My notes

Release 0.2

Tomasz Zieliński

September 06, 2012

CONTENTS

This is a compilation of things that I stumbled upon or learned during my work. I decided to make it public to give something back to the Open Source community which equipped me with most of the tools I use (Linux, Python, Django, etc.).

Contents:

CHAPTER

ONE

DJANGO 1.3, 1.4 TIPS&TRICKS

1.1 settings.py

- Either have a global, versioned *settings.py* file which imports a local (non-versioned) configuration:

  ```
  import settings_local
  ```

  which has a versioned template *settings_local.py.template*, or use the reverse approach - have a common settings file, e.g. *common_settings.py* and then a non-versioned settings.py which imports the common stuff. The latter seems to be the preferred way.

- Figure out project root using either:

  ```
  PROJECT_ROOT = os.path.dirname(os.path.realpath(__file__))
  ```

  or:

  ```
  PROJECT_ROOT = os.path.realpath(os.path.dirname(__file__))
  ```

  both forms seems to be actively used and they are pretty much equivalent.

- To get full file paths, use:

  ```
  os.path.join(PROJECT_ROOT, 'dir1', 'myfile.txt')
  ```

- You probably always want to have the detailed information about errors in templates. This is independent of the DEBUG setting:

  ```
  TEMPLATE_DEBUG = True
  ```

- You may want to use HttpOnly cookies:

  ```
  SESSION_COOKIE_PATH = '/; HttpOnly'
  SESSION_COOKIE_HTTPONLY = True
  ```

  Changed in version Django: 1.4 *SESSION_COOKIE_HTTPONLY* is True by default in Django 1.4+

- For multilingual sites use:

  ```
  USE_I18N = True
  ```

  You might also want:

  ```
  USE_L10N = True
  ```

  Language definitions:

```
gettext = lambda s: s
LANGUAGES = (
    ('sv', gettext('Swedish')),
    ('en', gettext('English')),
)
```

- If you use a global (per-project) template folder you need:

```
TEMPLATE_DIRS = (os.path.join(PROJECT_ROOT, 'templates'),)
```

1.2 (De facto) standard add-ons

- South migrations - you might want to use the following settings:

```
SKIP_SOUTH_TESTS  = True,
SOUTH_TESTS_MIGRATE  = False
```

(SKIP_SOUTH_TESTS, SOUTH_TESTS_MIGRATE)

- Django Debug Toolbar - make sure to configure it according to the docs
- Django Sentry - the preferred way to catch exceptions and log messages. It has been split into Sentry and Raven so now both are needed. Note that because Sentry/Raven are meant to replace Django's default mechanism and also to integrate deeply into the framework, some attention is needed during configuration. Also note that there were (still are?) unsolved problems like this one. But still, Sentry/Raven is probably one of the best such tools out there.

1.3 MySQL 5.x

- Create the database using the following command:

```
CREATE DATABASE CHARACTER SET UTF8;
```

- To convert an existing table with different encoding, use:

```
ALTER TABLE tab CONVERT TO CHARACTER SET utf8 COLLATE utf8_unicode_ci;
```

Note that `CONVERT TO` is critical to do the actual encoding conversion.

- Make sure your tables use the InnoDB engine. You can make sure that it is so by adding this line to your database configuration:

```
'OPTIONS': {'init_command': 'SET storage_engine=INNODB',}
```

More. Note that MySQL 5.5 (and probably 5.1) have already set InnoDB as the default engine).

