Class Base View

Class-based views provide an alternative way to implement views as Python objects instead of functions. They do not replace function-based views.

I. Base Class-Based Views / Base View
II. Generic Class-Based Views / Generic View

Advantages:-

1. Organization of code related to specific HTTP methods (GET, POST, etc.) can be addressed by separate methods instead of conditional branching.
2. Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

Base Class-Based View

Base class-based views can be thought of as parent views, which can be used by themselves or inherited from. They may not provide all the capabilities required for projects, in which case there are Mixins which extend what base views can do.

1. View
2. Template View
3. RedirectView

View:- The master class-based base view. All other class-based views inherit from this base class. It isn't strictly a generic view and thus can also be imported from django.views.

django.views.generic.base.View

C:\Users\GS\AppData\Local\Programs\Python\Python38-32\Lib\site-packages\django\views\generic

Attribute:-
http_method_names = ['get', 'post', 'put', 'patch', 'delete', 'head', 'options', 'trace']
The list of HTTP method names that this view will accept.

Method:-

1. **setup(self, request, *args, **kwargs)** - It initializes view instance attributes: self.request, self.args, and self.kwargs prior to dispatch()

2. **dispatch(self, request, *args, **kwargs)** - The view part of the view – the method that accepts a request argument plus arguments, and returns a HTTP response.

   The default implementation will inspect the HTTP method and attempt to delegate to a method that matches the HTTP method; a GET will be delegated to get(), a POST to post(), and so on.

   By default, a HEAD request will be delegated to get(). If you need to handle HEAD requests in a different way than GET, you can override the head() method.

3. **http_method_not_allowed(self, request, *args, **kwargs)** - If the view was called with a HTTP method it doesn't support, this method is called instead.

   The default implementation returns HttpResponseNotAllowed With a list of allowed methods in plain text

4. **options(self, request, *args, **kwargs)** - It handles responding to requests for the OPTIONS HTTP verb. Returns a response with the Allow header containing a list of the view's allowed HTTP method names.

5. **as_view(cls, **initkwargs)** - It returns a callable view that takes a request and returns a response.

6. **_allowed_methods(self)**

Example:-

Function Based View (views.py)	Class Based View (views.py)

from django.http import HttpResponse
 	
def myview(request)**:**
    return HttpResponse('Function Based View')
	from django.views import View
	
	class MyView(View**):**
	    def get(self, request)**:**
	        return HttpResponse('Class Based View')

Urlpatterns = [
    Path(‘func/’, **views.**myview, name=’func’),
]
	Urlpatterns = [
	    Path(‘cl/’, **views.**MyView.as_view(),
	            name= ‘cl’),
	]

TemplateView:-

django.views.generic.base. Template View

It renders a given template, with the context containing parameters captured in the URL.

This view inherits methods and attributes from the following views:

- django.views.generic.base. TemplateResponseMixin
- django.views.generic.base.ContextMixin
- django.views.generic.base.View

class TemplateView(TemplateResponseMixin, ContextMixin, View):

TemplateResponseMixin:-

It provides a mechanism to construct a TemplateResponse, given suitable context. The template to use is configurable and can be further customized by subclasses.

Attributes:-

template_name - The full name of a template to use as defined by a string. Not defining a template_name will raise a django.core.exceptions.ImproperlyConfigured exception.

template_engine - The NAME of a template engine to use for loading the template. template_engine is passed as the using keyword argument to response_class. Default is None, which tells Django to search for the template in all configured engines.

response_class - The response class to be returned by render_to_response method. Default is TemplateResponse. The template and context of TemplateResponse instances can be altered later (e.g. in template response middleware).
If you need custom template loading or custom context object instantiation, create a TemplateResponse subclass and assign it to response_class.

content_type - The content type to use for the response. content_type is passed as a keyword argument to response_class. Default is None - meaning that Django uses 'text/html'.

