Web technology reference for developers
Structure of content on the web
Code used to describe document style
General-purpose scripting language
Protocol for transmitting web resources
Interfaces for building web applications
Developing extensions for web browsers
Web technology reference for developers
Learn web development
Learn web development
Learn to structure web content with HTML
Learn to style content using CSS
Learn to run scripts in the browser
Learn to make the web accessible to all
A customized MDN experience
Learn how to use MDN Plus
Frequently asked questions about MDN Plus
This article shows how to define models for the LocalLibrary website. It explains what a model is, how it is declared, and some of the main field types. It also briefly shows a few of the main ways you can access model data.
To be able to design and create your own models, choosing fields appropriately.
Django web applications access and manage data through Python objects referred to as models. Models define the
  
 structure
  
 of stored data, including the field
  
 types
  
 and possibly also their maximum size, default values, selection list options, help text for documentation, label text for forms, etc. The definition of the model is independent of the underlying database — you can choose one of several as part of your project settings. Once you've chosen what database you want to use, you don't need to talk to it directly at all — you just write your model structure and other code, and Django handles all the dirty work of communicating with the database for you.
This tutorial shows how to define and access the models for the
  
 LocalLibrary website
  
 example.
Before you jump in and start coding the models, it's worth taking a few minutes to think about what data we need to store and the relationships between the different objects.
We know that we need to store information about books (title, summary, author, written language, category, ISBN) and that we might have multiple copies available (with globally unique id, availability status, etc.). We might need to store more information about the author than just their name, and there might be multiple authors with the same or similar names. We want to be able to sort information based on book title, author, written language, and category.
When designing your models, it makes sense to have separate models for every "object" (a group of related information). In this case, the obvious objects are books, book instances, and authors.
You might also want to use models to represent selection-list options (e.g. like a drop down list of choices), rather than hard coding the choices into the website itself — this is recommended when all the options aren't known up front or may change. Obvious candidates for models, in this case, include the book genre (e.g. Science Fiction, French Poetry, etc.) and language (English, French, Japanese).
Once we've decided on our models and field, we need to think about the relationships. Django allows you to define relationships that are one to one (
  
 OneToOneField
  
 ), one to many (
  
 ForeignKey
  
 ) and many to many (
  
 ManyToManyField
  
 ).
With that in mind, the UML association diagram below shows the models we'll define in this case (as boxes).
  
  
  
 We've created models for the book (the generic details of the book), book instance (status of specific physical copies of the book available in the system), and author. We have also decided to have a model for the genre so that values can be created/selected through the admin interface. We've decided not to have a model for the
  
 BookInstance:status
  
 — we've hardcoded the values (
  
 LOAN\_STATUS
  
 ) because we don't expect these to change. Within each of the boxes, you can see the model name, the field names, and types, and also the methods and their return types.
The diagram also shows the relationships between the models, including their
  
 multiplicities
  
 . The multiplicities are the numbers on the diagram showing the numbers (maximum and minimum) of each model that may be present in the relationship. For example, the connecting line between the boxes shows that Book and a Genre are related. The numbers close to the Genre model show that a book must have one or more Genres (as many as you like), while the numbers on the other end of the line next to the Book model show that a Genre can have zero or many associated books.
  
  
 Note:
  
 The next section provides a basic primer explaining how models are defined and used. As you read it, consider how we will construct each of the models in the diagram above.
This section provides a brief overview of how a model is defined and some of the more important fields and field arguments.
Models are usually defined in an application's
  
 models.py
  
 file. They are implemented as subclasses of
  
 django.db.models.Model
  
 , and can include fields, methods and metadata. The code fragment below shows a "typical" model, named
  
 MyModelName
  
 :
In the below sections we'll explore each of the features inside the model in detail:
A model can have an arbitrary number of fields, of any type — each one represents a column of data that we want to store in one of our database tables. Each database record (row) will consist of one of each field value. Let's look at the example seen below:
Our above example has a single field called
  
 my\_field\_name
  
 , of type
  
 models.CharField
  
 — which means that this field will contain strings of alphanumeric characters. The field types are assigned using specific classes, which determine the type of record that is used to store the data in the database, along with validation criteria to be used when values are received from an HTML form (i.e. what constitutes a valid value). The field types can also take arguments that further specify how the field is stored or can be used. In this case we are giving our field two arguments:
The field name is used to refer to it in queries and templates.
Fields also have a label, which is specified using the
  
 verbose\_name
  
 argument (with a default value of
  
 None
  
 ).
If
  
 verbose\_name
  
 is not set, the label is created from the field name by replacing any underscores with a space, and capitalizing the first letter (for example, the field
  
 my\_field\_name
  
 would have a default label of
  
 My field name
  
 when used in forms).
The order that fields are declared will affect their default order if a model is rendered in a form (e.g. in the Admin site), though this may be overridden.
The following common arguments can be used when declaring many/most of the different field types:
  