- You can make the InnoDB engine the default one in my.cnf file (if you're on MySQL <= 5.0), and you don't even have to modify the global my.cnf but use a custom config file for your Django project.
- In-memory database for tests, and also this. Rewritten in a cleaner way:

```
stop mysql
mount -t tmpfs -o size=400M tmpfs /tmp/ramdisk/
cp /var/lib/mysql /tmp/ramdisk/
mount --bind /tmp/ramdisk/ /var/lib/mysql
start mysql
```

- Speed tuning:
  - http://www.mysqlperformanceblog.com/2010/02/28/maximal-write-througput-in-mysql/
  - http://www.stereoplex.com/blog/speeding-up-django-unit-test-runs-with-mysql
  - http://www.stereoplex.com/blog/speeding-up-django-unit-test-runs-with-mysql
  - http://www.mysqlperformanceblog.com/2007/11/01/innodb-performance-optimization-basics/
  - http://www.mysqlperformanceblog.com/2007/11/03/choosing-innodb_buffer_pool_size/
  - http://www.mysqlperformanceblog.com/2006/09/29/what-to-tune-in-mysql-server-after-installation/
  - http://www.mysqlperformanceblog.com/2007/11/01/innodb-performance-optimization-basics/#comment-364739
  - Disable logging, slow-logging, binary log etc.
- Watch out for problems:
  - http://stackoverflow.com/questions/2235318/how-do-i-deal-with-this-race-condition-in-django/2235624#2235624
  - http://stackoverflow.com/questions/2221247/why-doesnt-this-loop-display-an-updated-object-count-every-five-seconds/2221400#2221400
  - http://www.no-ack.org/2010/07/mysql-transactions-and-django.html
  - http://www.no-ack.org/2011/05/broken-transaction-management-in-mysql.html
  - QuerySet.get_or_create() is clumsy anyway

1.4 Forms

Smart handling of forms in views (Credits go to PyDanny&Co). Instead of this:

```
def my_view(request):
    if request.method == 'POST':
        form = MyForm(request.POST)
        if form.is_valid():
            form.hooray()
            return HttpResponseRedirect('/success/')
    else:
        form = MyForm()
    return render_to_response('my_template.html', {'form': form})
```

do this:

```
def my_view(request):
    form = MyForm(request.POST or None)
    if form.is_valid():
        form.hooray()
        return HttpResponseRedirect('/success/')
    return render_to_response('my_template.html', {'form': form})
```

The catch here is that `form.is_valid()` returns `False` for unbound forms.

1.5 Rarely-known (and/or undocumented) Django features

- When converting *models.py* into a Python package, make sure that models there have `app_label` set in their Meta:

  ```
  class Meta:
      app_label = 'app-name'
  ```

  Without this trick Django won't see the models.

- form.Form.has_changed() - checks if form data is different than the initial data
- django.utils.html.linebreaks(...) - converts newlines into \<p\> and \<br\> tags
- django.utils.html.urlize(...) - safely converts URLs into clickable links. This is a hard task otherwise:
  1. http://stackoverflow.com/questions/37684/how-to-replace-plain-urls-with-links
  2. http://www.codinghorror.com/blog/2008/10/the-problem-with-urls.html
  3. http://www.ietf.org/rfc/rfc1738.txt
  4. http://www.codinghorror.com/blog/2008/08/protecting-your-cookies-httponly.html
- model.Meta.order_with_respect_to - adds an additional field to the model, purely for ordering purposes. The code behind this feature:
  1. https://github.com/django/django/blob/1.3.2/django/db/models/base.py#L227
  2. https://github.com/django/django/blob/1.3.2/django/db/models/base.py#L532
  3. https://github.com/django/django/blob/1.3.2/django/db/models/base.py#L603
  4. https://github.com/django/django/blob/1.3.2/django/db/models/base.py#L860
  5. https://github.com/django/django/blob/1.3.2/django/db/models/options.py#L114
  6. https://github.com/django/django/blob/1.3.2/django/db/models/fields/proxy.py
- Check the difference between `Model.objects.filter(a__x=1, a__y=2)` and `Model.objects.filter(a__x=1).filter(a__y=2)`
- A neat trick with aggregation and filtering - if `.filter()` precedes `.annotate()` then the annotation is applied only to the filtered elements.

