Reread this reference book and take very detailed notes. Reading schedule: 8-9 chapters per day (done by 5/1/16)

Coding Style

- Code is read more than it is written, so go an extra mile to ensure:
- No variable names abbreviations
- Function args must be written out
- Document classes and methods
- Refactor repeated code into reusable functions
- PyCharm has a built-in PEP8 suggestion system, so follow that
- import in this order:
    - standard library
    - core Django
    - related 3rd-party packages
    - local application or library specific imports
    - Ex:

```
# standard imports
from math import sqrt
from os.path import abspath

# core
from django.db import models
from django.utils.translation import ugettext_lazy as _

# 3rd-party
from django_extentions.db.models import TimeStampedModel

# apps specific
from splits.models import BananaSplitAppName
```

- Avoid hard-coded imports
    - Bad

```
from cones.models import WaffleCone
from cones.views import FoodMixin
```

    - Good

```
from .models import WaffleCone
from core.views import FoodMixin
```

• Code	Import Type	Usage
`from core.views import FoodMixin`	absolute import	import from outside current app
`from cones.views import FoodMixin`	explicit relative	import from another module in current app
`from cones.models import Waffle`	implicit relative	import from another module, not a good idea

- Don't ever use `import *`
- Use underscores names rather than dashes for URL pattern, template block names
- Make sure project could even be worked on using NotePad or Nano

Optimal Django Environment Setup

- Use the same database engine for local and production environment. Use PostgreSQL.
- Avoid using fixtures to migrate large data sets from one db to another
- Use `pip` and `virtualenv` to install packages and isolate environment
- Consider using `virtualenvwrapper`

Optimal Project Layout

- `<repository_root>`

```
        README.md
        .gitignore
        requirements.txt
        docs/
        <django_project_root>
            manage.py
            app1/
            app2/
            static/
            templates/
                app1/
                app2/
            <configuration_root>
                __init__.py
                settings/
                urls.py
                wsgi.py
```
- Consideration: place all `virtualenv` in one folder and projects in another

Django App Design Fundamentals
- Definitions:
    - Django project: a web application powered by the Django web framework
    - Django app: small library designed to represent a single aspect of a project. A Django project is made of many Django apps. Some of the apps are internal to the project and will never be reused; some are third-party Django packages.
    - Third-party Django packages: pluggable, reusable Django apps that have been packaged with the Python Packaging tools
- Golden Rule: Write programs that do one thing and do it well.
    - Each app should be tightly focused on its task. If an app can't be explained in a single sentence of moderate length or you need to say "and" more than once, it probably means the app is too big and should be broken down.
    - Ex: a website for Two Scoops Ice Cream Shop:
        - Project: twoscoops_project
        - *flavors* app to track all flavors and list them
        - *blog* app for the official pieces
        - *events* app to display listings of shop's events
- Use single word if possible for app names. Use valid, PEP8-compliant names: short, all-lowercase, no numbers, dashes, periods, spaces, or special character. Use _ to separate words, but this is highly discouraged
- Keep apps small. It's better to have a bunch of small apps than having a few giant ones

Settings and Requirement Files
- All settings files need to be version-controlled.
- Don't repeat yourself. Inherit from a base settings file rather than copying and pasting.
- Keep secret keys safe. Keep them out of version control. Same thing goes to API secret keys, database credentials
- Use different settings files that are inherited from the same base
    - `python manage.py runserver --setings=twoscoops.settings.local`
    - or use PyCharm's configuration widget
    - or export `DJANGO_SETTINGS_MODULE` env variable
- Use environment variable to store secret keys
- Follow the same DRY concept for requirements.txt files:
```
requirements/
    base.txt
    dev.txt

#base.txt
Django==1.5.1
psycopg2==2.4.5

#dev.txt
-r base.txt
django-debug-toolbar==0.9.4
```
- Don't hardcode paths to anything

Database/Model Best Practices

- Spend time thinking about the models design. This will save a lot of time down the road and prevent a lot of unnecessary work-around and corrupting existing data
- Useful packages that are pretty much in every project:
  - South: data migrations
  - `django-model-utils`: handle common patterns like TimeStampedModel
  - `django-extensions`: autoloaders model classes for all installed apps
- Break up apps into many models. No more than 5 models per app.
- Don't use raw SQL until it's absolutely necessary. Use Object-Relational Model (ORM).
- Only index models when:
  - index is used in 10-25% of all queries
  - there is real data to index and analyze the cost-benefit
  - can run tests to determine if indexing generates benefits
- `pg_stat_activity` gives good stats on indexes for PostgreSQL
- Model inheritance:
  - Abstract base classes: it's different from Python abstract class. Tables are only created for derived models
    - Can reuse common fields; no extra overhead of tables and joins that are incurred from multi-table inheritance, but cannot use parent class in isolation
  - Multi-table inheritance: tables created for both parent and child, an implied one to one relationship between parent and child:
    - Can give each model its own table, so can query either parent or child model, but adds substantial overhead because of table joins. Use with caution.
  - Proxy models: table is only created for the original model
    - Allows to alias a model with different Python behavior, but can't change the model's fields
- When to use which model inheritance:
  - If the overlap between models is minimal (only a couple that share one or two obvious fields), then no need for model inheritance.
  - If there is enough overlap that maintenance causes confusion, use abstract base class
  - Proxy models are occasionally useful
  - Avoid multi-table inheritance at all cost. Use explicit OneToOneFields and FK between models to gain control when joins are traversed
- Ex:

```
class TimeStampedModel(models.Model):
    created = models.DateTimeField(auto_now_add=True)
    modified = models.DateTimeField(auto_now=True)
class Meta:
    abstract = True
```

