1: Access Multiple Databases

1) The goal is to be able to access the following databases:

- MySQL

- PostgreSQL

- Microsoft SQL Server

- MongoDB

===🡺 Refer to production.py line 90

2) The environment should be configurable where the developer can easily switch from QA (Testing) and Production (Live) environments. Each environment will point to a different database instance. I should be able to add additional environements such as a Dev environment for developing purposes.

===🡺 By default Django runs on settings.py file. However, we’ve made this more flexible by changing the single settings.py file into a package.

1 The package was created by creating a new folder called ‘settings’ in the same directory as ‘settings.py’

2 In the settings folder, I created 4 files namely:

A \_\_init\_\_.py

B base.py

C db\_password.py

D development.py

E production.py

====🡺 Refer to comments on the top of these files for details on what each file holds

===🡺 To run in development mode run:

Python manage.py runserver --settings=restingblog.settings.development.py

==🡺 To run in production mode run

Python manage.py runserver –settings=restingblog.settings.production.py

3) I would like to see a mechanism to load classes/ modules at the top of any python script that requires it. There should be a centralize location where I can add classes/ modules

===🡺 First off, classes are different from module in python so their import statements vary with respect to the type of package.

However, to load classes classes or modules on top of python scripts that require it can be achieved using the import statement. For example in the settings package I mentioned earlier

I created a bas.py file that contains settings similar to development and production

Then in development and production, I included settings in each that only suits it’s purpose but first I imported the the content of the base,py file using

**From .base import \***

import my\_module

Note that I didn’t include the .py extention while importing also the \* tells python n that I’d like to import everything contained in the base.py file.

Also you can import classes a=or modules in other packages using the dot notation e.g

**django.template.context\_processors.debug**

from my\_package.timing.danger.internets import function\_of\_love

**restingblog.settings.production** (without the .py file. This is equivalent to restingblog/settings/production.py provided the folders restingblog and settings both contain a \_\_init.py file, the \_\_init\_\_.py file and either be empty or contain codes

The primary use of \_\_init\_\_.py is to initialize Python packages. The easiest way to demonstrate this is to take a look at the structure of a standard Python module.

\_\_init\_\_.py can be an empty file but it is often used to perform setup needed for the package(import things, load things into path, etc).

One common thing to do in your \_\_init\_\_.py is to import selected Classes, functions, etc into the package level so they can be convieniently imported from the package.

In our example above we can say that file.py has the Class File. So without anything in our \_\_init\_\_.py you would import with this syntax:

**from package.file import File**

settings/

\_\_init\_\_.py

base.py

development.py

production.py

package\_name/

\_\_init\_\_.py

class1.py

class2.py

class3.py

...

package/

\_\_init\_\_.py

file.py

file2.py

file3.py

subpackage/

\_\_init\_\_.py

submodule1.py

submodule2.py

However you can import File into your \_\_init\_\_.py to make it available at the package level:

# in your \_\_init\_\_.py

from file import File

# now import File from package

**from package import File**

Another thing to do is at the package level make subpackages/modules available with the \_\_all\_\_ variable. When the interpeter sees an \_\_all\_\_ variable defined in an \_\_init\_\_.py it imports the modules listed in the \_\_all\_\_ variable when you do:

**from package import \***

\_\_all\_\_ is a list containing the names of modules that you want to be imported with import \* so looking at our above example again if we wanted to import the submodules in subpackage the \_\_all\_\_ variable in subpackage/\_\_init\_\_.py would be:

**\_\_all\_\_ = ['submodule1', 'submodule2']**

With the \_\_all\_\_ variable populated like that, when you perform

**from subpackage import \***

it would import submodule1 and submodule2.

As you can see \_\_init\_\_.py can be very useful besides its primary function of indicating that a directory is a module.

Ref: <http://chimera.labs.oreilly.com/books/1230000000393/ch10.html#_discussion_169>

Difference between classes and modules reference:

<https://mail.python.org/pipermail/tutor/2004-July/030298.html>

<http://learnpythonthehardway.org/book/ex40.html>

<http://stackoverflow.com/questions/7948494/whats-the-difference-between-a-python-module-and-a-python-package>

<http://stackoverflow.com/questions/1250779/python-class-vs-module-attributes>

<http://code.activestate.com/lists/python-tutor/29820/>

<https://www.quora.com/What-is-the-difference-between-a-python-module-package-And-when-to-use-which-one>

4) I would like to be able to write custom classes and methods that accept dynamic arguments and be able to access them easily during any development.