1.6 REST, HTTP and Django

1.6.1 URLs, application structure

- A good practice is to design your URL structure so that it more or less follows the de facto standard convention. Note that this is mostly about "ordnung", not about being RESTful. It's very hard, if not impossible, to write a RESTful service - and if you violate any of the REST principles, you're not RESTful anymore. So just accept that and follow whatever is reasonable.
- Still not convinced that REST is not what it appears to be (i.e. a way of naming URLs)? Check these resources (in random order): S.O. thread #1, Roy Fielding's article, S.O. thread #2, Example of RESTful web service design.
- Specifically, Django sessions are not RESTful so to speak (check these: [1], [2], [3]). But they are great otherwise, so why not use them? Web development is not a purity contest!

- Still, adopting parts of the REST philosophy is a good idea. Some readings: [1], [2], [3], [4].
- `Get lost, my website is RESTful!!!!!` collapses if only it uses HTML forms. For illustration - let's imagine that we want to add books to a catalog. To create a new book resource you POST data to `/books/` collection. If there is any error, you can get one of the HTTP error codes. If the new book resource is created, you get #201 response.

  Now, that's not how it works in Django (or any other web framework)! In Django, if there is any form validation error, a normal (i.e. #200) response is returned, just with some additional HTML markup for presenting errors to the user. And even if the new book resource is created, a #302 redirect is returned. Moreover, you POST to the very same URL which you get the form from - and not to the `/books/` collection!

  Why do we have here such a big deviation from how it should look like in a RESTful case?

  The answer is simple - the HTML form is kind of a separate application, a user interface to the server-side service - in the old days it would just be a standalone program. It's simply a coincidence (or signum temporis) that now it's a part of the same web application.

  The moment we abandon the POST-REDIRECT-GET paradigm, and start POSTing forms to the backend using AJAX requests, we have a much cleaner separation of the user interface part and the underlying RESTful (or pseudo-RESTful) service. Only that the application is hooked to an URL in the same URL space..

  So what to do about that? Just treat forms as non-RESTful parts, separate applications that happen to live in the same house. Use a consistent URL naming for them, like `/books/1/edit`, and don't think about them more.
- Some back up for what I've written above: [1], [2], [3], [4], [5].
- Some more reading about "RESTful" URLs: [1], [2].
- Which HTTP error codes to use? Here's the answer. Ok ok, I know :-)
- But seriously, there are some rules that are worth following.
- `HttpResponseBadRequest [400]` seems to be a good choice when Django view is reached but request parameters are invalid. Here are some good discussions on that.
- `HttpResponseForbidden [403]` seems like a good choice to indicate that authentication is needed in a situation when redirection to the login page doesn't make sense - e.g. for AJAX requests. Note that there is also 401 code, but it is meant to be used for the purposes of HTTP authentication, and not a custom one. (A nice discussion)

1.6.2 Django and HTTP caching for static assets

- Introduction to HTTP caching
- Use an asset manager. There is one shipped with Django 1.3+ (django.contrib.staticfiles) but it's not too powerful
  - Pick your favourite one from django-pluggables
  - A pretty great one is (was?) django-mediagenerator (Hopefully someone will maintain it)
  - Your picked assed manager should be able to:
    - Combine & minimize CSS and JS scripts, preferably using YUI Compressor and/or Google Closure Compiler
    - Version the assets, i.e. give them unique names like `sitescripts.1fhdysjnry46.js` - this is required to efficiently cache them
    - Now, you want your web server to serve the assets with one of these headers:

```
Expires: (now + 1 year)
Cache-Control: public, max-age=31536000
```

plus this one:

```
Last-Modified: {{ date }}
```

* Thanks to the above headers, the browser caches the assets for up to one year - and in case it wants to check if an asset has changed, it sends a conditional request (using `If-Modified-Since` header) that makes it possible for the web server to reply with `304 Not Modified` status code.
* Perfect caching headers
* Even more, from Yahoo
* In Apache one need to add something like this to the virtual host definition (after making sure that the relevant modules are loaded):

```
<Directory /my/project/dir/_generated_media>
    ExpiresActive On
    ExpiresDefault "access plus 1 year"
    Header merge Cache-Control "public"
    Header unset Etag
    FileETag None
</Directory>
```