- South: the defacto migration tool. Learn to use if not familiar.
  - As soon as a new app or model is created, create the initial South migrations
  - Write reverse migrations and test them. Not being to roll back hurts bug tracking big time
  - Flatten migrations before production. Only include enough. No more, no less.
  - Do not ever remove migration files in production
  - Test data should be in similar size of production data
- Model Design:
  - Start Normalized. No model should contain data already stored in another model.
  - Cache before Denormalizing
  - Denormalize only if absolutely needed. It's a tricky process that risks adding complexity to the project and dramatically raises the risk of losing data. Cache before denormalization.
- When to use Null and Blank
  - `CharField, TextField, SlugField, EmailField,CommaSeparatedIntegerField`
    - `null=True` Don't. Django's convention is to store empty values as empty string and to always retrieve NULL or empty values as the empty string
    - `blank=True` Ok if corresponding form widget can accept empty values
  - `BooleanField`
    - `null=True` Don't. Use `NullBooleanField`.
    - `blank=True` Don't.
  - `IntegerField, FloatField, DecimalField`
    - `null=True` Ok if value can be set to `NULL` in db.
    - `blank=True` Ok if corresponding form widget can accept empty values. Set `null=True` too.

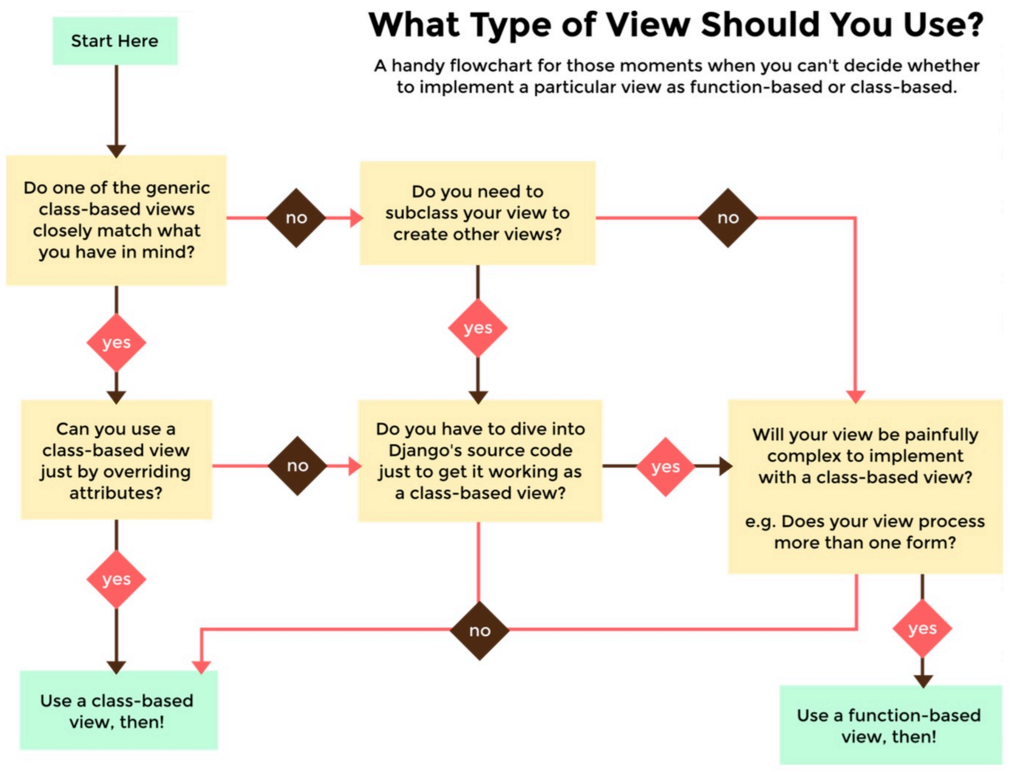
- `DateTimeField, DateField, TimeField`
  - `null=True` Ok if value can be set to `NULL` in db.
  - `blank=True` Ok if corresponding form widget can accept empty values. Set `null=True` too.
- `ForeignKey, ManyToManyField, OneToOneField`
  - `null=True` Ok if value can be set to `NULL` in db.
  - `blank=True` Ok if the select box widget can accept empty values.
- `IPAddressField, GenericIPAddressField`
  - `null=True` Ok if value can be set to `NULL` in db.
  - `blank=True` Not recommended.