Method:-

render_to_response(context, **response_kwargs)- It returns a self.response_class instance.
If any keyword arguments are provided, they will be passed to the constructor of the response class. Calls get_template_names() to obtain the list of template names that will be searched looking for an existent template.

get_template_names() - It returns a list of template names to search for when rendering the template. The first template that is found will be used.

If template_name is specified, the default implementation will return a list containing template_name (if it is specified).

ContextMixin:-

A default context mixin that passes the keyword arguments received by **get_context_data()** as the template context.

Attribute:-

extra_context - A dictionary to include in the context. This is a convenient way of specifying some context in **as_view().**

Method:-

get_context_data(kwargs)**- It returns a dictionary representing the template context. The keyword arguments provided will make up the returned context.

TemplateView Define:-

views.py

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

class HomeView(TemplateView):
template_name = 'school/home.html'
  def get_context_data(self, **kwargs):
      context = super().get_context_data(**kwargs)
      context['name'] = 'Rakib'
      context['roll'] = 1515
      return context
```

urls.py

```
from school import views
urlpatterns = [
        path('home/', views.HomeView.as_view(extra_ context ={'course': 'Django'}), name='home'),
]
```

RedirectView:-

django.views.generic.base.RedirectView

It redirects to a given URL.
The given URL may contain dictionary-style string formatting, which will be interpolated against the parameters captured in the URL. Because keyword interpolation is always done (even if no
arguments are passed in), any "%" characters in the URL must be written as "%%" so that Python will convert them to a single percent sign on output.
If the given URL is None, Django will return an HttpResponseGone (410).

This view inherits methods and attributes from the following view:

- django.views.generic.base. View

Attributes:-
url - The URL to redirect to, as a string. Or None to raise a 410 (Gone) HTTP error.

pattern_name - The name of the URL pattern to redirect to. Reversing will be done using the same and kwargs as are passed in for this view.
args
permanent - Whether the redirect should be permanent. The only difference here is the HTTP status code returned. If True, then the redirect will use status code 301. If False, then the redirect will use status code 302. By default, permanent is False.

query_string - Whether to pass along the GET query string to the new location. If True, then the query string is appended to the URL. If False, then the query string is discarded. By default, query_string is False.

Methods:-
get_redirect_url(*args, **kwargs) - It constructs the target URL for redirection.

The args and kwargs arguments are positional and/or keyword arguments captured from the URL pattern, respectively.

The default implementation uses url as a starting string and performs expansion of % named parameters in that string using the named groups captured in the URL.

If url is not set, get_redirect_url() tries to reverse the pattern_name using what was captured in the URL (both named and unnamed groups are used).

If requested by query_string, it will also append the query string to the generated URL. Subclasses may implement any behavior they wish, as long as the method returns a redirect-ready URL string.

Example:

views.py

```
from django.shortcuts import render
from django.views.generic.base import RedirectView, Template View

class ResultShows RedirectView(RedirectView):
    url='https://geekyshows.com'

class ResultRedirectView(RedirectView):
    pattern_name = 'mindex'
    permanent = False
    query_string = False

    def get_redirect_url(self, *args, **kwargs):
        print(kwargs)
        kwargs['pk'] = 16
        return super().get_redirect_url(*args, **kwargs)
```

urls.py

```
from django.urls import path
from django.views.generic.base import RedirectView, Template View
from school import views

urlpatterns = [
    path('admin/, admin.site.urls),

    path('', views. TemplateView.as_view(template_name='school/home.html'), name='blankindex'),

    path('home/, views. RedirectView.as_view(url='/'), name='home'),

    path('index/'', views.RedirectView.as_view(pattern_name='home'), name='index'),

    path('geekyshows/'),views.RedirectView.as_view(url='https://geekyshows.com'), name='go-to-geekyshows'

    path('geekyshows/', views.GeekyShowsRedirectView.as_view(), name='go-to-geekyshows'),

    path('home/<int:pk>/, views.GeekRedirectView.as_view(), name='geek'),

    path('<int:pk>/', views.TemplateView.as_view(template_name='school/home.html'), name='mindex'),