Below is an example:

class Shape:

def name(self):

print("Shape")

class Circle(Shape):

def name(self):

print("Circle")

# Call name method from parent class.

super().name()

Here is an example of a simple custom class which stores information about a person:

import datetime # we will use this for date objects

class Person:

def \_\_init\_\_(self, name, surname, birthdate, address, telephone, email):

self.name = name

self.surname = surname

self.birthdate = birthdate

self.address = address

self.telephone = telephone

self.email = email

def age(self):

today = datetime.date.today()

age = today.year - self.birthdate.year

if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):

age -= 1

return age

person = Person(

"Jane",

"Doe",

datetime.date(1992, 3, 12), # year, month, day

"No. 12 Short Street, Greenville",

"555 456 0987",

"jane.doe@example.com"

)

print(person.name)

print(person.email)

print(person.age())

We start the class definition with the class keyword, followed by the class name and a colon. We would list any parent classes in between round brackets before the colon, but this class doesn’t have any, so we can leave them out.

Inside the class body, we define two functions – these are our object’s methods. The first is called \_\_init\_\_, which is a special method. When we call the class object, a new instance of the class is created, and the \_\_init\_\_ method on this new object is immediately executed with all the parameters that we passed to the class object. The purpose of this method is thus to set up a new object using data that we have provided.

The second method is a custom method which calculates the age of our person using the birthdate and the current date.

Note

\_\_init\_\_ is sometimes called the object’s constructor, because it is used similarly to the way that constructors are used in other languages, but that is not technically correct – it’s better to call it the initialiser. There is a different method called \_\_new\_\_ which is more analogous to a constructor, but it is hardly ever used.

You may have noticed that both of these method definitions have self as the first parameter, and we use this variable inside the method bodies – but we don’t appear to pass this parameter in. This is because whenever we call a method on an object, the object itself is automatically passed in as the first parameter. This gives us a way to access the object’s properties from inside the object’s methods.

In some languages this parameter is implicit – that is, it is not visible in the function signature – and we access it with a special keyword. In Python it is explicitly exposed. It doesn’t have to be called self, but this is a very strongly followed convention.

Now you should be able to see that our \_\_init\_\_ function creates attributes on the object and sets them to the values we have passed in as parameters. We use the same names for the attributes and the parameters, but this is not compulsory.

The age function doesn’t take any parameters except self – it only uses information stored in the object’s attributes, and the current date (which it retrieves using the datetime module).

Note that the birthdate attribute is itself an object. The date class is defined in the datetime module, and we create a new instance of this class to use as the birthdate parameter when we create an instance of the Person class. We don’t have to assign it to an intermediate variable before using it as a parameter to Person; we can just create it when we call Person, just like we create the string literals for the other parameters.

Remember that defining a function doesn’t make the function run. Defining a class also doesn’t make anything run – it just tells Python about the class. The class will not be defined until Python has executed the entirety of the definition, so you can be sure that you can reference any method from any other method on the same class, or even reference the class inside a method of the class. By the time you call that method, the entire class will definitely be defined.

It is important to note that the attributes set on the object in the \_\_init\_\_ function do not form an exhaustive list of all the attributes that our object is ever allowed to have.

In some languages you must provide a list of the object’s attributes in the class definition, placeholders are created for these allowed attributes when the object is created, and you may not add new attributes to the object later. In Python, you can add new attributes, and even new methods, to an object on the fly. In fact, there is nothing special about the \_\_init\_\_ function when it comes to setting attributes. We could store a cached age value on the object from inside the age function:

def age(self):

if hasattr(self, "\_age"):

return self.\_age

today = datetime.date.today()

age = today.year - self.birthdate.year

if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):

age -= 1

self.\_age = age

return age

Note

Starting an attribute or method name with an underscore (\_) is a convention which we use to indicate that it is a “private” internal property and should not be accessed directly. In a more realistic example, our cached value would sometimes expire and need to be recalculated – so we should always use the age method to make sure that we get the right value.