- Model Managers:
  - Can define a custom Manager beside the default objects Manager:

```
class PublishedManager(models.Manager):
    use_for_related_fields = True

def published(self, **kwargs):
    return self.filter(pub_date__lte=timezone.now(), **kwargs)

class FlavorReview(models.Model):
    review = models.CharField(max_length=255)
    pub_date = models.DateTimeField()

    objects = PublishedManager()

FlavorReview.objects.count() # 35
FlavorReview.objects.published().count() #31
```

  - Be very careful when replacing the default model manager with the custom one:
    - When using model inheritance, children of abstract base classes inherit their parent's model manager, but children of concrete base classes do not.
    - The first manager applied to a model class is the default one. This breaks significantly with the normal Python pattern, causing unpredictable results from QuerySets.
  - `objects = models.Manager()` should be placed above any custom model manager that has a new name

Function- and Class-Based Views


What Type of View Should You Use?
A handy flowchart for those moments when you can't decide whether to implement a particular view as function-based or class-based.
Start Here
Do one of the generic class-based views closely match what you have in mind?
no
yes
Do you need to subclass your view to create other views?
no
yes
Can you use a class-based view just by overriding attributes?
no
yes
Do you have to dive into Django's source code just to get it working as a class-based view?
yes
no
Will your view be painfully complex to implement with a class-based view?
e.g. Does your view process more than one form?
yes
no
Use a class-based view, then!
Use a function-based view, then!

- Keep View logic out of URLConfs. The coupling of views with urls is loose to allow infinite flexibility.
- The views modules should contain View logic. The URL modules should contain URL logic.
- Don't do this, even the official Django doc does:

```
    urlpatterns = patterns(url(r'\d+/', DetailView.as_view(model=Tasting,
template_name="detail.html"), name="detail"))
```

- Keep business logic out of views. It's not always possible to do this at the beginning, so once it has come to duplicating business logic instead of Django boilerplate between views, it's time to move the code out of the view.

Best Practices for Class-Based Views

- `django-braces` + CBVs is a great combination. The package contains many useful, clearly coded mixins that make CBVs much easier and faster to implement.
- Since CBVs are so powerful, they are complex. The inheritance chain can have up to 8 superclasses on import, making it hard to know what to override.
- When writing CBVs:
    - less view code is better
    - never repeat code in views
    - views should handle presentation logic. Keep business logic in models or forms
    - keep views simple and mixins simpler
- Mixins are to CBVs is like toppings are to ice cream. Mixin is a class that provides functionality to be inherited, but isn't meant for instantiation on its own. Use these rules when using mixins to composite view classes; they follow Python's method resolution order:
    - the base view classes provided by Django always to go the right
    - mixins go to the left of the base view
    - mixins should inherit from Python's built-in object type
    - Ex:

```
class FreshFruitMixin(object):
    def get_context_data(self, **kwargs):
        context = super(FreshFruitMixin, self).get_context_data(**kwargs)
        context["has_fresh_fruit"] = True
        return context

class FruityFlavorView(FreshFruitMixin, TemplateView):
    template_name = "fruit.html"
```

- View usage:

`View`	Base view or handy view that can be used for anything
`RedirectView`	Redirect user to another URL
`TemplateView`	Display HTML template
`ListView`	List object
`DetailView`	Display an object
`FormView`	Submit a form
`CreateView`	Create an object
`UpdateView`	Update an object
`DeleteView`	Delete an object
Generic date views	Display objects that occur over a range of time

- The 3 schools of Django CBV usage:
    - use all the views
        - only `django.views.generic.View` is enough
    - avoid them unless you're actually subclassing views (aka use function views)
        - Constrain access to authenticated users by using `LoginRequiredMixin`
        - Use `form_valid()` in `CreateView` to perform custom action on a view with a valid form
    - Views + ModelForm: when you create a model, you need to add new records and update existing records that correspond to the model
- Prefix template's name with _ if it is used in many parts of the project

Common Patterns for Forms

- `django-crispy-forms` is a very useful package used for advanced form layout controls
- Simple ModelForm with default Validators:

```
from .models import Flavor
class FlavorCreateView(LoginRequiredMixin, CreateView):
    model = Flavor

class FlavorUpdateView(LoginRequiredMixin, UpdateView):
    model = Flavor
```

  - Both views assign Flavor as their model and auto-generates a ModelForm based on the fields of Flavor model. The ModelForm will then rely on the default field validation rules of the Flavor model to validate input. In practice, the defaults are never enough.