* That's basically all - for static assets there is no need to worry about things like proxy caches storing sensitive data etc.
* Ah, one more thing - you probably want to have `Keep-Alive` on for static assets, but it's not that good for your Django application. So better think about some nginx. Useful link
* Btw do not get frustrated if the caching doesn't work when you refresh the page using F5. That's a known issue.

1.6.3 HTTP caching for Django views

- There's probably no single setup suitable for all your views (pages)
- So let me just give you a few links:
  - Caching in IE9 Take a look at Vary-related issues, HTTPS caching, redirect caching etc.. It's not trivial to set it all up properly.
  - Controlling HTTP caching from Django
  - django.utils.cache module
- Because of all these things to consider, if you don't have enough manpower to handle it properly, I think that it's not that unreasonable to just disable HTTP caching using something like this (idea borrowed from Google Docs):

```
response['Cache-Control'] = 'no-cache, no-store, max-age=0, must-revalidate'
response['Expires'] = 'Fri, 01 Jan 2010 00:00:00 GMT'
```

- Otherwise you would have to make sure that there's no leak of sensitive data, no old content is presented to users etc. (Btw using `must-revalidate` causes the back button in the browser to refresh (reload) the page when pressed.)

1.6.4 Useful links

- HTTP 1.1 - RFC 2616
- Cache-Control summary

1.6.5 Other HTTP performance tips

- Read Yahoo guidelines
- Read Google guidelines
- Use YSlow, PageSpeed or even "Audits" tool from Chrome inspector to learn what are the bottlenecks of your site
- There are also other online: Pingdom, Redbot
- One thing that I think is interesting: optimize the order of stylesheets and scripts
- Remember, performance is a feature!

1.7 Non-HTTP caching and Django

- Learn to use the cache framework
- Employ template source caching - look for `django.template.loaders.cached.Loader`
- Consider using two-phased template rendering
- Try Redis <http://redis.io/, it's more powerful than 'Memcached and not slower. Even if you're not impressed by its command set it has one major advantage over Memcached...
- ...which is the persistent storage. It's great not only because of being persistent, but also because it allows to decrease the chances of learning dog piling aka thundering herd problem in practice. If you can dump your cached data and reload it later, then server crashes or restarts don't hurt that much.
- A nice Redis tutorial
- Btw, the thundering herd problem is related also to the normal usage of the cache - check django-newcache's README.

1.8 Avoid Apache :)

- Apache is a mature and stable piece of software...
- ...but it's also a complex one. It's not that hard to leave a security hole or misconfigure it:
  - MPM vs Prefork
  - mod_wsgi embedded vs daemon mode
  - Are you sure /etc/passwd is not exposed? I'm never sure :) Apache "thinks" in terms of files and folders so there might be a way (i.e. URL) to access sensitive data.
  - http://stackoverflow.com/questions/6248772/should-django-python-apps-be-stored-in-the-web-server-document-root/6249943#6249943
  - http://stackoverflow.com/questions/5021424/mod-wsgi-daemon-mode-wsgiapplicationgroup-and-python-interpreter-separation

- nginx is simpler and is the preferred server for static assets anyway.
- Btw use `KeepAlive=0` for wsgi apps (to not run out of connections) vs `KeepAlive=1` for static assets (to speed up serving them)

CHAPTER

TWO

DJANGO 1.4 GOTCHAS

- Password hasing makes unit tests very slow. The solution is to switch back to MD5 hashing during when running tests:

```
if sys.argv[1] == 'test':
    PASSWORD_HASHERS = ('django.contrib.auth.hashers.MD5PasswordHasher',)
```

CHAPTER

THREE

BROWSERS, HTML5 & JAVASCRIPT

3.1 The hashbang hell

- http://danwebb.net/2011/5/28/it-is-about-the-hashbangs
- http://isolani.co.uk/blog/javascript/BreakingTheWebWithHashBangs
- http://webmasters.stackexchange.com/questions/32472/pros-cons-of-hash-navigation-from-seo-perspective

3.2 HTML5

I've spent some time looking for the best explanations of different aspects of HTML5. Here are my findings.