We could even add a completely unrelated attribute from outside the object:

person.pets = ['cat', 'cat', 'dog']

It is very common for an object’s methods to update the values of the object’s attributes, but it is considered bad practice to create new attributes in a method without initialising them in the \_\_init\_\_ method. Setting arbitrary properties from outside the object is frowned upon even more, since it breaks the object-oriented paradigm

The \_\_init\_\_ method will definitely be executed before anything else when we create the object – so it’s a good place to do all of our initialisation of the object’s data. If we create a new attribute outside the \_\_init\_\_ method, we run the risk that we will try to use it before it has been initialised.

In the age example above we have to check if an \_age attribute exists on the object before we try to use it, because if we haven’t run the age method before it will not have been created yet. It would be much tidier if we called this method at least once from \_\_init\_\_, to make sure that \_age is created as soon as we create the object.

Initialising all our attributes in \_\_init\_\_, even if we just set them to empty values, makes our code less error-prone. It also makes it easier to read and understand – we can see at a glance what attributes our object has.

An \_\_init\_\_ method doesn’t have to take any parameters (except self) and it can be completely absent.

getattr, setattr and hasattr¶

What if we want to get or set the value of an attribute of an object without hard-coding its name? We may sometimes want to loop over several attribute names and perform the same operation on all of them, as we do in this example which uses a dictionary:

**for key in ["a", "b", "c"]:**

**print(mydict[key])**

How can we do something similar with an object? We can’t use the . operator, because it must be followed by the attribute name as a bare word. If our attribute name is stored as a string value in a variable, we have to use the getattr function to retrieve the attribute value from an object:

**for key in ["a", "b", "c"]:**

**print(getattr(myobject, key, None))**

Note that getattr is a built-in function, not a method on the object: it takes the object as its first parameter. The second parameter is the name of the variable as a string, and the optional third parameter is the default value to be returned if the attribute does not exist. If we do not specify a default value, getattr will raise an exception if the attribute does not exist.

Similarly, setattr allows us to set the value of an attribute. In this example, we copy data from a dictionary to an object:

**for key in ["a", "b", "c"]:**

**setattr(myobject, key, mydict[key])**

The first parameter of setattr is the object, the second is the name of the function, and the third is the new value for the attribute.

As we saw in the previous age function example, hasattr detects whether an attribute exists.

There’s nothing preventing us from using getattr on attributes even if the name can be hard-coded, but this is not recommended: it’s an unnecessarily verbose and round-about way of accessing attributes:

**getattr(myobject, "a")**

# means the same thing as

myobject.a

You should only use these functions if you have a good reason to do so.

Reference: <http://python-textbok.readthedocs.io/en/1.0/Classes.html>

Reference: <http://www.tutorialspoint.com/python/python_classes_objects.htm>

Also here’s a simple example:

Also, we use them for managing inheritance.

class Super( object ):

def \_\_init\_\_( self, this, that ):

self.this = this

self.that = that

class Sub( Super ):

def \_\_init\_\_( self, myStuff, \*args, \*\*kw ):

super( Sub, self ).\_\_init\_\_( \*args, \*\*kw )

self.myStuff= myStuff

x= Super( 2.7, 3.1 )

y= Sub( "green", 7, 6 )

This way Sub doesn't really know (or care) what the superclass initialization is. Should you realize that you need to change the superclass, you can fix things without having to sweat the details in each subclass.

Putting \*args and/or \*\*kwargs as the last items in your function definition’s argument list allows that function to accept an arbitrary number of arguments and/or keyword arguments.

For example, if you wanted to write a function that returned the sum of all its arguments, no matter how many you supply, you could write it like this:

def my\_sum(\*args):

return sum(args)

It’s probably more commonly used in object-oriented programming, when you’re overriding a function, and want to call the original function with whatever arguments the user passes in.

You don’t actually have to call them args and kwargs, that’s just a convention. It’s the \* and \*\*that do the magic.

