refactoring101 Documentation

Release 0.1

Serdar Tumgoren (AP) and Jeremy Bowers (NPR)

March 10, 2018

CHAPTER 1

Inspiration

“Complexity kills.” ~ *Ray Ozzie*

“The art of simplicity is a puzzle of complexity.” ~ *Douglas Horton*

”...you’re not refactoring; you’re just changing shit.” ~ *Hamlet D’Arcy*

CHAPTER 2

Overview

So you’ve written a few scripts that get the job done. The machine does your bidding, but the initial euphoria has worn off.

Bugs are cropping up. Data quirks are creeping in. Duplicate code is spreading like a virus across projects, or worse, inside the same project. Programs aren’t failing gracefully.

There *must* be a better way, but the path forward is not clear.

If you’re like us and have had that itchy feeling, this tutorial is for you.

After you’ve mastered the basics of writing code, you need to understand how to *design* programs. The goal of this tutorial is to bridge that gap. We’ll demonstrate how to use Python language features – functions, modules, packages and classes – to organize code more effectively. We also introduce unit testing as a strategy for writing programs that you can update with confidence.

The overarching theme: **As a program grows in size, writing readable code with tests can help tame complexity and keep you sane.**

CHAPTER 3

How To Use This Tutorial

The Github repo contains code samples demonstrating how to transform a complex, linear script into a modular, easier-to-maintain package. The code was written as a reference for Python classes at NICAR 2014 and 2015, but can also be used as a stand-alone tutorial.

We use a small, fake set of election results for demonstration purposes. Project code evolves through four phases, each contained in a numbered *elex* directory in the code repo.

Each section ends with questions and/or exercises. These are the most important part of the tutorial. You're supposed to wrestle with these questions and exercises. Tinker with the code; break the code; write alternative versions of the code. Then email me (it's not Jeremy's fault) and explain why the code sucks. Then read your own code from six months ago ;)

CHAPTER 4

Questions and Resources

Still have questions? Check out the *FAQ*, as well the *Resources* page for wisdom from tribal elders.

CHAPTER 5

Table of Contents

(Re)designing code

Code smells

We begin with a single, linear script in the elex1/ directory. Below are a few reasons why this code smells (some might say it reeks):

- It's hard to understand. You have to read the entire script before getting a full sense of what it does.
- It's hard to debug when something goes wrong.
- It's pretty much impossible to test, beyond eye-balling the output file.
- None of the code is reusable by other programs.

Hacking-It-Out-As-You-Go

Scripts like this are often born when a programmer dives immediately into implementing his code. He sees the end goal – "summarize election data" – and gets right to it, hacking his way through each step of the script until things "work".

The hack-it-out-as-you-go approach can certainly produce working code. But unless you're extremely disciplined, this process can also yield spaghetti code – a jumble of hard-to-decipher and error-prone logic that you fear changing.

So, how do we avoid spaghetti code? *By choosing to have lots of small problems instead of one big problem.*

Lots of small problems

A key step in the art of designing code is hitting the breaks up front and spending a few minutes thinking through the problem at hand. Using this approach, you'll quickly discover that you don't really have one big problem (*"summarize some election data"*) but a series of small problems:

- Download election data
- Parse election data
- Calculate candidate vote totals and determine winners
- Create a summary spreadsheet

Each of those smaller problems, in turn, can often be decomposed into a series of smaller steps, some of which don't become clear until you've started writing the code.

But it's critical at this phase to *NOT* start writing code!!! You will be tempted, but doing so will switch your brain from "design mode" to the more myopic "code" mode (it's a thing). Trust in your ability to implement the code when the time is right (we promise, you'll figure it out), and instead grant yourself a few minutes of freedom *to design the code*.

If you just can't resist implementing code as you design, then close your laptop and go old school with pen and paper. A mind-map or flow-chart is a great way to hash out the high-level design and flow of your program. Or if you're lucky enough to have a whiteboard, use that to sketch out the initial steps of your program.

Some folks also like writing pseudocode, though beware the siren's call to slip back into implementing "working" code (Python in particular makes this extremely easy).

> *Fun fact*: Jeremy and I are so enthusiastic about whiteboarding that we once sketched out a backyard goat roast on an office wall (said design was never implemented).

Shred this code (on paper)

In this tutorial, we already have some ready-baked spaghetti code for you to slice and dice into smaller components.