3.2.1 General

- http://mathiasbynens.be/notes/html5-levels
- http://html5doctor.com/avoiding-common-html5-mistakes/

3.2.2 Outlining

- New document outlines - sectioning flowchart (source)
- http://html5doctor.com/the-section-element/
- http://html5doctor.com/the-article-element/
- Sections and outline
- When to use sections
- http://stackoverflow.com/questions/8734350/html5-structure-article-section-and-div-usage
- http://stackoverflow.com/questions/6947489/html5-appropriate-use-of-article-tag

3.2.3 Headings

- In general it seems that <header> tag is optional it's only meant to wrap a single <h1> tag. <h1> tag sort of implies <header> around it.
- http://html5doctor.com/the-header-element/ - http://html5doctor.com/the-header-element/#comment-5769

- http://stackoverflow.com/questions/7712871/difference-between-heading-inside-section-or-before-it-in-html5
- http://stackoverflow.com/questions/7796367/why-does-the-html5-header-element-require-a-h-tag
- http://stackoverflow.com/questions/4837269/html5-using-header-or-footer-tag-twice
- http://stackoverflow.com/questions/9663559/html5-section-headings
- http://www.w3.org/TR/html5/the-header-element.html#the-header-element
- http://www.w3.org/TR/html5/the-h1-h2-h3-h4-h5-and-h6-elements.html#the-h1-h2-h3-h4-h5-and-h6-elements
- http://www.w3.org/TR/html5/the-hgroup-element.html#the-hgroup-element
- http://www.w3.org/TR/html5/content-models.html#heading-content-0 (note no <header> tag!)
- http://www.w3.org/TR/html5/headings-and-sections.html#headings-and-sections

3.3 Browsers' bfcache

- Firefox has so called bfcache ("Back-Forward Cache") that keeps the state of the whole page, including JavaScript context, and restores it when user presses the Back button. This is separate from the in-browse page (HTTP) cache which stores only the initial page data, as sent by the server. More on this here, here.
- Example of how bfcache works.
- Bfcache in Opera.
- Bfcache in WebKit I.
- Bfcache in WebKit II.
-

3.4 jQuery Mobile

- https://github.com/jquery/jquery-mobile/issues/1571#issuecomment-1602190
-

3.5 JavaScript

- JS has some evil parts, use CoffeeScript (also protects from RSI ;))

CHAPTER

FOUR

PYTHON 2.X RARITIES

4.1 Slicing, extended slicing, *Ellipsis* - `a[i:j:step]`, `a[i:j, k:l]`, `a[..., i:j]`

More: [1], [2], [3].

```
>>> class C(object):
... def __getitem__(self, sli):
... print sli

>>> c = C()
>>> c[2, 1:3, 1:4:6, ..., 4:, :6, :, ::-1]
(2, slice(1, 3, None), slice(1, 4, 6), Ellipsis, slice(4, None, None), slice(None, 6, None), slice(No
```

4.2 NotImplemented

Special value which can be returned by the "rich comparison" special methods (__eq__(), __lt__(), and friends), to indicate that the comparison is not implemented with respect to the other type..

NotImplemented and reflected operands.

4.3 iter(obj, sentinel)

The iter(callable, until_value) function repeatedly calls callable and yields its result until until_value is returned.