- Custom Form Field Validators with ModelForms
  - Use case: assume we have a project with 2 different models: Flavor and Milkshake, both of which have a title field. We put a constraint that every title field must start with "Tasty".
  - Create a validators.py module for custom validation

```
def validate_tasty(value):
    if not value.startswith(u"Tasty"):
        raise ValidationError(u"Must start with Tasty")
```

  - In a central models.py module, create an abstract model:

```
from .validators import validate_tasty
class TastyTitleAbstractModel(models.Model):
    title = models.CharField(validators=[validate_tasty])

class Meta:
    abstract = True
```

  - Then, in the respective app, make the Flavor model inherit from this abstract model:

```
class Flavor(TastyTitleAbstractModel):
    slug = models.SlugField()
```

  - Any model inheriting from `TastyTitleAbstractModel` will now have the title validation enforced.

- What if we only wanted `validate_tasty()` in forms?

```
class FlavorForm(forms.ModelForm):
    def __init__(self, *args, **kwargs):
        super(FlavorForm, self).__init__(*args, **kwargs)
        super.fields["title"].validators.append(validate_tasty)

class Meta:
    model = Flavor
```

  - Notice we don't need to change the logic of `validate_tasty()` at all. Use this in a view:

```
class FlavorActionMixin(object):
    @property
    def action(self):
        msg = "{0} is missing action.".format(self.__class__)
        raise NotImplementedError(msg)

    def form_valid(self, form):
        return super(FlavorActionMixin, self).form_valid(form)

class FlavorCreateView(FlavorActionMixin, CreateView):
    model = Flavor
    action = "created"
    form_class = FlavorForm
```

- Overriding clean stage of validation
  - Use case: multi-field validation, validation involving existing data from the database that has already been validated
  - Ex: prevent users from ordering out of stock flavors

```
class IceCreamOrderForm(forms.Form):
    slug = forms.ChoiceField("Flavor")
    toppings = forms.CharField()
```

```
        def __init__(self, *args, **kwargs):
            super(IceCreamOrderForm, self).__init__(*args, **kwargs)
            self.fields["slug"].choices = [(x.slug, x.title) for x in Flavor.objects.all()]

        def clean_slug(self):
            slug = self.cleaned_data["slug"]
            if Flavor.objects.get(slug=slug).scoops_remaining <= 0:
                raise forms.ValidationError("Out of order.")
            return slug
```

- Hacking form fields (2 CBVs, 2 Forms, 1 Model)

```
    class StoreCreateForm(forms.ModelForm):
        class Meta:
            model = IceCreamStore
            fields  = ("title", "address")

    class StoreUpdateForm(StoreCreateForm):
        class Meta(StoreCreateForm.Meta):
            fields = ("title", "address", "phone", "description")

        def __init__(self, *args, **kwargs):
            super(StoreUpdateForm, self).__init__(*args, **kwargs)
            self.fields["phone"].required = True
            self.fields["description"].required = True

        class CreateView(CreateView):
            model = IceCreamStore
            form_class = StoreCreateForm

        class UpdateView(UpdateView):
            model = IceCreamStore
            form_class = StoreUpdateForm
```

- Reusable Search Mixin View
    - Use case: a single CBV can provide simple search functionality on multiple models

```
    class TitleSearchMixin(object):
        def get_queryset(self):
            queryset = super(TitleSearchMixin, self).get_queryset()

            q = self.request.GET.get("q")
            if q:
                return queryset.filter(title__icontains=q)
            return queryset

    class FlavorListView(TitleSearchMixin, ListView):
        model = Flavor

    class IceCreamStoreListView(TitleSearchMixin, ListView):
        model = Store

    <form action="" method="GET">
        <input type="text" name="q"/>
        <button type="submit">search</button>
    </form>
```

More Things to Know About Forms

- All form that alters data must submit via POST method
- Do NOT disable CSRF protection
- `form.is_valid()` under the hood:
    - If form has bound data, `form.is_valid()` calls `form.full_clean()`
    - `form.full_clean()` iterates through form fields and validates
    - Data coming into the field is coerced into Python via `to_python()` or raises `ValidationError`
    - Data is validated against field-specific rules, including custom validators
    - If there are any custom `clean_<field>()` method, they're invoked

    - `form.full_clean()` calls `form.clean()`
- If a ModelForm instance, `from._post_clean()` sets ModelForm data to Model instance, regardless of `form.is_valid()`'s return value. Then it calls the model's `clean()` method. Saving a model instance through the ORM doesn't call the `clean` method.