We encourage you to print the code on paper – yes, dead trees! – and use a marker to group code bits into separate functions. As you to try to make sense of the logic and data structures, it's a good idea to reference the source data.

This exercise is intended to familiarize you with the data and the mechanics of the code, and get your creative juices flowing. As you read the code, think about which sections of logic are related (perhaps they process some piece of data, or apply a process to a bunch of data in a loop).

Use circles, brackets, arrows – whatever marks on paper you need to group together such related bits of code. Then, try to give them *meaningful names*. These names will become the functions that wrap these bits of logic.

Naming things is hard, and can become *really hard* if a function is trying to do too many things. If you find yourself struggling to come up with a clear function name, ask yourself if breaking down the section of code into even smaller parts (say two or three functions instead of one) would make it easier to assign a clear and meaningful name to each function.

Finally, spend some time thinking about how all these new bits of code will interact. Will one of the functions require an input that comes from another function? This orchestration of code is typically handled in a function called main, which serves as the entry point and quarterback for the entire script.

Keep in mind there's no "right" approach or solution here. The overarching goal is to improve the *readability of the code*.

> Whether you resort to pseudocode, a whiteboard, or simple pen-on-paper, the point is to stop thinking about *how to implement the code* and instead focus on *how to design the program*.

Once the code design process is complete, try implementing the design. Ask yourself how this process compared to prior efforts to write a script (or unravel someone else's code). Was it easier? Harder? Is the end product easier to read and understand?

In the next section, you'll see our pass at the same exercise, and learn how to further improve this script by organizing functions into new source files.

Questions

- What are unit tests?
- Can you identify three sections of logic that could be unit tested?

- What are modules?
- What are packages?

Exercises

- Slice up this code into a bunch of functions, where related bits of logic are grouped together. Do function names accurately reflect what they actually do? If not, how could you rename functions and/or re-organize the code to clarify the purpose of each function?
- Compare your revised script to our version. What's similar? What's different? Explain 5 things you like or dislike about each and why.
- Write a unit test for one or more functions extracted from this module.

Function breakdown and Testing Intro

In the elex2/ directory, we've chopped up the original election_results.py code into a bunch of functions and turned this code directory into a package by adding an __init__.py file.

We've also added a suite of tests. This way we can methodically change the underlying code in later phases, while having greater confidence that we haven't corrupted our summary numbers.

> **Note**: We can't stress this step enough: Testing existing code is *THE* critical first step in refactoring.

If the code doesn't have tests, write some, at least for the most important bits of logic. Otherwise you're just changing shit.

Fortunately, our code has a suite of unit tests for name parsing and, most importantly, the summary logic.

Python has built-in facilities for running tests, but they're a little raw for our taste. We'll use the nose library to more easily run our tests:

```
nosetests -v tests/test_parser.py
# or run all tests in the tests/ directory
nosetests -v tests/*.py
```

Observations

At a high level, this code is an improvement over *elex1/*, but it could still be much improved. We'll get to that in Phase 3, when we introduce modules and packages.

Questions

- What is __init__.py and why do we use it?
- In what order are test methods run?
- What does the TestCase *setUp* method do?
- What other TestCase methods are available?

Exercises

- Install nose and run the tests. Try breaking a few tests and run them to see the results.
- List three ways this code is better than the previous version; and three ways it could be improved.
- Organize functions in *election_results.py* into two or more new modules. (Hint: There is no right answer here. Naming things is hard; aim for directory and file names that are short but meaningful to a normal human).

Modules, packages, oh my!!

In this third phase, we chop up our original *election_results.py* module into a legitimate Python package. The new directory structure is (hopefully) self-explanatory:

```
-- elex3
|   -- __init__.py
|   -- lib
|   |   -- __init__.py
|   |   -- parser.py
|   |   -- scraper.py
|   |   -- summary.py
|   -- scripts
|   |   -- save_summary_to_csv.py
|   -- tests
|       -- __init__.py
|       -- sample_results.csv
|       -- sample_results_parsed.json
|       -- sample_results_parsed_tie_race.json
|       -- test_parser.py
|       -- test_summary.py
```

- `lib/` contains re-usable bits of code.
- `scripts/` contains...well..scripts that leverage our re-usable code.
- `tests/` contains tests for re-usable bits of code and related fixtures.