    #path('home/<slug:post>/', views.GeekRedirectView.as_view(), name='geek'),

    #path(''<slug:post>/, views. Template View.as_view(template_name='school/home.html'), name='index'),

]
```

Generic Class Based View

Django's generic views are built off of those base views, and were developed as a shortcut for common usage patterns such as displaying the details of an object.

They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to repeat yourself.

Most generic views require the queryset key, which is a QuerySet instance.

- **Display View - ListView, DetailView**
- **Editing View - FormView, CreateView, Update View, DeleteView**
- **Date Views - ArchiveIndex View, YearArchive View, MonthArchive View, WeekArchive View, DayArchive View, TodayArchive View, DateDetailView**

Display View:
The two following generic class-based views are designed to display data.

i. **ListView**
ii. **DetailView**

ListView:-

django.views.generic.list.ListView

A page representing a list of objects.

While this view is executing, self.object_list will contain the list of objects (usually, but not necessarily a queryset) that the view is operating upon.

This view inherits methods and attributes from the following views:

- django.views.generic.list.**MultipleObjectTemplateResponseMixin**
- django.views.generic.base. **TemplateResponseMixin**
- django.views.generic.list.**BaseListView**
- django.views.generic.list.**MultipleObjectMixin**
- django.views.generic.base.**View**

MultipleObjectTemplateResponseMixin:

A mixin class that performs template-based response rendering for views that operate upon a list of object instances. Requires that the view it is mixed with provides self.object_list, the list of object instances that the view is operating on. self.object_list may be, but is not required to be, a QuerySet. This inherits methods and attributes from the following views:

django.views.generic.base.TemplateResponseMixin

Attribute:-

template_name_suffix - The suffix to append to the auto-generated candidate template name. Default suffix is_list

Method:-

get_template_names() - It returns a list of candidate template names.

BaseListView:-

A base view for displaying a list of objects. It is not intended to be used directly, but rather as a parent class of the django.views.generic.list.ListView or other views representing lists of objects.
This view inherits methods and attributes from the following views:

django.views.generic.list.MultipleObjectMixin django.views.generic.base. View

Methods:-
get(request, *args, **kwargs) - It adds object_list to the context. If allow_empty is True then display an empty list. If allow_empty is False then raise a 404 error.

MultipleObjectMixin:-

django.views.generic.list.MultipleObjectMixin

A mixin that can be used to display a list of objects.

If paginate_by is specified, Django will paginate the results returned by this. You can specify the page number in the URL in one of two ways:

Use the page parameter in the URL conf.

Pass the page number via the page query-string parameter.

These values and lists are 1-based, not 0-based, so the first page would be represented as page 1.

As a special case, you are also permitted to use last as a value for page.

This allows you to access the final page of results without first having to determine how many pages there are.

Note that page must be either a valid page number or the value last; any other value for page will result in a 404 error.

Attribute:-

allow_empty - A boolean specifying whether to display the page if no objects are available. If this is False and no objects are available, the view will raise a 404 instead of displaying an empty page. By default, this is True.

model - The model that this view will display data for. Specifying **model = Student** is effectively the same as specifying **queryset = Student.objects.all**, where objects stands for Student's default manager.

