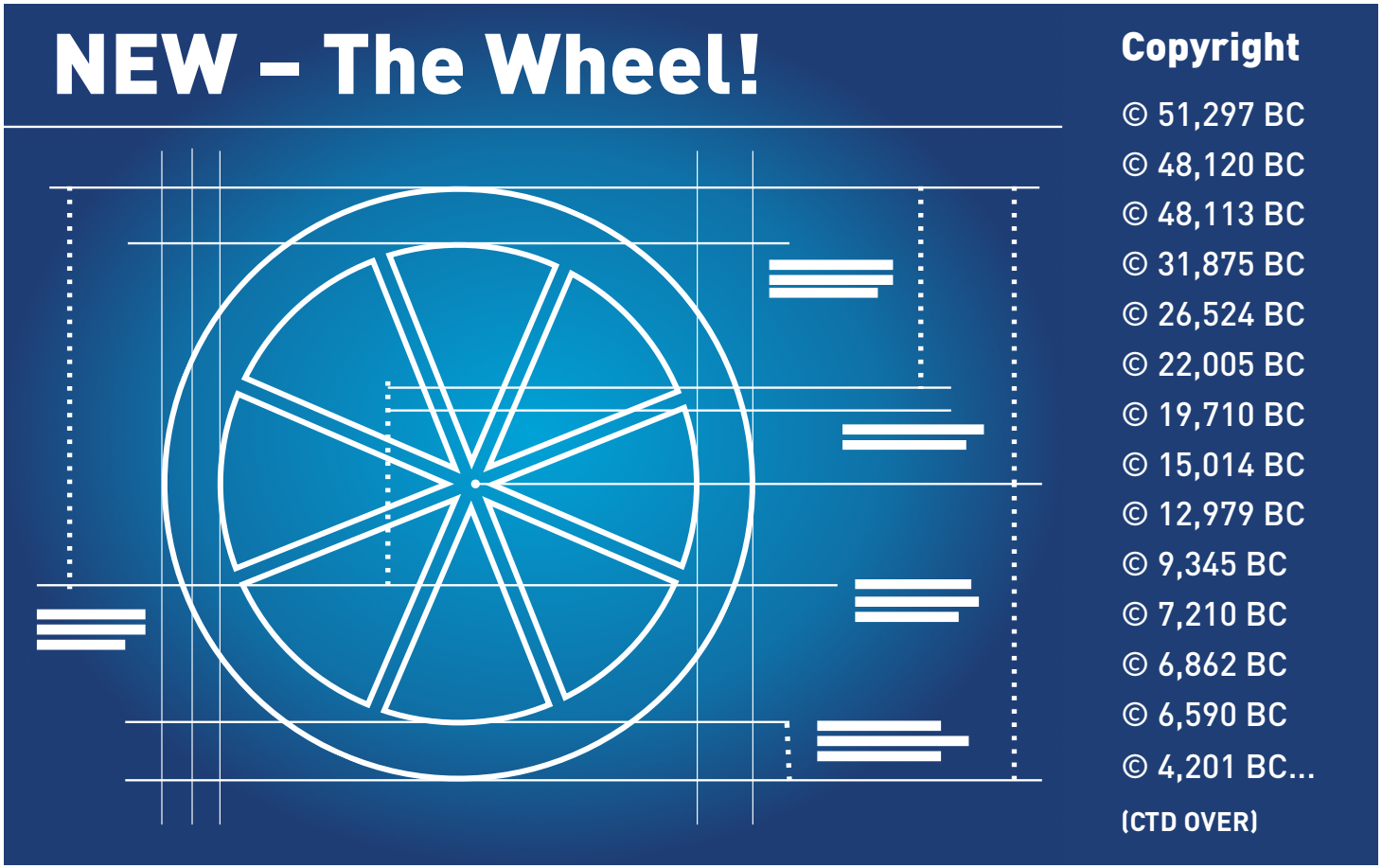
kathik

Antipatterns: Study The Problems

- How can we learn how to do better?
- Study the problem
- Look at failure cases
- Learn from them
- See how to avoid them