The official Python documentation has [a more in-depth look](http://docs.python.org/2/tutorial/controlflow.html#more-on-defining-functions)

MORE EXAMPLES:

|  |  |
| --- | --- |
|  | [The syntax is the \* and \*\*](http://docs.python.org/tutorial/controlflow.html#arbitrary-argument-lists). The names \*args and \*\*kwargs are only by convention but there's no hard requirement to use them.  You would use \*args when you're not sure how many arguments might be passed to your function, i.e. it allows you pass an arbitrary number of arguments to your function. For example:  >>> def print\_everything(\*args):  for count, thing in enumerate(args):  ... print '{0}. {1}'.format(count, thing)  ...  >>> print\_everything('apple', 'banana', 'cabbage')  0. apple  1. banana  2. cabbage  Similarly, \*\*kwargs allows you to handle named arguments that you have not defined in advance:  >>> def table\_things(\*\*kwargs):  ... for name, value in kwargs.items():  ... print '{0} = {1}'.format(name, value)  ...  >>> table\_things(apple = 'fruit', cabbage = 'vegetable')  cabbage = vegetable  apple = fruit  You can use these along with named arguments too. The explicit arguments get values first and then everything else is passed to \*args and \*\*kwargs. The named arguments come first in the list. For example:  def table\_things(titlestring, \*\*kwargs)  You can also use both in the same function definition but \*args must occur before \*\*kwargs.  You can also use the \* and \*\* syntax when calling a function. For example:  >>> def print\_three\_things(a, b, c):  ... print 'a = {0}, b = {1}, c = {2}'.format(a,b,c)  ...  >>> mylist = ['aardvark', 'baboon', 'cat']  >>> print\_three\_things(\*mylist)  a = aardvark, b = baboon, c = cat  As you can see in this case it takes the list (or tuple) of items and unpacks it. By this it matches them to the arguments in the function. Of course, you could have a \* both in the function definition and in the function call. |

Basically, here’s the format

class Foo(object):

def \_\_init\_\_(self, value1, value2):

# do something with the values

print value1, value2

class MyFoo(Foo):

def \_\_init\_\_(self, \*args, \*\*kwargs):

# do something else, don't care about the args

print 'myfoo'

super(MyFoo, self).\_\_init\_\_(\*args, \*\*kwargs)

reference: <http://stackoverflow.com/questions/3394835/args-and-kwargs>’

5) Create examples of simple and complex queries that can be performed for the following actions (GET, PUT, DELETE, POST, HEAD, OPTIONS) for all the databases

Referencing: <https://docs.djangoproject.com/en/1.9/ref/class-based-views/base/>

**Example views.py**:

**from** **django.http** **import** HttpResponse

**from** **django.views.generic** **import** View

**class** **MyView**(View):

**def** get(self, request, \*args, \*\*kwargs):

**return** HttpResponse('Hello, World!')

**Example urls.py**:

**from** **django.conf.urls** **import** url

**from** **myapp.views** **import** MyView

urlpatterns = [

url(r'^mine/$', MyView.as\_view(), name='my-view'),

]

**Attributes**

**http\_method\_names**[**¶**](https://docs.djangoproject.com/en/1.9/ref/class-based-views/base/#django.views.generic.base.View.http_method_names)

The list of HTTP method names that this view will accept.

Default:

['get', 'post', 'put', 'patch', 'delete', 'head', 'options', 'trace']

**Example views.py**:

**from** **django.views.generic.base** **import** TemplateView

**from** **articles.models** **import** Article

**class** **HomePageView**(TemplateView):

template\_name = "home.html"

**def** get\_context\_data(self, \*\*kwargs):

context = super(HomePageView, self).get\_context\_data(\*\*kwargs)

#retrieve last 5 articles from the default database

context['latest\_articles'] = Article.objects.all()[:5]

#retrieve articles from the ‘named’ database

context['latest\_articles'] = Article.objects.using(‘named’).all()

#where named is the name of a database settings stated in your project settings

#retrieve one precise article using id or primary key from a named database

one\_precise\_article = Article.objects.using(‘named’).get(id = 5)

#create new article

New\_article = Article(name = ‘Article of python’, content = ‘Be ythonic in all your doing so that syntax errors might guide your way to fulfilment’)

**return** context

If any of this is not understood do contact me, so I can shed more light on it by giving a better explanation of it.