Note that we did not change any of our functions. Mostly we just re-organized them into new modules, with the goal of grouping related bits of logic in common-sense locations. We also updated imports and "namespaced" them of our own re-usable code under *elex3.lib*.

Here's where we start seeing the benefits of the tests we wrote in the *elex2* phase. While we've heavily re-organized our underlying code structure, we can run the same tests (with a few minor updates to *import* statements) to ensure that we haven't broken anything.

> **Note**: You must add the *refactoring101* directory to your *PYTHONPATH* before any of the tests or script will work.

```
$ cd /path/to/refactoring101
$ export PYTHONPATH=`pwd`:$PYTHONPATH

$ nosetests -v elex3/tests/*.py
$ python elex3/scripts/save_summary_to_csv.py
```

Check out the results of the *save_summary_to_csv.py* command. The new *summary_results.csv* should be stored *inside* the *elex3* directory, and should match the results file produced by *elex2/election_results.py*.

Questions

- Do you *like* the package structure and module names? How would you organize or name things differently?
- Why is it necessary to add the *refactoring101/* directory to your PYTHONPATH?
- What are three ways to add a library to the PYTHONPATH?
- What is a class? What is a method?
- What is an object in Python? What is an instance?
- What is the **init** method on a class used for?
- What is *self* and how does it relate to class instances?

Exercises

- Look at the original results data, and model out some classes and methods to reflect "real world" entities in the realm of elections.
- Examine functions in *lib/* and try assigning three functions to one of your new classes.
- Try extracting logic from the *summarize* function and re-implement it as a method on one of your classes.

OO Design and Refactoring

Real World Objects

In this section, we create classes that model the real world of elections. These classes are intended to serve as more intuitive containers for data transformations and complex bits of logic currently scattered across our application.

The goal is to hide complexity behind simple interfaces.

We perform these refactorings in a step-by-step fashion and attempt to write tests before the actual code.

So how do we start modeling our domain? We clearly have races and candidates, which seem like natural...wait for it... "candidates" for model classes. We also have county-level results associated with each candidate.

Let's start by creating Candidate and Race classes with some simple behavior. These classes will eventually be our workhorses, handling most of the grunt work needed to produce the summary report. But let's start with the basics.

Candidates, of course

An election has a set of races, each of which have candidates. So a *Candidate* class is a natural starting point for data modeling.

What basic characteristics does a candidate have in the context of the source data?

Name, party and county election results jump out.

A candidate also seems like a natural place for data transforms and computations that now live in *lib/parser.py* and *lib/summary.py*:

- candidate name parsing
- total candiate votes from all counties

- winner status

Before we migrate the hard stuff, let's start with the basics.

We'll store new election classes in a *lib/models.py* (Django users, this should be familiar). We'll store tests for our new classes in *test_models.py* module.

Now let's start writing some test-driven code!

Name Parsing

The Candidate class should be responsible for parsing a full name into first and last names (remember, candidate names in our source data are in the form *(Lastname, Firstname*).

- Create *elex4/tests/test_models.py* and add test for Candidate name parts
- Run test; see it fail
- Write a Candidate class with a private method to parse the full name

  Note: You can cheat here. Recall that the name parsing code was already written in *lib/parser.py*.

- Run test; see it pass

Observations

In the refactoring above, notice that we're not directly testing the *name_parse* method but simply checking for the correct value of the first and last names on candidate instances. The *name_parse* code has been nicely tucked out of sight. In fact, we emphasize that this method is an *implementation detail* – part of the *Candidate* class's internal housekeeping – by prefixing it with two underscores.

This syntax denotes a private method that is not intended for use by code outside the *Candidate* class. We're restricting (though not completely preventing) the outside world from using it, since it's quite possible this code wil change or be removed in the future.

> More frequently, you'll see a single underscore prefix used to denote private methods and variables. This is fine, though note that only the double underscores trigger the name-mangling intended to limit usage of the method.

Questions

- In order to migrate functions to methods on the Candidate class, we had to make the first parameter in each method *self*. Why?

County results

In addition to a name and party, each *Candidate* has county-level results. As part of our summary report, county-level results need to be rolled up into a racewide total for each candidate. At a high level, it seems natural for each candidate to track his or her own vote totals.