AntiPatterns

Not Invented Here



Blame Donkey

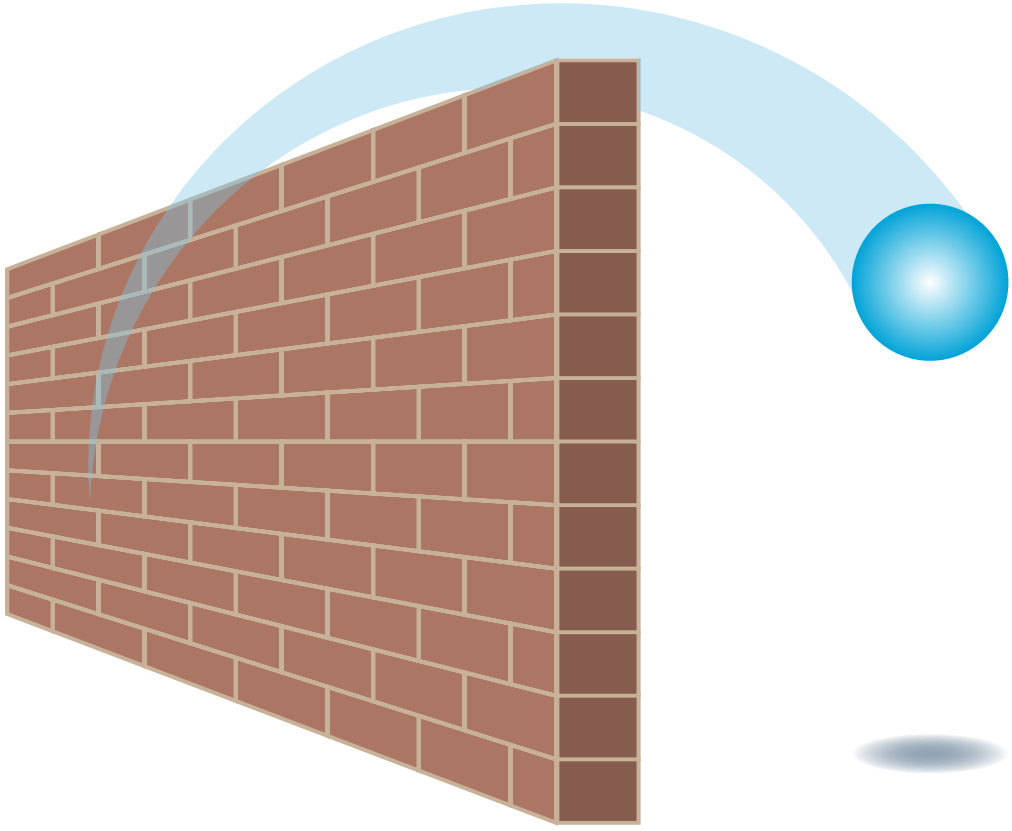
Blame Donkey

- Certain components are always identified as the issue
  - "It's always JMS / Hibernate / ANOTHERTECH"
    - AntiPattern often displayed by management / biz
    - But Technology is not immune

- Reality
  - Insufficient analysis has been done to reach this conclusion

- Solutions
  - Resist pressure to rush to conclusions
  - Perform analysis as normal
  - Communicate the results of analysis to all stakeholders

Throw It Over The Wall



kathik

Throw It Over The Wall

- Teams focus on their immediate domain
  - "It has plenty of unit tests. QA should take over from here"
- Reality
  - Preventable problems pass into production
- Building better cross-team relationships helps
  - Seek to understand what other teams actually do
  - Outages and releases are much less painful
  - Hard to do with remote L1 operations teams

The Python Standard Library

- Is huge!
  - Modules for a vast array of possible uses
  - Check what's in the standard lib first
- Case Study: Regular Expressions

https://docs.python.org/2/library/

Case Study: Regular Expressions

- Python modules are made available using import

```
>>> import re
>>> venkman = "Human sacrifice, dogs and cats living together... mass hysteria!"
>>> mObj = re.search(r'(Cats) ', venkman, re.I)
>>> if mObj:
...     print "Found: ", mObj.group()
...
Found:  cats
```

