

# Feedback on Lab Exam

---

# *Test a Specification, not an Implementation*

Don't test attributes (implementaton) -- the code may change.

```
def test_bids(self):
    auction = Auction("Sailboat")
    auction.start()
    auction.bid("John", 10000)
    self.assertEqual("Sailboat", auction.name)
    self.assertEqual(10000, auction.bids["John"])

# trivial methods can be verified by code review
def test_active(self):
    auction = Auction("Sailboat")
    auction.start()
    self.assertTrue( auction.active )
```

# Don't test attributes

Test functionality instead of private attributes.

```
def test_init(self):  
    auction = Auction("iPad")  
    self.assertEqual(auction.name, "iPad")  
    self.assertEqual(auction.bids, {"no bids": 0})  
    self.assertEqual(auction.increment, 1)  
    self.assertEqual(auction.active, False)
```

`auction.increment` -- OK to test since it is important and no other way to verify in unit tests.

`auction.name` is not important and can test using `str(auction)`

`auction.bids` is implementation detail -- test behavior instead

`auction.active` is implementation detail -- test using `is_active()`

# Use setUp to Create Test Fixture

Eliminate redundant code

```
def setUp(self):  
    self.auction = Auction("Sailboat")  
    self.auction.start( )  
  
def test_bids(self):  
    #auction = Auction("Sailboat")  
    self.auction.bid("John", 10000)  
    self.assertEqual(10000,  
                     self.auction.best_bid())
```

Must be setUp(self) - NOT setUp(self)  
- NOT setUp(self, name, bid)  
- unittest calls setUp **itself!**

# *Don't "guess" the specification*

---

The Auction class did not specify what happens if bid increment = 0 or < 0, so don't test that.

```
def test_negative_increment(self):  
    with self.assertRaises( AuctionError ):  
        auction = Auction("Sailboat", -1)
```

In a software project, tests help you find  
**ambiguity** and **incomplete** specification.

Open an issue & discuss how to clarify spec.

Try not to *guess* what the spec "should" be.

# Correct Code Should Pass All Tests

1. If code is correct, all tests should **PASS**.

```
cmd> python -m unittest -v auction_test.py
```

```
-----
```

```
Ran 8 tests in 0.001s
```

```
OK
```

2. If code contains **bugs** but no syntax or semantic errors, the **appropriate** test should fail. Or, sometimes "error".

```
test_bid_too_low (auction_test.TestAuction) ...FAIL
```

```
-----
```

```
Ran 8 tests in 0.001s
```

```
FAILED (failures=1)
```

# Should not make these mistakes...

```
def test_bid(self):  
    # what is the error?  
    self.assertTrue(Auction("Sailboat"))  
    Auction("Pizza")  
    Auction.start( )  
    Auction.bid("Jim", 100)  
  
def test_best_bid(self):  
    a = Auction("ISP Final Exam")  
    a.start()  
    a.bid("Hacker", 10000)  
    self.assertEqual( a, 1000)
```

# Respect Object Encapsulation

O-O programs **encapsulate** object information as **attributes**, and protect it ...but not in Python.

Objects **expose** information using **methods**.

Avoid directly accessing an object's attributes!

A class should be free to **change** its implementation, as long as the **interface** (methods) behaves the same way.

```
def setUp(self) :  
    self.auction = Auction("TDD in Python")  
    # BAD - directly setting an attribute  
    self.auction.increment = 2
```



# Test Sensitivity

**Sensitivity** - ability to detect a flaw.

In scientific research: *probability that a test will correctly detect a condition (when it is present).*

## What is wrong with this test?

```
def test_bid_invalid_parameters(self):  
    with self.assertRaises( ValueError ):  
        # both of these should raise error  
        self.auction.bid("", 100)  
        self.auction.bid("Joe", 0)
```

# What is wrong with this test?

Suppose the code allows bidder name to be "" (a bug), but `auction.bid("Joe",0)` raises `ValueError` (amount = 0).