Below are a few basic assumptions, or requirements, that will help us flesh out vote-handling on the *Candidate* class:

- A candidate should start with zero votes
- Adding a vote should increment the vote count
- County-level results should be accessible

With this basic list of requirements in hand, we're ready to start coding. For each requirement, we'll start by writing a (failing) test that captures this assumption; then we'll write code to make the test pass (i.e. meet our assumption).

1. Add test to ensure *Candidate*'s initial vote count is zero

   Note: We created a new *TestCandidateVotes* class with a *setUp* method that lets us re-use the same candidate instance across all test methods. This makes our tests less brittle – e.g., if we add a parameter to the *Candidate* class, we only have to update the candidate instance in the *setUp* method, rather than in every test method (as we will have to do in the *TestCandidate* class)

1. Run test; see it fail
2. Update *Candidate* class to have initial vote count of zero
3. Run test; see it pass

Now let's add a method to update the candidate's total vote totals for each county result.

1. Add test for *Candidate.add_votes* method
2. Run test; see it fail
3. Create the *Candidate.add_votes* method
4. Run test; see it pass

Finally, let's stash the county-level results for each candidate. Although we're not using these lower-level numbers in our summary report, it's easy enough to add in case we need them for down the road.

1. Create test for county_results attribute
2. Run test; see it fail
3. Update *Candidate.add_votes* method to store county-level results
4. Run test; see it pass

Questions

Exercises

- The *Candidate.add_votes* method has a potential bug: It can't handle votes that are strings instead of proper integers. This bug might crop up if our parser fails to convert strings to integers. Write a unit test to capture the bug, then update the method to handle such "dirty data" gracefully.

Races have Candidates!

With the basics of our *Candidate* class out of the way, let's move on to building out the *Race* class. This higher-level class will manage updates to our candidate instances, along with metadata about the race itself such as election date and office/district.

Recall that in *elex3*, the *lib/parser.py* ensured that county-level results were assigned to the appropriate candidate. We'll now migrate that logic over to the *Race* class, along with a few other repsonsibilities:

- Tracking overall vote count for the race
- Updating candidates with new county-level votes
- Determining which candidate, if any, won the race

Metadata and Total votes

The *Race* class keeps a running tally of all votes. This figure is the sum of all county-level votes received by individual candidates.

Let’s build out the *Race* class with basic metadata fields and an *add_result* method that updates the total vote count.

This should be pretty straight-forward, and you’ll notice that the tests mirror those used to perform vote tallies on *Candidate* instances.

```
# Don't forget to import Race from elex4.lib.models at the top of your test module!

class TestRace(TestCase):

    def setUp(self):
        self.smith_result = {
            'date': '2012-11-06',
            'candidate': 'Smith, Joe',
            'party': 'Dem',
            'office': 'President',
            'county': 'Fairfax',
            'votes': 2000,
        }
        self.race = Race("2012-11-06", "President", "")

    def test_total_votes_default(self):
        "Race total votes should default to zero"
        self.assertEquals(self.race.total_votes, 0)

    def test_total_votes_update(self):
        "Race.add_result should update racewide vote count"
        self.race.add_result(self.smith_result)
        self.assertEquals(self.race.total_votes, 2000)
```

Go ahead and run those tests and watch them fail.

Now let’s build out our initial *Race* class with an *add_result* method to make the tests pass.

```
class Race(object):

    def __init__(self, date, office, district):
        self.date = date
        self.office = office
        self.district = district
        self.total_votes = 0

    def add_result(self, result):
        self.total_votes += result['votes']
```

Candidate Bookkeeping

In earlier phases of the project, the parser code ensured that county-level results were grouped with the appropriate, unique candidate in each race. If you recall, those county results were stored in a list for each candidate:

```
# elex3.lib.parser.py

def parse_and_clean
```

```
# ... snipped...

    # Store county-level results by slugified office and district (if there is one),
      # then by candidate party and raw name
      race_key = row['office']
      if row['district']:
          race_key += "-%s" % row['district']

      # Create unique candidate key from party and name, in case multiple candidates have same
      cand_key = "-".join((row['party'], row['candidate']))

      # Get or create dictionary for the race
      race = results[race_key]

      # Get or create candidate dictionary with a default value of a list; Add result to the list
      race.setdefault(cand_key, []).append(row)
```

We now have Candidate classes that manage their own county results. But we need to migrate the bookkeeping of Candidate instances from the parser code to the *Race* class. Specifically, we need create a new Candidate instance or fetch a pre-existing instance, as appropriate, for each county result.