Example: `for line in iter(f.read(), '\n'):  ...`

4.4 Rot13 source encoding

http://stackoverflow.com/questions/101268/hidden-features-of-python/1024693#1024693

4.5 Negative *round()*

Negative precision affects digits in front of the decimal point:

```
>>> str(round(1234.5678, -2))
'1200.0'
>>> str(round(1234.5678, 2))
'1234.57'
```

4.6 Reversing a string or a list (well, a sequence)

It's is as simple as making a copy of it with negative increment: `sequence[::-1]` - which is equivalent to `sequence[-1::-1]` (see: Extended slices).

CHAPTER

FIVE

PYTHON 2.X TYPE SYSTEM, METACLASSES AND MORE

5.1 General information

- Basic fact: *EVERYTHING IS AN OBJECT*
- **Object** is an instance of a **class**, which is called its type: `type(x) is x.__class__` /always True/
- Each&every **class** object inherits directly or indirectly from root base **class** `object`
- Thus each&every **object** (i.e. class instance) is a direct or indirect instance of `object` class: `isinstance(x, object) is True` /always/
- **Classes** are also **objects**, therefore they also are instances of (other) classes (called metaclasses) *[My own idea: objects can be mentally split into ()* "plain" objects and (*) class objects (kind of plain objects with additional class stuff attached to them)]*
- Because every **object**, including **class** object, has its **class** `x.__class__`, and that **class** has its own **class** `x.__class__.__class__`, the chain would be infinite. As a solution, there is a **class** named `type` which is its own type, i.e. `type.__class__ is type` - that `type` class works as type of types (sth like "the ultimate type")

5.2 Built-in types

For most built-in types the following relationships occur:

```
type(1) is int; int.__bases__ == (object,); type(int) is type; int.__class__ is type
type(1.0) is float; float.__bases__ == (object,); type(float) is type; float.__class__ is type
type(Ellipsis) is ellipsis; ellipsis.__bases__ == (object,); type(ellipsis) is type; # note that 'el
type(lambda:1) is function; function.__bases__ == (object,); type(function) is type; # same as with
```

As for strings, it's the same after taking into account one minor detail:

```
type('text') is str; str.__bases__ == (basestring,); basestring.__bases__ == (object,); type(str) is
str.__class__ is type; basestring.__class__ is type;
type(u'text') is unicode; unicode.__bases__ == (basestring,); type(unicode) is type
```

There is one edge case on the top of class hierarchy: `type` inherits from `object` (which is the root base class for all other classes; doesn't inherit from anything else), while `object` is instance of `type`:

```
object.__bases__ == ()             # object is a root base class
type.__bases__ == (object,)        # object is a root base class, so type has to inherit from it
object.__class__ is type           # object is an instance of the type of all types, i.e. type
type.__class__ is type             # type is a type of itself
isinstance(type, object) is True   # type class object is an instance of object class
isinstance(object, type) is True   # object class object is an instance of type which is a descendant
```

5.3 Prerequisites for the subsequent sections

"For new-style classes, implicit invocations of special methods are only guaranteed to work correctly if defined on an object's type, not in the object's instance dictionary." In other words, `C()` resolves to `C.__class__.__call__(C)` and not to `C.__call__()`. The latter `__call__` method is injected into the created `C` instance.

```
>>> type.__call__(int)
0
>>> type.__call__(int, 1)
1

>>> int.__new__(int)
0
>>> int.__new__(int, 1)
1
```

5.4 Object creation a.k.a. class instantiation

To create an object of class C one use: `c = C(...)`.

`C(...)` is a syntatic sugar for `C.__class__.__call__(...)`, ??? which is a method call on class object C, a method which is taken from class of C class (unless called explicitely as `c.__call__()` ???, more on this here) i.e. `C.__class__`, i.e. metaclass, i.e. often the built-in `type` class (uff!):

```
def __call__(self, *kargs, **kwargs):
    obj = self.__new__(self, *kargs, **kwargs)
    obj.__init__(*kargs, **kwargs)
    return obj
```

`self.__new__()` is a static method meant to create an instance of a class passed to it as a first parameter. It's often taken from `object` base class, but can be overriden in given class, to customize the creation of class instances. More on `__new__()` is here and here.