  
 Note:
  
 Apps created using
  
 manage.py
  
 set the type of the primary key to a
  
 BigAutoField
  
 .
You can see this in the local library
  
 catalog/apps.py
  
 file:
There are many other options — you can view the
  
 full list of field options here
  
 .
The following list describes some of the more commonly used types of fields.
There are many other types of fields, including fields for different types of numbers (big integers, small integers, floats), booleans, URLs, slugs, unique ids, and other "time-related" information (duration, time, etc.). You can view the
  
 full list here
  
 .
You can declare model-level metadata for your Model by declaring
  
 class Meta
  
 , as shown.
One of the most useful features of this metadata is to control the
  
 default ordering
  
 of records returned when you query the model type. You do this by specifying the match order in a list of field names to the
  
 ordering
  
 attribute, as shown above. The ordering will depend on the type of field (character fields are sorted alphabetically, while date fields are sorted in chronological order). As shown above, you can prefix the field name with a minus symbol (-) to reverse the sorting order.
So as an example, if we chose to sort books like this by default:
the books would be sorted alphabetically by title, from A-Z, and then by publication date inside each title, from newest to oldest.
Another common attribute is
  
 verbose\_name
  
 , a verbose name for the class in singular and plural form:
Other useful attributes allow you to create and apply new "access permissions" for the model (default permissions are applied automatically), allow ordering based on another field, or to declare that the class is "abstract" (a base class that you cannot create records for, and will instead be derived from to create other models).
Many of the other metadata options control what database must be used for the model and how the data is stored (these are really only useful if you need to map a model to an existing database).
The full list of metadata options are available here:
  
 Model metadata options
  
 (Django docs).
A model can also have methods.
  
  
 Minimally, in every model you should define the standard Python class method
  
 \_\_str\_\_()
  
 to return a human-readable string for each object.
  
 This string is used to represent individual records in the administration site (and anywhere else you need to refer to a model instance). Often this will return a title or name field from the model.
Another common method to include in Django models is
  
 get\_absolute\_url()
  
 , which returns a URL for displaying individual model records on the website (if you define this method then Django will automatically add a "View on Site" button to the model's record editing screens in the Admin site). A typical pattern for
  
 get\_absolute\_url()
  
 is shown below.
  
  
 Note:
  
 Assuming you will use URLs like
  
 /myapplication/mymodelname/2
  
 to display individual records for your model (where "2" is the
  
 id
  
 for a particular record), you will need to create a URL mapper to pass the response and id to a "model detail view" (which will do the work required to display the record). The
  
 reverse()
  
 function above is able to "reverse" your URL mapper (in the above case named
  
 'model-detail-view'
  
 ) in order to create a URL of the right format.
Of course to make this work you still have to write the URL mapping, view, and template!
You can also define any other methods you like, and call them from your code or templates (provided that they don't take any parameters).
Once you've defined your model classes you can use them to create, update, or delete records, and to run queries to get all records or particular subsets of records. We'll show you how to do that in the tutorial when we define our views, but here is a brief summary.
To create a record you can define an instance of the model and then call
  
 save()
  
 .
  
  
 Note:
  
 If you haven't declared any field as a
  
 primary\_key
  
 , the new record will be given one automatically, with the field name
  
 id
  
 . You could query this field after saving the above record, and it would have a value of 1.
You can access the fields in this new record using the dot syntax, and change the values. You have to call
  
 save()
  
 to store modified values to the database.
You can search for records that match certain criteria using the model's
  
 objects
  
 attribute (provided by the base class).
  