Let's start by adding a test to our *TestRace* class that ensures we're updating a single candiate instance, rather than accidentally creating duplicate instances.

```
class TestRace(TestCase):

    # ... snipped ...

    def test_add_result_to_candidate(self):
        "Race.add_result should update a unique candidate instance"
        # Add a vote twice. If it's the same candidate, vote total should be sum of results
        self.race.add_result(self.smith_result)
        self.race.add_result(self.smith_result)
        cand_key = (self.smith_result['party'], self.smith_result['candidate'])
        candidate = self.race.candidates[cand_key]
        self.assertEquals(candidate.votes, 4000)
```

Run that test and watch it fail. You'll notice we have a new *candidates* attribute that is a dictionary. This is pretty much the same approach we used in earlier phases, where we stored candidate data by a unique key. However, instead of using a slug, we're now using tuples as keys.

> Accessing *candidate* data directly in this way is a code smell, and it could be argued that we should also write a candidate lookup method. We'll leave that as an exercise.

Now let's update the *Race* class and its *add_result* method to make the test pass.

```
class Race(object):

    def __init__(self, date, office, district):
        # .... snipped ....
        # We add the candiddates dictionary
        self.candidates = {}

    def add_result(self, result):
        self.total_votes += result['votes']
        # Below lines
        candidate = self.__get_or_create_candidate(result)
        candidate.add_votes(result['county'], result['votes'])
    # Private methods
    def __get_or_create_candidate(self, result):
        key = (result['party'], result['candidate'])
        try:
            candidate = self.candidates[key]
        except KeyError:
            candidate = Candidate(result['candidate'], result['party'])
            self.candidates[key] = candidate
        return candidate
```

Above, the bulk of our work is handled by a new private method called __get_or_create_candidate. This method attempts to fetch a pre-existing *Candidate* instance or creates a new one and adds it to the dictionary, before returning the instance.

Once we have the correct instance, we call its *add_votes* method to update the vote count and add the result to that candidate's county results list.

Our test verifies this by calling the *add_result* method twice and then checking the candidate instance's vote count to ensure the vote count is correct.

> Testing purists may point out that we've violated the principle of test isolation, since this unit test directly accesses the candidate instance and relies on its underlying vote tallying logic. There are testing strategies and tools, such as mocks, to help avoid or minimize such *tight coupling* between unit tests. For the sake of simplicity, we'll wave our hand at that issue in this tutorial and leave it as a study exercise for the reader.

Assigning Winners

We're now ready for the last major piece of the puzzle, namely, migrating the code that determines race winners. This logic was previously handled in the *summary* function and its related tests.

```
# elex3/lib/summary.py

# ... snipped ....

    # sort cands from highest to lowest vote count
    sorted_cands = sorted(cands, key=itemgetter('votes'), reverse=True)

    # Determine winner, if any
    first = sorted_cands[0]
    second = sorted_cands[1]

    if first['votes'] != second['votes']:
        first['winner'] = 'X'

# ... snipped ....
```

We'll migrate our tests and apply some minor updates to reflect the fact that we're now storing data in Candidate and Race classes, rather than nested dictionaries and lists.

> It's important to note that while we're modifying the test syntax to accommodate our new objects, we're not changing the *substance* of the tests.

First, let's add an extra sample result to the *setUp* method to support each test.

```
# elex4/tests/test_models.py

class TestRace(TestCase):
    def setUp(self):


      # ... snipped ....

        self.doe_result = {
            'date': '2012-11-06',
            'candidate': 'Doe, Jane',
            'party': 'GOP',
            'office': 'President',
            'county': 'Fairfax',
            'votes': 1000,
        }
```