queryset - A QuerySet that represents the objects. If provided, the value of queryset supersedes the value provided for model.

ordering - A string or list of strings specifying the ordering to apply to the queryset. Valid values are the same as those for **order_by()**

paginate_by - An integer specifying how many objects should be displayed per page. If this is given, the view will paginate objects with paginate_by objects per page. The view will expect either a page query string parameter (via request.GET) or a page variable specified in the URL conf.

paginate_orphans - An integer specifying the number of "overflow" objects the last page can contain. This extends the paginate_by limit on the last page by up to paginate_orphans, in order to keep the last page from having a very small number of objects.

page_kwarg - A string specifying the name to use for the page parameter. The view will expect this parameter to be available either as a query string parameter (via request.GET) or as a kwarg variable specified in the URL conf. Defaults to page.

paginator_class - The paginator class to be used for pagination. By default, django.core.paginator.Paginator is used. If the custom paginator class doesn't have the same constructor interface as django.core.paginator.Paginator, you will also need to provide an implementation for get_paginator().

context_object_name - Designates the name of the variable to use in the context.

Methods:-

get_queryset() - Get the list of items for this view. This must be an iterable and may be a queryset (in which queryset-specific behavior will be enabled).

get_ordering() - Returns a string (or iterable of strings) that defines the ordering that will be applied to the queryset.
Returns ordering by default.

paginate_queryset(queryset, page_size) - Returns a 4-tuple containing (paginator, page, object_list, is_paginated).
Constructed by paginating queryset into pages of size page_size. If the request contains a page argument, either as a captured URL argument or as a GET argument, object_list will correspond to the objects from that page.

get_paginate_by(queryset) - Returns the number of items to paginate by, or None for no pagination. By default this returns the value of paginate_by.

get_paginator(queryset, per_page, orphans=0, allow_empty_first_page=True) - Returns an instance of the paginator to use for this view. By default, instantiates an instance of paginator_class.

get_paginate_orphans() - An integer specifying the number of "overflow" objects the last page can contain. By default this returns the value of paginate_orphans.

get_allow_empty() - Return a boolean specifying whether to display the page if no objects are available. If this method returns False and no objects are available, the view will raise a 404 instead of displaying an empty page. By default, this is True.

get_context_object_name(object_list) - Return the context variable name that will be used to contain the list of data that this view is manipulating. If object_list is a queryset of Django objects and context_object_name is not set, the context name will be the model_name of the model that the queryset is composed from, with postfix '_list' appended. For example, the model Article would have a context object named article_list.

get_context_data(kwargs)** - Returns context data for displaying the list of objects.
Context

object_list: The list of objects that this view is displaying. If context_object_name is specified, that variable will also be set in the context, with the same value as object_list.

is_paginated: A boolean representing whether the results are paginated. Specifically, this is set to False if no page size has been specified, or if the available objects do not span multiple pages.

paginator: An instance of django.core.paginator.Paginator. If the page is not paginated, this context variable will be None.

page_obj: An instance of django.core.paginator.Page. If the page is not paginated, this context variable will be None.

ListView Example:-

views.py