Subsequently, `self.__init__()` takes the class instance object and initializes it.

5.5 Special case of object creation: class declaration a.k.a. metaclass instantiation

The following declaration:

```
class C(object):
    a = 1
```

is nothing more than just a syntatic sugar for: `C = C.__metaclass__('C', (object,), {'a': 1})` where `__metaclass__` is determined according to this. and very often it resolves to the built-in `type` class, therefore the above can often be rewritten as: `C = type('C', (object,), {'a': 1})`.

`type('C', (object,), {'a': 1})` is a syntatic sugar for `type.__class__.__call__('C', (object,), {'a': 1})` (which can be simplified to `type.__call__('C', (object,), {'a': 1})` because `type.__class__ is type` is always true) and this is resolved like a standard object creation described in the previous section.

5.6 A more complex example of "class + metaclass + instantiation" hell

This:

```
class MetaC(type):
    def __new__(cls, *kargs, **kwargs):   # static method, called by type.__call__() to create MetaC
        print 'MetaC.__new__:', cls, kargs, kwargs
        return type.__new__(cls, *kargs, **kwargs)  # this is *most probably* inherited from 'object

    def __init__(self, *kargs, **kwargs):   # instance method, called to initialize MetaC instance,
        print 'MetaC().__init__:', self, kargs, kwargs

class C(object):   # equivalent to: C = MetaC('C', (object,), {'__metaclass__': MetaC})
        __metaclass__ = MetaC
```

gives in the interactive shell:

```
MetaC.__new__: <class '__main__.MetaC'> ('C', (<type 'object'>,), {'__module__': '__main__', '__meta

MetaC().__init__: <class '__main__.C'> ('C', (<type 'object'>,), {'__module__': '__main__', '__metac
```

5.7 Another - even more complex - example of "class + metaclass + instantiation" hell

This:

```
class MetaC(type):  # equivalent to: MetaC = MetaC('MetaC', (type,), {'__metaclass__': MetaC})
    __metaclass__ = MetaC  # MetaC is own metaclass!

    def __call__(cls, *kargs, **kwargs):
        print 'MetaC.__call__:', cls, kargs, kwargs
        return type.__call__(cls, *kargs, **kwargs)

    def __new__(cls, *kargs, **kwargs): # this is *most probably* inherited from 'object' class
        print 'MetaC.__new__:', cls, kargs, kwargs
        return type.__new__(cls, *kargs, **kwargs)

    def __init__(self, *kargs, **kwargs):
        print 'MetaC().__init__:', self, kargs, kwargs
```

gives in the interactive shell:

```
MetaC.__call__: <class '__main__.MetaC'> ('MetaC', (<type 'type'>,), {'__call__': <function __call__
MetaC.__new__: <class '__main__.MetaC'> ('MetaC', (<type 'type'>,), {'__call__': <function __call__ a
MetaC().__init__: <class '__main__.MetaC'> ('MetaC', (<type 'type'>,), {'__call__': <function __call_
```