Next, let's migrate the winner, non-winner and tie race tests from *elex3/tests/test_summary* to the *TestRace* class in *elex4/tests/test_models.py*.

```
class TestRace(TestCase):

      # ... snipped ....

    def test_winner_has_flag(self):
        "Winner flag should be assigned to candidates with most votes"
        self.race.add_result(self.doe_result)
        self.race.add_result(self.smith_result)
        # Our new method triggers the assignment of the winner flag
        self.race.assign_winner()
        smith = [cand for cand in self.race.candidates.values() if cand.last_name == 'Smith'][0]
        self.assertEqual(smith.winner, 'X')

    def test_loser_has_no_winner_flag(self):
        "Winner flag should not be assigned to candidate that does not have highest vote total"
        self.race.add_result(self.doe_result)
        self.race.add_result(self.smith_result)
        self.race.assign_winner()
        doe = [cand for cand in self.race.candidates.values() if cand.last_name == 'Doe'][0]

    def test_tie_race(self):
        "Winner flag should not be assigned to any candidate in a tie race"
        # Modify Doe vote count to make it a tie for this test method
        self.doe_result['votes'] = 2000
        self.race.add_result(self.doe_result)
        self.race.add_result(self.smith_result)
        self.race.assign_winner()
        for cand in self.race.candidates.values():
            self.assertEqual(cand.winner, '')
```

These tests mirror the test methods in *elex3/tests/test_summary.py*. We've simply tweaked them to reflect our class-based apprach and to exercise the new *Race* method that assigns the winner flag.

We'll eventually delete the duplicative tests in *test_summary.py*, but we're not quite ready to do so yet.

First, let's make these tests pass by tweaking the *Candidate* class and implementing the *Race.assign_winner* method:

```
# elex4/lib/models.py

class Candidate(object):

    def __init__(self, raw_name, party):
        # ... snipped...

        # Add a new winner attribute to candidate class with empty string as default value
        self.winner = ''


class Race(object):

    # ... snipped...

    def assign_winner(self):
        # Sort cands from highest to lowest vote count
        sorted_cands = sorted(self.candidates.values(), key=attrgetter('votes'), reverse=True)

        # Determine winner, if any
        first = sorted_cands[0]
        second = sorted_cands[1]

        if first.votes != second.votes:
            first.winner = 'X'
```

Above, notice that we added a default *Candidate.winner* attribute, and a *Race.assign_winner* method. The latter is nearly a straight copy of our original winner-assignment logic in the *summarize* function. The key differences are:

- We're calling *self.candidate.values()* to get a list of *Candidate* instances, since these are now stored in a dictionary.
- We're using *attrgetter* instead of *itemgetter* to access the candidate's vote count for purposes of sorting. This is necessary, of course, because we're now sorting by the value of an instance attribute rather than the value of a dictionary key.
- We're accessing the *votes* attribute on candidate instances rather than performing dictionary lookups.

Enter stage left - Races and Candidates

The *Candidate* and *Race* classes now encapsulate our core logic. It's time to put these classes to work.

This is the step we've been waiting for – where we simplify the parser ond summary code by outsourcing complex logic to simple domain models (i.e. *Candidate* and *Race* classes).

Major code updates such as this feel like changing the engine on a moving car: It's scary, and you're never quite sure if an accident is waiting around the corner. Fortunately, we have a suite of tests that let us apply our changes and quickly get feedback on whether we broke anything.

Let's start by swapping in the *Race* class in the parser code, the entry point of our application. The *Race* class replaces nested dictionaries and lists.

Update Parser

```
def parse_and_clean():

    # ... snipped ...

    results = {}

    # Initial data clean-up
    for row in reader:
        # Convert votes to integer
        row['votes'] = int(row['votes'])

        # Store races by slugified office and district (if there is one)
        race_key = row['office']
        if row['district']:
            race_key += "-%s" % row['district']

        try:
            race = results[race_key]
        except KeyError:
            race = Race(row['date'], row['office'], row['district'])
            results[race_key] = Race

        race.add_result(row)

    # ... snipped ...
```

Here are the list of changes:

- Delete the candidate name parsing code
- Simplify results storage and use try/except to get/create Race instances
- Update Race and, by extension, candidate vote totals, by calling *add_result* on *Race* instance.

Before porting the *summarize* function to use this new input, let's update the parser tests and ensure evertyhing runs correctly. We'll tweak our test to use dotted-attribute notation instead of dictionary lookups, to reflect the new class-based approach.

```
# elex4/tests/test_parser.py

class TestParser(TestCase):

    def test_name_parsing(self):
        "Parser should split full candidate name into first and last names"
        race = results['President']
        smith = [cand for cand in race.candidates.values() if cand.last_name == 'Smith'][0]
        # Below lines changed from dictionary access
        self.assertEqual(smith.first_name, 'Joe')    # formerly, smith['first_name']
        self.assertEqual(smith.last_name, 'Smith')   # formerly, smith['last_name']
```

Now run the tests:

```
nosetests -v elex4/tests/test_parser.py
```

The updated *parse_and_clean* function is easier to read and maintain than its original version, but it could still be much improved. For instance, we could easily hide the race-key logic and type conversion of votes inside the *Race* class.