```
from django.views.generic.list import ListView

from .models import Student

class Student ListView(ListView):
model = Student
```

urls.py

```
urlpatterns = [

        path('student/', views. StudentListView.as_view(), name='student'),

]
```

Default Template

Syntax:- AppName/ModelClassName_list.html
Example:- school/student_list.html
Default Context
Syntax:- ModelClassName_list
Example:- student_list We can also use object_list

Example 2: Default Template

views.py

```
from django.views.generic.list import ListView
from .models import Student

class StudentListView(ListView):
model = Student
template_name = 'school/student.html'
context_object_name = 'students'
```

Custom Template Name

Custom Context Name

urls.py

```
urlpatterns = [
	path('student/', views. StudentListView.as_view(), name='student'),
]
```

Note - school/students.html, school/student list.html These both will work

DetailView:

django.views.generic.detail.DetailView

While this view is executing, self.object will contain the object that the view is operating upon. This view inherits methods and attributes from the following views:

- django.views.generic.detail.SingleObjectTemplateResponseMixin
- django.views.generic.base.TemplateResponseMixin
- django.views.generic.detail.BaseDetailView
- django.views.generic.detail.SingleObjectMixin
- django.views.generic.base.View

SingleObjectTemplateResponseMixin:-

django.views.generic.detail.SingleObjectTemplateResponseMixin

A mixin class that performs template-based response rendering for views that operate upon a single object instance. Requires that the view it is mixed with provides self.object, the object instance that the view is operating on. self.object will usually be, but is not required to be, an instance of a Django model. It may be None if the view is in the process of constructing a new instance.

This view inherits methods and attributes from the following views:

- django.views.generic.base.TemplateResponseMixin

Attribute:-

template_name_field - The field on the current object instance that can be used to determine the name of a candidate template. If either template_name_field itself or the value of the template_name_field on the current object instance is None, the object will not be used for a candidate template name.

template_name_suffix - The suffix to append to the auto-generated candidate template name. Default suffix is detail.

Method:-

get_template_names()- Returns a list of candidate template names. Returns the following list: the value of template_name on the view (if provided)
the contents of the template_name_field field on the object instance that the view is operating upon (if available)
<app_label>/<model_name><template_name_suffix>.html

SingleObjectMixin:-

django.views.generic.detail.SingleObjectMixin

Provides a mechanism for looking up an object associated with the current HTTP request.

Attribute:-

model - The model that this view will display data for. Specifying model = Student is effectively the same as specifying queryset = Student.objects.all(), where objects stands for Student's default manager.

queryset - A QuerySet that represents the objects. If provided, the value of queryset supersedes the value provided for model.

slug_field - The name of the field on the model that contains the slug. By default, slug_field is 'slug'.

slug_url_kwarg - The name of the URLConf keyword argument that contains the slug. By default, slug_url_kwarg is 'slug'.

pk_url_kwarg - The name of the URLConf keyword argument that contains the primary key. By default, pk_url_kwarg is 'pk'.

context_object_name - Designates the name of the variable to use in the context.

query_pk_and_slug - If True, causes get_object() to perform its lookup using both the primary key and the slug. Defaults to False.

Methods:-

get_object(queryset=None) - Returns the single object that this view will display. If queryset is provided, that queryset will be used as the source of objects; otherwise, get_queryset() will be used. get_object() looks for a pk_url_kwarg argument in the arguments to the view; if this argument is found, this method performs a primary-key based lookup using that value. If this argument is not found, it looks for a slug_url_kwarg argument, and performs a slug lookup using the slug_field.

When query_pk_and_slug is True, get_object() will perform its lookup using both the primary key and the slug.

get_queryset() - Returns the queryset that will be used to retrieve the object that this view will display. By default, get_queryset() returns the value of the queryset attribute if it is set, otherwise it constructs a QuerySet by calling the all() method on the model attribute's default manager.

get_slug_field() - Returns the name of a slug field to be used to look up by slug. By default this returns the value of slug_field.

get_context_object_name(obj) - Return the context variable name that will be used to contain the data that this view is manipulating. If context_object_name is not set, the context name will be constructed from the model_name of the model that the queryset is composed from. For example, the model Article would have context object named 'article'.

get_context_data(kwargs)** - Returns context data for displaying the object.

The base implementation of this method requires that the self.object attribute be set by the view (even if None). Be sure to do this if you are using this mixin without one of the built-in views that does so.

It returns a dictionary with these contents:

object: The object that this view is displaying (self.object).

Example 1:-

views.py

```
from django.views.generic.detail import DetailView

from .models import Student

class Student DetailView(DetailView):
        model = Student
```

urls.py

```
urlpatterns = [
       path('student/<int:pk>', views.Student DetailView.as_view(), name='student'),
]
```

Default:- AppName/ModelClassName_detail.html
Ex:- school/student_detail.html

Default Context:- ModelClassName
Ex:- student

Example 2:-

```
from django.views.generic.detail import DetailView

from .models import Student

class Student DetailView(DetailView):
        model = Student
        template_name = 'school/student.html'
        context_object_name = 'student'
```

Custom Template Name

Custom Context Name

urls.py

```
urlpatterns = [
       path('student/<int:pk>', views.Student DetailView.as_view(), name='student'),
]
```