Building REST APIs in Django

- Packages:
    - django-rest-framework: builds off of Django CBVs, adding a browsable API feature
    - django-tastypie: feature-rich, mature, powerful, stable tool for creating APIs from Django models
    - django-braces: can be used in direct conjunction with Django CBVs to create super quick and simple one-off REST API views
- Fundamentals:
    - Hypertext Transfer Protocol (HTTP) is a protocol for distributing content that provides a set of methods to declare actions

Create a new resource	POST
Read an existing resource	GET
Request the header of an existing resource	HEAD
Update an existing resource	PUT
Update part of an existing resource	PATCH
Delete an existing resource	DELETE
Return the supported HTTP methods for given URL	OPTIONS
Echo back request	TRACE
Tunneling over TCP/IP	CONNECT

    - If implementing a read-only API, then only implement GET
    - If implementing a read-write API, then at least also use POST, but consider PUT and DELETE
    - By definition, GET, PUT, and DELETE are idempotent. POST and PATCH are not idempotent: same request repeatedly should produce the same result. Meaning making multiple identical requests has the same effect on making a single request
    - PATCH is often not implemented, but should if API supports PUT
    - Common server response code:

200 OK	Success	GET - return the resource PUT - provide status message or return resource
201 Created	Success	POST - provide status message or return newly created resource
204 No Content	Success	DELETE
304 Unchanged	Redirect	ANY - indicates no changes in the last request
400 Bad Request	Failure	PUT, POST - return error messages, including from validation errors
401 Unauthorized	Failure	ALL - authentication required but user did not provide credentials
403 Forbidden	Failure	ALL - user attempted to access restricted content
404 Not Found	Failure	ALL - resource is not found
405 Method Not Allowed	Failure	ALL - an invalid HTTP method was attempted

- Implementing a simple JSON API. Ex:

```
class Flavor(models.Model):
    title = models.CharField(max_length=255)
    scoops_remaining = models.IntegerField(default=0)

from rest_framework.generics import ListCreateAPIView, RetrieveUpdateDestroyAPIView

class FlavorCreateReadView(ListCreateAPIView):
    model = Flavor

class FlavorReadUpdateDeleteView(RetrieveUpdateDestroyAPIView):
    model = Flavor
```

    - Add in user authentication and wire this into urls.py

Best Practices for Templates

- Django's limitations on what can be done on the template may seem inconvenient at first, but it forces business logic to live in Python modules

- For sites with consistent layout overall from app to app:
```
templates/
    base.html
    dashboard.html
    profiles/
        detail.html
        form.html
```
- For sites whose sections require a distinctive layout
```
templates/
    base.html
    dashboard.html
    profiles/
        base_profiles.html
        profile_detail.html
        profile_form.html
```
- Flat template hierarchies is always better than nested
- Limit processing in templates. When iterating over a query in a template, study its size and amount of processing per iteration.
    - Don't do aggregation operations like "add template filter" on the template level. Leave all the processing on the model level and then use the template to display the result.
    - Don't filter with conditionals in templates. Again, keep the filtering in the model level.
    - Don't use complex implied queries in templates. Use ORM's `select_related()` method to avoid hitting the database again for extra queries
    ```
    {% for user in user_list %}
    {{ user.name }}
    {{ user.flavor.title }}
    ```
    - If `user_list` was the variable for `User.object.all()`, every `user.flavor.title` is an implicit query to the database. Use `User.object.all().select_related()` to avoid this.
- Beware of hidden CPU load in templates. For example, when dealing with saving image data to file systems, it's always a good idea to leave this logic in the model, where asynchronous message queues like Celery could take over.
- REST API consumption should occur in JavaScript code so after the content is served, the client's browser handles the work, or the View code, where slow processes can be handled by message queues, additional threads, multiprocesses, etc.
- HTML will be rendered with correct formatting, so don't bother making it look pretty. However, in your template, use indentation and one operation per line for readability and maintainability.
- Use template inheritance. DRY.
- Avoid coupling styles too tightly to Python code. Aim to control all stylings entirely via CSS and JS. Don't hardcode stylings in Python code.
- Use underscores for template names, block names, and other names in templates. Make the names clear and intuitive.
- Include the block name in `endblock` tags
- Templates called by other templates should be prefixed with _. This applies to `{% includes %}` templates or custom template tags.
- Use named context objects in templates
- Do not hardcode URL paths. Use `{% url 'url_name' %}` instead.

Template Tags and Filters

- Default template tags and filters all have clear, obvious names. They all do one thing and don't alter any sort of persistent data. If you want to write your own template tag, follow those guidelines.
- Filters are just functions that take in one or two args. This simplicity prevents abuse in template. They are good for modifying the presentation of data.
- Template tags are harder to debug, make code reuse harder, and can have a significant performance cost, especially when loading other templates. So, only write new template tags when they are only responsible for rendering of HTML.