The 2nd statement causes this test to *always* pass.

```
def test_bid_invalid_parameters(self):  
    with self.assertRaises( ValueError ):  
        # both of these should raise error  
        self.auction.bid("", 100)  
        self.auction.bid("Joe", 0)
```

# Another Example

Suppose:

#1. auction allows bidding when stopped (**incorrect**)

#2. auction raises AuctionError if bid too low (**correct!**)

Will this test detect the bug?

```
def test_bid_when_auction_stopped(self):  
    self.auction.bid("Good", 600)  
    self.auction.stop()  
    with self.assertRaises( AuctionError ):  
        self.auction.bid("Bug", 1000)    #1  
        self.auction.bid("Cheap", 500)  #2
```

# Can You Summarize?

If a "with self.assertRaises" contains many statements, the test will pass when?

[ ] all statements throw the expected exception

[ ] any statement throws the expected exception

```
def test_bid_when_auction_stopped(self):  
  
    with self.assertRaises( AuctionError ):   
        statement1( )  
        statement2( )  
        statement3( )
```

# Tests Should Help Diagnose Problem

Tests should be **specific** -- not one big test.

```
def test_normal_bidding(self):  
    # auction created and started in setUp()  
    auction.bid("Ant", 100)  
    with self.assertRaises(AuctionError):  
        auction.bid("Bird", 100) #identical bid  
  
    # test bidding not allowed when stopped  
    with self.assertRaises(AuctionError):  
        auction.stop()           # wrong location. Why?  
        auction.bid("Dog", 1000)
```

These tests should be  
in separate methods.

# Focused tests help to identify problem

Use descriptive names, test a single kind of behavior

```
def test_normal_bidding(self):
    self.auction.bid("Ant", 1)    # minimum bid
    self.auction.bid("Bat", 2)    # minimum increase
    self.auction.bid("Cat", 50.5) # decimals ok?
    self.auction.bid("Ant", 51.5) # bid again ok?
    self.assertEqual(51.1, auction.best_bid())

def test_low_bid_throws_exception(self):
    self.auction.bid("Good guy", 10)
    with self.assertRaises(AuctionError):
        self.auction.bid("Cheap", 9.99) # too low
    with self.assertRaises(AuctionError):
        self.auction.bid("Cheap", 10)   # same bid
```

# Test What Could *Reasonable* Fail

We can't test *everything*, so don't test for *stupid code*.

```
def test_bid_must_be_positive(self):  
    with self.assertRaises(ValueError):  
        auction.bid("Ant", 0)  
    with self.assertRaises(ValueError):  
        auction.bid("Bird", -0.1)  
    with self.assertRaises(ValueError):  
        auction.bid("Cat", -20)
```

**#OVERKILL** if above tests pass, these will, too

```
    with self.assertRaises(ValueError):  
        auction.bid("Dog", -100)  
        auction.bid("elephant", -500)  
        auction.bid("Frog", -10000000)
```

# Good Student Codes

---

Many short, specific tests with descriptive names:

Chayathon - 12 tests

Tharathorn - 12 tests

Mai - 31 tests, but some are "out of spec"

and several others.



# Reset Polls Voting

[Back to Questions](#)

## Best Django Applications

Poll	Total Votes	Reset Votes?
What is the best programming language?	11000	<a href="#">Reset Votes</a>
What is the WORST web framework?	34	<a href="#">Reset Votes</a>
What is the coolest university in Thailand?	3621	<a href="#">Reset Votes</a>
What place is shown in this background image?	13	<a href="#">Reset Votes</a>

#1 Tharathorn  
#2 Chananchida

2019 | Made with ♥ by [Tharathorn Bunrattanasathian](#)

MADE WITH **django**

Made with **BULMA**

# total\_votes() common mistakes

---

1. Using a query for Question instead of self.
2. Using a query for Choices instead of self.choice\_set.
3. Using an **attribute** to sum votes instead of local var.
4. Poor variable names. Misuse of plurality.