5.8 Further reading

- http://python.org/doc/newstyle/
- http://docs.python.org/reference/datamodel.html, especially http://docs.python.org/reference/datamodel.html#customizing-class-creation
- http://stackoverflow.com/questions/395982/metaclass-new-cls-and-super-can-someone-explain-the-mechanism-exa/396109
- http://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python, http://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python/6581949#6581949
- http://stackoverflow.com/questions/3798835/understanding-get-and-set-and-python-descriptors
- http://docs.python.org/reference/datamodel.html#implementing-descriptors
- http://docs.python.org/howto/descriptor.html#invoking-descriptors
- http://docs.python.org/reference/datamodel.html#special-method-lookup-for-new-style-classes
- http://docs.python.org/reference/datamodel.html#more-attribute-access-for-new-style-classes
- https://groups.google.com/forum/#!topic/secrets-of-the-framework-creators/UTCMHguEhKs
- http://users.rcn.com/python/download/Descriptor.htm
- Python descriptors/descriptor protocol: http://users.rcn.com/python/download/Descriptor.htm, http://docs.python.org/howto/descriptor.html, http://martyalchin.com/2007/nov/23/python-descriptors-part-1-of-2/
- Descriptors vs bound/unbound methods: http://stackoverflow.com/questions/1015307/python-bind-an-unbound-method, http://stackoverflow.com/questions/114214/class-method-differences-in-python-bound-unbound-and-static/114289#114289, http://stackoverflow.com/questions/114214/class-method-differences-in-python-bound-unbound-and-static/114289#114289

CHAPTER

SIX

USEFUL SERVICES

6.1 Logs, monitoring, metrics

- http://newrelic.com/
- http://airbrake.io
- http://www.exceptional.io/
- http://graylog2.org/about
- https://www.metricfire.com/
- http://loggly.com/
- http://www.statsmix.com/
- https://www.getsentry.com
- https://scoutapp.com/
- https://papertrailapp.com/

6.2 PaaS

- http://cloudfoundry.com/
- https://openshift.redhat.com
- http://appfog.com
- http://dotcloud.com
- http://www.activestate.com/stackato

6.3 CI

- https://www.shiningpanda-ci.com/

6.4 Usability/browser testing

- http://www.feedbackarmy.com/
- http://www.browserstack.com/

6.5 Misc APIs

- http://www.fullcontact.com/
- http://pusher.com/
- http://chart.io/
- http://www.elasticsearch.org/

6.6 Private cloud storage

- https://owncloud.com/

CHAPTER

SEVEN

MISCELLANEOUS TIPS&TRICKS

7.1 Postfix

- After updating `/etc/aliases`, in order for Postfix to see the updated aliases, `newaliases` command has to be issued in bash
- Set up a Gmail relay:
  - http://serverfault.com/questions/119278/configure-postfix-to-send-relay-emails-gmail-smtp-gmail-com-via-port-587
  - http://productforums.google.com/forum/#!category-topic/gmail/composing-and-sending-messages/7QWAO_aunhc
  - http://www.postfix.org/postconf.5.html#relay_transport
  - http://www.postfix.org/TLS_README.html
  - Postfix configuration is not that complex, there's a lot of options but the docs are well-written
  - http://www.howtoforge.com/forums/showthread.php?p=105989

7.2 git

- In the precommit hook one can add `ack-grep "pdb\.set_trace\(\)"` to find all remaining pdb calls. You can also do much more there.

7.3 Virtualbox (+ Ubuntu)

- Port mapping in the NAT mode http://superuser.com/questions/424083/virtualbox-host-ssh-to-guest. Then: `ssh -p 2222 user@localhost`.
- An imported Vbox image cannot connect to the network: https://forums.virtualbox.org/viewtopic.php?f=6&t=24383. One have to comment out entries in `/etc/udev/rules.d/70-persistent-net.rules` (in the guest OS of course) as they contain MAC address of the formerly used virtual machine, and the restart the guest.
- Using host's DNS resolver in the NAT mode: http://www.virtualbox.org/manual/ch09.html#nat_host_resolver_proxy. That makes host's `/etc/hosts` used for DNS lookups in the virtual machine.
- Watch out for DNS resolving in Ubuntu Precise (12.04+): http://www.stgraber.org/2012/02/24/dns-in-ubuntu-12-04/, https://plus.google.com/105897381673403508112/posts/UNrkEtAw6MX.

CHAPTER

EIGHT

ONLINE BOOKS

- TCP/IP Guide

CHAPTER

NINE

INDICES AND TABLES

- *genindex*
- *search*