Tradeoffs of Replacing Core Components

- Don't do it. It's certainly possible, but only worth if:
  - okay with sacrificing the ability to use 3rd party Django packages
  - okay without Django admin
  - already built project with core Django components but are running into major blockers
  - already analyzed the code to fix root causes of problems
  - explored all alternatives
  - project is a real, live production site with tons of users
  - okay with the pain or impossibility of upgrading to new Django
- A few popular reasons why people want to replace core components:
  - Replacing db/ORM with a NoSQL db and corresponding ORM replacement
  - Replacing Django's template engine with Jinja2, Mako, or something else

Django Admin

- This is not the interface for the end users. It's the place where site administrators add/edit/del data and perform site management tasks. Don't stretch it into something the end user could use.
- It's usually not worth it to heavily customize the Django admin.
- Always implement `__unicode__()` method for each model.
- Use `list_display` to show data for additional fields. Use callables such as methods and functions to add view functionality to `django.contrib.admin.ModelAdmin` class.

```
class IceCreamBarAdmin(admin.ModelAdmin):
    list_display = ("name", "shell", "filling", )
    readonly_fields = ("show_url",)

    def show_url(self, instance):
        return """<a href="{0}">{0}</a>""".format(instance.pk, instance.name)

show_url.short_description = "Ice Cream Bar URL"
show_url.allow_tags = True

admin.site.register(IceCreamBar, IceCreamBarAdmin)
```

Dealing with User Model

- `from django.contrib.auth import get_user_model`
- Attach FK, OneToOneField, M2M to User:

```
class IceCreamStore(models.Model):
    owner = models.OneToOneField(settings.AUTH_USER_MODEL)
    title = models.CharField(max_length=255)
```

- Using custom User fields:
  - Linking from a related model using `settings.AUTH_USER_MODEL` like above
  - Subclass `AbstractUser`
  - Subclass `AbstractBaseUser`
- This is the bare-bones option with only 2 fields: `password`, `last_login`, and `is_active`
  - Use if you're not happy with the default fields of User model or prefer to subclass from an extremely bare-bones clean slate but still want to take advantage of the `AbstractBaseUser` approach to strong passwords
- Third-party packages should not be defining the User model

Third-Party Packages

- PyPI (http://pypi.python.org/pypi) is a repo of software for Python. This is where `pip` downloads packages from
- Django Packages (http://djangopackages.com) is a directory of reusable apps, sites, tools, and more. It's used to compare and evaluate package features
- Don't reinvent the wheel. Use 3rd party packages whenever possible. Check their licenses first, though.
- Always freeze the package dependencies with version numbers

Testing

- Replace the test.py file created by `python manage.py startproject` with a test/ directory. Include an init.py file to make it a module. Tests that apply to forms go into test_forms.py, to models go into test_models.py
- Each test should only test one thing

- Test should be written as simply as possible. Don't write tests that have to be tested
- Don't rely on data fixtures

Documentation

- Install sphinx system wide since documentation is important for every project
- reStructuredText(RST) is the most common markup language used for documenting Python projects. Learn and follow the standard

```
Section Header
==============

**emphasis (bold/strong)**
*italics*

Simple link: http://django.2scoops.org
Fancier Link: `Two Scoops of Django` _

.. _ `Two Scoops of Django`: https://django.2scoops.org

Subsection Header
-----------------------

#) An enumerated list item
#) Second item
* First bullet
* Second bullet
* indented bullet

Literal code block::
def like():
    print "I like"

Python colored code block (requires pygments):
code-block:: python
for i in range(10):
    print "pip install pygments"

Javascript colored code block:
code-block:: javascript
console.log("Alerts are bad.");
```

- Doc should contain:

README.rst	Must have. Briefly describe what this project does and installation instruction.
docs/	All docs go in here.
docs/deployment.rst	Point-by-point set of instructions on how to install/update the project into production
docs/installation.rst	Point-by-point set of instructions on how to onboard yourself or a newbie
docs/architecture.rst	A guide for understanding what things evolved from as the project grows in scope.

- If developer-facing docs can't be placed in the project's version control, at least consider wikis, online documents stores, or even Pages or Word document. Some doc is better than no doc.