  
 Note:
  
 Explaining how to search for records using "abstract" model and field names can be a little confusing. In the discussion below, we'll refer to a
  
 Book
  
 model with
  
 title
  
 and
  
 genre
  
 fields, where genre is also a model with a single field
  
 name
  
 .
We can get all records for a model as a
  
 QuerySet
  
 , using
  
 objects.all()
  
 . The
  
 QuerySet
  
 is an iterable object, meaning that it contains a number of objects that we can iterate/loop through.
Django's
  
 filter()
  
 method allows us to filter the returned
  
 QuerySet
  
 to match a specified
  
 text
  
 or
  
 numeric
  
 field against particular criteria. For example, to filter for books that contain "wild" in the title and then count them, we could do the following.
The fields to match and the type of match are defined in the filter parameter name, using the format:
  
 field\_name\_\_match\_type
  
 (note the
  
 double underscore
  
 between
  
 title
  
 and
  
 contains
  
 above). Above we're filtering
  
 title
  
 with a case-sensitive match. There are many other types of matches you can do:
  
 icontains
  
 (case insensitive),
  
 iexact
  
 (case-insensitive exact match),
  
 exact
  
 (case-sensitive exact match) and
  
 in
  
 ,
  
 gt
  
 (greater than),
  
 startswith
  
 , etc. The
  
 full list is here
  
 .
In some cases, you'll need to filter on a field that defines a one-to-many relationship to another model (e.g. a
  
 ForeignKey
  
 ). In this case, you can "index" to fields within the related model with additional double underscores.
So for example to filter for books with a specific genre pattern, you will have to index to the
  
 name
  
 through the
  
 genre
  
 field, as shown below:
  
  
 Note:
  
 You can use underscores (
  
 \_\_
  
 ) to navigate as many levels of relationships (
  
 ForeignKey
  
 /
  
 ManyToManyField
  
 ) as you like.
For example, a
  
 Book
  
 that had different types, defined using a further "cover" relationship might have a parameter name:
  
 type\_\_cover\_\_name\_\_exact='hard'.
  
  
 There is a lot more you can do with queries, including backwards searches from related models, chaining filters, returning a smaller set of values, etc. For more information, see
  
 Making queries
  
 (Django Docs).
In this section we will start defining the models for the library. Open
  
 models.py
  
 (in /locallibrary/catalog/). The boilerplate at the top of the page imports the
  
 models
  
 module, which contains the model base class
  
 models.Model
  
 that our models will inherit from.
Copy the
  
 Genre
  
 model code shown below and paste it into the bottom of your
  
 models.py
  
 file. This model is used to store information about the book category — for example whether it is fiction or non-fiction, romance or military history, etc.
As mentioned above, we've created the genre as a model rather than as free text or a selection list so that the possible values can be managed through the database rather than being hard coded.
The model has a single
  
 CharField
  
 field (
  
 name
  
 ), which is used to describe the genre (this is limited to 200 characters and has some
  
 help\_text
  
 ). At the end of the model, we declare a
  
 \_\_str\_\_()
  
 method, which returns the name of the genre defined by a particular record. No verbose name has been defined, so the field will be called
  
 Name
  
 in forms.
Copy the
  
 Book
  
 model below and again paste it into the bottom of your file. The
  
 Book
  
 model represents all information about an available book in a general sense, but not a particular physical "instance" or "copy" available for loan. The model uses a
  
 CharField
  
 to represent the book's
  
 title
  
 and
  
 isbn
  
 . For
  
 isbn
  
 , note how the first unnamed parameter explicitly sets the label as "ISBN" (otherwise, it would default to "Isbn"). We also set the parameter
  
 unique
  
 as
  
 true
  
 to ensure all books have a unique ISBN (the unique parameter makes the field value globally unique in a table). The model uses
  
 TextField
  
 for the
  
 summary
  
 , because this text may need to be quite long.
The genre is a
  
 ManyToManyField
  
 , so that a book can have multiple genres and a genre can have many books. The author is declared as
  
 ForeignKey
  
 , so each book will only have one author, but an author may have many books (in practice a book might have multiple authors, but not in this implementation!)
In both field types the related model class is declared as the first unnamed parameter using either the model class or a string containing the name of the related model. You must use the name of the model as a string if the associated class has not yet been defined in this file before it is referenced! The other parameters of interest in the
  
 author
  
 field are
  
 null=True
  
 , which allows the database to store a
  
 Null
  
 value if no author is selected, and
  
 on\_delete=models.SET\_NULL
  
 , which will set the value of the book's author field to
  
 Null
  
 if the associated author record is deleted.
  