# Querying for Question instead of using `self`

---

What object is being retrieved by this query?

```
# In models.Question class
def total_votes(self):
    question = Question.objects.get(id=self.id)
```

# *Data* is same, but is it same object?

Which **message** will be logged?

```
# In models.Question class
def total_votes(self):
    question = Question.objects.get(id=self.id)

    if question == self:
        log.info( "Its me!" )
    else:
        log.info( "Born again.  A new me." )
```

This is an important feature of ORM:

If we retrieve the same thing using different queries,  
does the ORM return the same object (instance) or distinct copies?

# This could bring a server to its knees

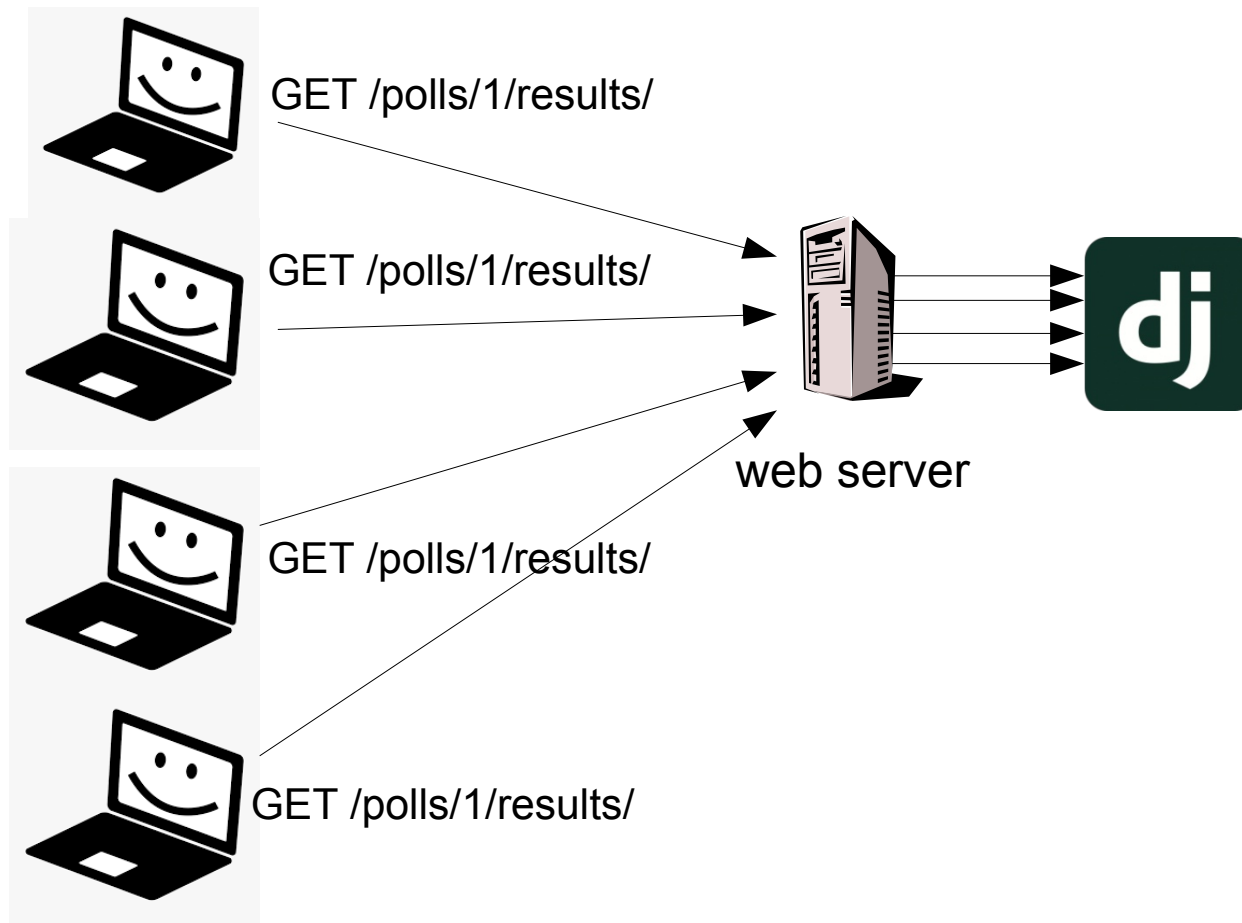
Explain how this code could use a **lot** of memory & I/O.