Case Study: Regular Expressions

- What's changed here?

```
>>> import re
>>> venkman = "Human sacrifice, dogs and cats living together... mass hysteria!”
>>> mObj = re.match(r'(Cats) ', venkman, re.I)
>>> if mObj:
...     print "Match: ", mObj.group()
... else:
...     print "Didn't match"
...
Didn't match
```

Case Study: Regular Expressions

- Support all the grep capabilities
- + : 1 or more
- ? : 0 or 1
- {m} : Exactly m copies
- {m,n} : Between m & n
- .... BUT WHAT ELSE?

Group Exercise B

- Exploring Python's Regular Expressions

- (Groups of 4)

Modern Development Practice

- Testing
- Test Driven Development
  - Designing Testable Code
- Introduction to Agile

Testing

- Broadly 5 categories of testing:
  - Unit Testing
  - Integration Testing
  - System Testing
  - System Integration Testing
  - Performance Testing

Unit Testing

- Unit Test
  - Tests to verify code contained in a class, in isolation.
  - Any non-trivial dependencies should be "test doubles"
  - e.g. Test the `BigDecimal` class
- These should be be fast

Integration Testing

- Integration Test
  - Tests to verify interaction between classes
  - Probably the most common type of test
  - e.g. Test `Currency` class & its interaction with `BigDecimal`
- Integration tests may include systems
  - e.g. in-memory databases
- Misconceptions
  - Frequently confused with unit tests

System Testing

- System Test
  - Tests to verify a running system
  - Start as little of the system as possible
  - Test it's external API
  - Shouldn't rely on any 3rd external system being up at all
  - e.g. Test the accounting system from UI down to datastore

System Testing

- These have disproportionately high value to outsiders
  - They're understandable
  - They're phrased in APIs that people will have seen

- Often written in a BDD manner
  - **Given** X, **When** Y, **Then** Z

System Integration Testing

- System Integration Test
  - Tests to verify a running system
  - Including 3rd party resources
    - e.g. Real database(s), really hitting the internet etc
  - These tests can also be run against the UI of the system
  - e.g. Test the whole accounting system including 3rd party calls

- Thoughtworks Screencast Central
  - Contains some useful tips for System Integration testing

Performance Testing

- Performance Test
  - Specific tests mimicking real system behaviour
  - Designed to find performance bottlenecks
  - Involved writing non-trivial test harnesses

- 4 types
  - Load (e.g. A on-line retailler)
  - Stress (e.g Olympics site)
  - Spike (e.g A ticket sales site, 5 hours before Bieber tickets go on sale)
  - Endurance (e.g. A 24/7 trade reporting system)

Test Driven Development

TDD --> Eliminating fear and uncertainty

Fear makes you tentative.

Fear makes you want to communicate less.

Fear makes you shy away from feedback.

Fear makes you grumpy.

—Kent Beck (co-inventor of the JUnit testing framework)

Other TDD Benefits

- Cleaner code
  - You write only the code you need
- Better design
  - Some developers call TDD "test-driven design"
- Greater flexibility
  - TDD encourages coding to interfaces

Other TDD Benefits

- Fast feedback
  - You learn about bugs now, not in production
- Refactor without Fear
  - Your tests back you up

Red-Green-Refactor

- There are 3 steps to writing TDD code
  - Red (failing test)
  - Green (passing test)
  - Refactored (high quality implementation)

- When testing, don't forget your SOLID principles

Red

- Write a failing test (RED)
  - Fail at the compilation stage OR
  - Fail at the assertion stage

Red-Green-Refactor

- Write an implementation that passes the test (Green)
  - NOTE: This doesn't mean a nice implementation
  - Means others can instantly start integrating with your code

Red-Green-Refactor

- Refactor the implementation and/or test (REFACTOR)
  - Write a higher quality implementation
  - Don't forget you can refactor the test as well!
  - Tests should be first class code in the application