Finding and Reducing Bottlenecks

- Premature optimization is bad. If your site is small- or medium-sized and the performance is fine, then it's okay to skip this step.
- Speed up query-heavy pages:
    - Use `django-debug-toolbar` to find excessive queries: duplicated queries in a page, slow queries, ORM calls that resolve to many more queries than expected
    - Determine pages with undesirable number of queries and figure out the way to reduce that number.
    - Try using `select_related()` to combine queries.
    - If the same query is generated more than once per template, move the query into View, add it to the context as variable, and point the template ORM calls at this new context variable.
    - Implement caching using key/value store such as Memcached. Then write tests to assert the number of

    queries run in a view.
- Speed up common queries that take a long time to execute:
    - Make sure indexes are helping with speeding up common slow queries. Look at the raw SQL generated and index on the fields that you filter/sort most frequently.
    - Understand what indexes are actually doing in production. Dev environment will never perfectly replicate what happens in production.
    - Turn on db's slow query logging feature to see if any slow queries occur frequently
    - Use `django-debug-toolbar` in dev to defensively identify potential slow queries before they hit production
    - Always rewrite logic to return smaller result sets when possible.
    - Re-model data in a way that allows indexes to work effectively
    - Drop to raw SQL in places where it would be more efficient than generated query
- For PostgreSQL, use `EXPLAIN ANALYZE` to get an extremely detailed query plan and analysis of any raw SQL query
- Optimize the database:
    - Logs don't belong in the db
    - Move ephemeral data to Memcached, Redis, Riak and other non-relational stores
    - Binary data shouldn't be stored in db either
    - Cache queries with Memcached or Redis
- Identify which views/templates contain the most queries, which URLs are being requested the most, and when should a cache for a page be invalidated.
- Popular Django packages for caching: `django-cache-machine`, `johnny-cache`
- Compress and minify CSS and JS. `django-pipeline` is a good package to consider
- Use Upstream caches like Varnish or CDNs to serve static media like images, video, CSS, JS

Best Practices for Security
- Harden the servers by doing things like changing SSH port and disabling/removing unnecessary services
- Django's security features include these; most work out of the box, but it is the developer's responsibility to setup proper configurations and follow best practices:
    - cross-site scripting (XSS) protection
    - cross-site request forgery (CSRF) protection
    - SQL injection protection, clickjacking protection
    - support for SSL/HTTPS, including secure cookies
    - validation of files uploaded by users
    - secure password storage using PBKDF2 algorithm with a SHA256 hash by default
    - automatic HTML escaping
- Turn off `DEBUG` mode in production, since attackers can find out more than they need to know about production setup
- Keep `SECRET_KEY` out of settings module and out of version control. Follow tips in the Optimal Environment section
- Use HTTPS everywhere, even for static resources. Not having HTTPS means malicious network users can sniff authentication credentials and data between your site and end users
- All requests to HTTP should be redirected to HTTPS. This should be done through configuration of the web server. If that's not an option, Django middleware is fine as well
- Purchase an SSL certificate from a reputable source rather than creating a self-signed one
- Use secure cookies. Turn them on in settings.py
    - `SESSION_COOKIE_SECURE = True`
    - `CSRF_COOKIE_SECURE = True`
- Use HTTP Strict Transport Security (HSTS)
    - HSTS is usually configured at the web server level. If enabled, the site's web pages include a HTTP header that tells HSTS-compliant browsers to only connect to the site via secure connections. If connection is not possible, access will be disallowed.
    - use HSTS's `includeSubdomains` mode if possible. This prevents attacks using non-secured subdomains to write cookies for the parent domain
    - Set `max-age` to a large value like 31536000 (12 months) or 63072000 (24 months) if possible, but beware you can't change this
    - HSTS should be enabled in addition to redirecting all HTTP to HTTPS. This is not an alternative
- Use Django 1.5's allowed host validation:
    - In the settings module, set `ALLOWED_HOSTS` to a list of allowed host/domain name. Avoid setting wildcard

values.

- Always use CSRF protection with HTTP forms that alter data. The only use case for turning off CSRF is for creating APIs. `django-tastypie` and `django-rest-framework` take care of this for you.
    - Use `CsrfViewMiddleware` as a blanket protection across your site rather than manually decorating views with `csrf_protect`
    - Use Django's CSRF protection even when posting data via AJAX. Do not make AJAX views CSRF-exempt. You'll need to set an HTTP header called `X-CSRFToken` when posting via AJAX.
- Defend against XSS
    - Django by default escapes a lot of specific characters meaning most XSS attacks involving enter malicious JS as user input will fail. However, developers has the ability to mark content strings as safe, meaning Django won't escape user inputs. So beware of what you mark safe.
    - Defend against Python injection attacks
    - Beware of `eval()`, `exec()`, `execfile()` built-ins if your project allows arbitrary strings or files to be passed into any of these functions
    - Do not use pickle module to serialize or deserialize anything that might contain user input
    - When using PyYAML, only use `safe_load()`. For reference, `yaml.load()` will let you create Python objects, which is very, very bad.
    - Validate all user input with Django forms
- Handle user-uploaded files carefully:
    - Use python-magic library to ensure uploaded file's headers are of certain file types that are being accepted.
    - Take extra care with web server's configuration to prevent users from uploading malicious executable like a CGI or PHP script onto web servers and execute them via URL
- Don't use `ModelForms.Meta.exclude` to list out the fields to render. Always explicitly list out
    - `fields = ("field_1", "field_2", "field_3",)`
    - Often, a model could be used for many forms. If you add a new field into a model and want to exclude it, you'll have to remember to exclude it from all the forms that use this model. Explicitly listing out the fields in `fields` prevents this hassle and security vulnerability.
- Beware of SQL injection when using raw SQL. Django ORM generates properly-escaped SQL, but when you drop down to raw SQL, you're on your own
- Never store credit card data. Use Stripe, Balanced Payments, PayPal that handle this for you
- Secure Django Admin
    - Change the default admin URL to something hard to guess
    - `django-admin-honeypot` puts a fake Django admin at login screen at admin/ and logs info about anyone trying to log in
    - Only allow admin access over HTTPS
    - Limit admin access based on IP address
    - `allow_tags` attribute can only be used on system generated data like PK, dates, and calculated values, never for character and text fields. `allow_tags` attribute (False by default) allows HTML tags to be displayed in the admin when set to True
    - Secure admin docs. Only serve via HTTPS and allowed on certain IP address
- Monitor your sites by checking server's access and error logs regularly
- Keep all dependencies up-to-date to get all the security fixes
- Prevent clickjacking (when malicious site tricks users to click on a concealed element of another site that they have loaded in a hidden frame or iframe).
- http://ponycheckup.com is a good tool to check up on security. Obviously we don't want to do this with internal apps
- Put up vulnerability reporting page for those who want to fix your code
- Always be curious - keep up with general security practices.