Suppose Facebook is using your polls app for custom polls.

```
# In models.Question class
def total_votes(self):
    # get the choices
    choices = Choice.objects.all( )
    # sum only votes for this question
    total = 0
    for choice in choices:
        if choice.question_id == self.id:
            total += choice.votes
    return total
```

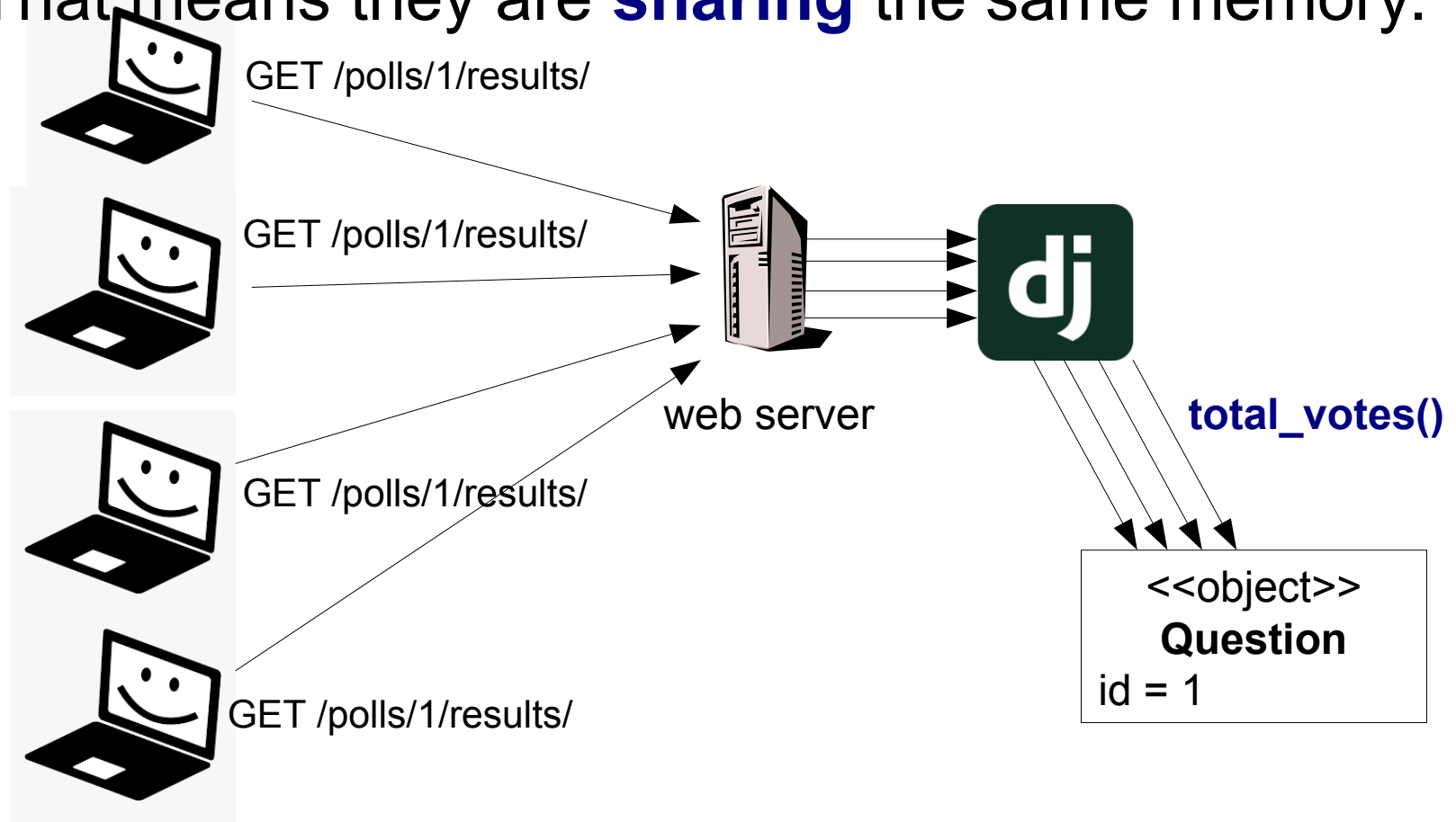
# Using attribute instead of local vars

A web application may have many simultaneous visitors. Some of them may invoke same code.



# Requests are handled using threads

Each request is usually handled in a **separate thread** of the **same process** (for efficiency). That means they are **sharing** the same memory.



# Using attribute instead of local vars

If many **simultaneous** visitors (separate threads) invoke `question1.total_votes( )`,  
how might this code **produce the wrong results?**

```
# In models.Question class
def total_votes(self):
    self.count = 0
    for self.choice in self.choice_set.all():
        self.count += self.choice.votes
    return self.count
```

*There are 2 bugs.*



# Good web app design

1. Use local vars for computation
  2. Be aware of "thread (un)safe" behavior
- Most code provided by Django is thread safe.

```
# In models.Question class
def total_votes(self):
    total = 0 # local var, descriptive name
    for choice in self.choice_set.all():
        total += choice.votes
    return total
```

Each time a method is invoked, it gets its **own memory** for local vars.  
So, multiple threads can safely call same method at the same time.

# Misleading variables names

Are these variable names descriptive?

- Use **descriptive** names
- Correct **plurality**: list or set name should be **plural**

```