We could also transform the function into a class, and encapsulate the get/create logic for *Race* instances in a private method, similar to the *Race.__get_or_create_candidate* method.

We'll leave such refactorings as exercises for the reader.

Exercises

- The *parse_and_clean* function, though simplified, still has too much cruft. Perform the following refactorings:

- Move code that converts votes to an integer inside the *Race* class
- Create a *Race.key* property that encapsulates this logic, and remove it from the parser function
- Simplify the return value of *parse_and_clean* to only return a list of *Race* instances, rather than a dictionary. This will require also refactoring the *summarize* function
- Refactor the *parse_and_clean* function into a *Parser* class with a private *__get_or_create_race* method.

Update Summary

Refactoring the *summarize* function is a bit trickier than the parser code, since we plan to change the input data for this function. Recall that the parser code now returns a dict of *Race* instances, rather than nested dicts. The *summarize* function needs to be updated to handle this type of input.

This also means that we can no longer feed the test fixture JSON, as is, to the *summarize* function in our *setUp* method. Instead, we need to build input data that mirrors what would be returned by the updated *parse_and_clean* function: Namely, a dictionary containing *Race* instances as values.

First, we'll simplify the test fixtures by removing the nested object structure. Instead, we'll make them a simple array of result objects.

> Note: We could re-use the same JSON fixtures from *elex3* without modification, but this would result in a more convoluted *setUp* method. Wherever possible, use the simplest test data possible.

Then we'll update the *setUp* method to handle our simpflified JSON fixtures, and we'll move into a new *TestSummaryBase* class. *TestSummaryResults* and *TestTieRace* will *sub-class* this new base class instead of *TestCase*, allowing them both to make use of the same *setUp* code.

This is an example of class inheritance. Python classes can inherit methods and attributes from other classes by *subclassing* one or more parent classes. This is a powerful, core concept of object-oriented programming that helps keep code clean and re-usable.

And it's one that we've been using for a while, when we subclassed *unittest.TestCase* in our test classes. We're essentially substituting our own parent class, one that blends the rich functionality of *TestCase* with a custom *setUp* method. This allows the same *setUp* code to be used by methods in multiple subclasses.

```
class TestSummaryBase(TestCase):

    def setUp(self):
        # Recall that sample data only has a single Presidential race
        race = Race('2012-11-06', 'President', '')
        for result in self.SAMPLE_RESULTS:
            race.add_result(result)
        # summarize function expects a dict, keyed by race
        summary = summarize({'President': race})
        self.race = summary['President']


# Update the main test classes to inherit this base class, instead of
# directly from TestCase

class TestSummaryResults(TestSummaryBase):

# ... snipped ...


class TestTieRace(TestSummaryBase):

# ... snipped ...
```

If you ran the *test_summary.py* suite now, you'd see all tests failing.

Now we're ready to swap in our new class-based implementation. This time we'll be deleting quite a bit of code, and tweaking what remains. Below is the new code, followed by a list of major changes:

```
# We removed the defaultdict and use a plain-old dict
summary = {}

for race_key, race in results.items():
    cands = []
    # Call our new assign_winner method
    race.assign_winner()
    # Loop through Candidate instances and extract a dictionary
    # of target values. Basically, we're throwing away county-level
    # results since we don't need those for the summary report
    for cand in race.candidates.values():
        # Remove lower-level county results
        # This is a dirty little trick to botainfor easily obtaining
        # a dictionary of candidate attributes.
        info = cand.__dict__.copy()
        # Remove county results
        info.pop('county_results')
        cands.append(info)

    summary[race_key] = {
        'all_votes': race.total_votes,
        'date': race.date,
        'office': race.office,
        'district': race.district,
        'candidates': cands,
    }

return summary
```

Changes to the *summariz* function include:

- Convert *summary* output to plain dictionary (instead of defaultdict)
- Delete all code for sorting and determining winner. This is replaced by a call to the *assign_winner* method on Race classes.
- Create a list of candidate data as dictionaries without county-level results
- Update code that adds data to the *summary* dictionary to use the race instance and newly created *cands* list.