Logging

- Logging can be used not only to debug application errors, but also track interesting performance metrics
- Different log levels: `DEBUG, INFO, WARNING, ERROR, CRITICAL`
    - `CRITICAL`: something catastrophic has happened that requires urgent attention; something that core functionality depends on but is no longer available
    - `ERROR`: whenever code raises an exception that is not caught. Everyone listed in ADMINS get a detailed email about the problem
    - `WARNING`: events that are unusual and potentially bad, but not as bad as `ERROR`-level events
    - `INFO`: any details that may be important for analysis like startup/shutdown of important components not

    logged elsewhere, state changes that occur in response to an important event, or changes to permissions
  - `DEBUG`: in place for print statement for production and because print statements are not recorded. Python 3 will break old-styled print statement
- `Logger.exception()` automatically includes the traceback and logs at `ERROR` level
- Use one logger per module that uses logging
- Log locally to files incase network goes down or emails can't be sent to admins for some reasons
- While debugging, use Python logger at `DEBUG` level. After testing, run them at `INFO` and `WARNING` levels
- Color-code your log outputs using `logutils` package. It also comes with the ability to log to queues, classes that allow you to write unit tests for log messages, and an enhanced HTTPHandler that supports secure connection over HTTPS

Signals: Use Cases and Avoidance Techniques

- Use signals as a last resort. Signals are synchronous and blocking
- Do not use signals when:
  - the signal relates to one particular model and can be moved into one of that model's methods
  - the signal can be replaced with a custom model manager method
  - the signal related to a particular view and can be moved into that view
- Might be okay to use signals when:
  - signal receiver needs to make changes to more than on model
  - want to dispatch the same signal from multiple apps and have them handled the same way by a common receiver
  - want to invalidate a cache after a model save
  - have an unusual scenario that needs a callback and there's no other way to handle it besides using signal
- Signal avoidance techniques:
  - Use custom model `Manager` methods instead of signals.
  - Ex: need to do some custom logic when creating a new instance of a model
    - Instead of using signal, create a custom `Manager` for the model and include the logics here
  - Validate model elsewhere: don't use `pre_save` signal to trigger input cleanup for a specific model. Write custom validator for fields instead. If validating through a ModelForm, override `clean()` method instead
  - Override Model's Save and Delete method to avoid `pre_save` and `post_save` signals to trigger additional logic that only applied to one model. Same idea goes to `delete()`

Random Utilities

- Place all the random utilities into a utils.py module
- Don't rely on django.utils package as they might change between versions

Deployment

- Using your own web servers:
  - deploy with WSGI. The most commonly-use WSGI deployment setups are:
    - Gunicorn behind a Nginx proxy
    - Apache with mod_wsgi
- Gunicorn (sometimes with Nginx)
  - written in pure Python. Supposedly this option has slight better memory usage, but your mileage may vary. Has built-in Django integration.
  - doc is brief for nginx (but growing). Not as time-tested so may run into confusing configuration edge cases and occasional bugs
- Apache with mod_wsgi
  - been around for a long time and is tried and tested. Very stable and lots of great docs
  - configuration can be overly complex and painful for some
- Disadvantage of setting up your own web servers is the overhead of extra sysadmin work
- Using PaaS: Obviously this doesn't really apply to us. Contact Chris Crowe.
  - Heroku
  - Gondor.io
  - DotCloud