# In models.Question class

def total_votes(self):
    q =
    Question.objects.get(id=self.pk).choice_set.all()
    vote = 0
    for choice in q:
        vote += q.votes
    return vote
```

# Code is too long

---

Can you reduce it to **zero** lines?

```
# In models.Question class

def total_votes(self):
    return sum(choice.votes
               for choice in self.choice_set.all())
```

# Common Mistakes in urls.py

---

1. Mixing `reset_index` and `reset` (one poll) views.

2. Duplicate name for views.

```
path("reset/", views.reset_index, name="reset"),  
path("reset/<int:id>/", views.reset, name="reset"),
```

3. Forget trailing "/" in URL.

4. Be careful! extra space at end of URL

```
path('reset/ ', views.reset_index, ...)
```

Surprisingly, Django includes the space in required URL!

```
http://localhost:8000/polls/reset/%20
```

# Django Template Filters

Django template filters let you add custom behavior or formatting in templates.

In a template, apply a filter with 2 commands:

```
{% load filterapp_name %} written only once
```

```
You are {{ 3 | ordinal }} in the queue.
```



```
You are 3rd in the queue.
```

# Humanize `intcomma` filter

**Humanize** filters convert numbers and dates into text format. To use it:

1. In `settings.py` add:

```
INSTALLED_APPS = [  
    ...,  
    django.contrib.humanize,  
]
```

2. in your poll `reset_index` template add:

```
{% load humanize %}  
<table>  
{% for question in question_list %}  
...  
    <td>{{question.total_votes | intcomma}}</td>
```

# What place is shown in this background image?

- ☐ Atacama Desert in Chile
- ☐ Grand Canyon in Arizona, USA
- ☐ Monte Desert in Argentina
- ☐ Mount Sharp on planet Mars
- ☐ Rocky Mountains in western USA
- ☐ Tabernas Desert in Spain

Vote

Go Back