Of course, we should run our test to make sure the implementation works.

```
nosetests -v elex4/tests/test_summary.py
```

At this point, our refactoring work is complete. We should verify that all tests run without failures:

```
nosetests -v elex4/tests/test_*.py
```

Overall, the *summarize* function has grown much simpler by outsourcing the bulk of work to the *Race* and *Candidate* classes. In fact, it could be argued that the *summarize* function doesn't do enough at this point to justify its existence. Its main role is massaging data into a form that plays nice with the *save_summary_to_csv.py* script.

It might make sense to push the remaining bits of logic into the Race/Candidate model classes and the *save_summary_to_csv.py* script.

You'll also notice that the *summary* tests closely mirror those for the *Race* class in *elex4/tests/test_models.py*. Redundant tests can cause confusion and add maintenance overhead.

It would make sense at this point to delete the *summarize* tests for underlying functionality – tallying votes, assigning winners – and create new tests specific to the summary output. For example, you could write a test that ensures the output structure meets expections.

Questions

- What is a class attribute?
- How does Python construct classes?
- What is the __dict__ special attribute on a class?
- How can the built-in type function be used to construct classes dynamically?

Exercises

- Implement a *Race.summary* property that returns all data for the instance, minus the *Candidate* county results. Swap this implementation into the *summarize* function.
- Delete tests in *elex4/tests/test_summary.py* and add a new test that verifies the structure of the output.

What's Next?

There is a much bigger world of Python language features and object-oriented programming that we didn't cover here. Below are just a few topics that are worth studying as you develop your programming skills.

- *super* (for calling methods on parent classes)
- Multiple inheritance and method resolution order (see the Diamond problem)
- Decorators
- Descriptors
- Meta-programming and __new__ constructor

FAQ

Why refactoring?

First, what the hell is refactoring? The technical definition is explained nicely by Wikipedia. But in a nutshell, it's a deliberate process of changing code so that it's easier to understand and change in the future. Refactoring has a long and rich history and can get quite technical, but we use the term loosely here. We called this git repo/tutorial *refactoring101* because it attempts to show how you can apply some basic principles and techniques to manage larger programs. These skills aren't only useful for changing existing programs. They're also a handy way of designing larger applications from the outset.

Who is this tutorial for?

We assume you're already comfortable with basic programming concepts in Python. You understand loops, conditionals, variables, and basic data types (strings, integers, lists, dictionaries, even sets!). You've also written a few functions in your day. But you're in that nether realm where you get the basics, but aren't quite sure how to write larger programs. Or perhaps like many before you (we've all been there), you've written a monstrosity of a script that is error-prone, brittle, and hard to decipher. You suspect there must be a better way to craft large programs, but you're not quite sure how. If you have that itchy feeling, this tutorial is for you.

How should I use this tutorial?

It's an immersion exercise. This is not a tutorial where we walk through the code, explaining every step. We provide an overview of the code at each stage, and some teasers in the Questions and Exercises sections to nudge you toward new concepts and techniques. But ultimately, it's up to you to read the code, research language features that are new or fuzzy, and experiment by modifying and running the scripts (and tests). That said, you're not alone in the deep end. Hit us up on PythonJournos as you work through the tutorial. We're friendly. We promise :)

Why did we write this?

Because like you, we've experienced the thrill of mastering the basics – writing those first few scripts to get things done – and the inevitable frustration of not knowing what comes next. How do I write a bigger program that does more than one thing? How do I know if some part of the program failed? How can I use this bit of code in another script, without having to update the code in both places? We've wrung our fists in the air over the same questions. Hopefully this tutorial helps nudge you toward some answers.

Who wrote this tutorial?

Jeremy Bowers and Serdar Tumgoren, nerds from the journalism community. Don't be shy. Hit us up with questions, pull requests, angry criticisms, etc.

Am I alone?

That's pretty deep. But we're inclined to say no.

Resources

Some resources geared for the intermediate Python programmer.

- Hitchhiker's Guide to Python, especially the section on Structuring your Project and Testing Your Code
- Dive Into Python 3 is a solid book for the intermediate programmer looking to deepen his or her skills. Especially check out the sections on Unit Testing and Refactoring. There's also the Python 2.x version.
- Refactoring is a classic. Yes, it's printed on dead trees and code samples are in Java. But the collective wisdom and many of the practical techniques remain invaluable. It's an eye-opener.