  
 Warning:
  
 By default
  
 on\_delete=models.CASCADE
  
 , which means that if the author was deleted, this book would be deleted too! We use
  
 SET\_NULL
  
 here, but we could also use
  
 PROTECT
  
 or
  
 RESTRICT
  
 to prevent the author being deleted while any book uses it.
The model also defines
  
 \_\_str\_\_()
  
 , using the book's
  
 title
  
 field to represent a
  
 Book
  
 record. The final method,
  
 get\_absolute\_url()
  
 returns a URL that can be used to access a detail record for this model (for this to work, we will have to define a URL mapping that has the name
  
 book-detail
  
 , and define an associated view and template).
Next, copy the
  
 BookInstance
  
 model (shown below) under the other models. The
  
 BookInstance
  
 represents a specific copy of a book that someone might borrow, and includes information about whether the copy is available or on what date it is expected back, "imprint" or version details, and a unique id for the book in the library.
Some of the fields and methods will now be familiar. The model uses:
We additionally declare a few new types of field:
The method
  
 \_\_str\_\_()
  
 represents the
  
 BookInstance
  
 object using a combination of its unique id and the associated
  
 Book
  
 's title.
  
  
 Note:
  
 A little Python:
Copy the
  
 Author
  
 model (shown below) underneath the existing code in
  
 models.py
  
 .
All of the fields/methods should now be familiar. The model defines an author as having a first name, last name, and dates of birth and death (both optional). It specifies that by default the
  
 \_\_str\_\_()
  
 returns the name in
  
 last name
  
 ,
  
 firstname
  
 order. The
  
 get\_absolute\_url()
  
 method reverses the
  
 author-detail
  
 URL mapping to get the URL for displaying an individual author.
All your models have now been created. Now re-run your database migrations to add them to your database.
Imagine a local benefactor donates a number of new books written in another language (say, Farsi). The challenge is to work out how these would be best represented in our library website, and then to add them to the models.
Some things to consider:
After you've decided, add the field. You can see what we decided on GitHub
  
 here
  
 .
Don't forget that after a change to your model, you should again re-run your database migrations to add the changes.
In this article we've learned how models are defined, and then used this information to design and implement appropriate models for the
  
 LocalLibrary
  
 website.
At this point we'll divert briefly from creating the site, and check out the
  
 Django Administration site
  
 . This site will allow us to add some data to the library, which we can then display using our (yet to be created) views and templates.
  
  
 Last modified:
  
  
  
 Aug 17, 2022
  
 ,
  
  
  
 by MDN contributors
  
  
 Your blueprint for a better internet.
Visit
  
  
  
 Mozilla Corporation’s
  
  
  
 not-for-profit parent, the
  
  
  
 Mozilla Foundation
  
 .
  
 Portions of this content are ©1998–
  
 2022
  
 by individual mozilla.org contributors. Content available under
  
  
  
 a Creative Commons license
  
 .
Web technology reference for developers
Structure of content on the web
Code used to describe document style
General-purpose scripting language
Protocol for transmitting web resources
Interfaces for building web applications
Developing extensions for web browsers
Web technology reference for developers
Learn web development
Learn web development
Learn to structure web content with HTML
Learn to style content using CSS
Learn to run scripts in the browser
Learn to make the web accessible to all
A customized MDN experience
Learn how to use MDN Plus
Frequently asked questions about MDN Plus
This article shows how to define models for the LocalLibrary website. It explains what a model is, how it is declared, and some of the main field types. It also briefly shows a few of the main ways you can access model data.
To be able to design and create your own models, choosing fields appropriately.
Django web applications access and manage data through Python objects referred to as models. Models define the
  
 structure
  
 of stored data, including the field
  
 types
  
 and possibly also their maximum size, default values, selection list options, help text for documentation, label text for forms, etc. The definition of the model is independent of the underlying database — you can choose one of several as part of your project settings. Once you've chosen what database you want to use, you don't need to talk to it directly at all — you just write your model structure and other code, and Django handles all the dirty work of communicating with the database for you.
This tutorial shows how to define and access the models for the
  
 LocalLibrary website
  
 example.
Before you jump in and start coding the models, it's worth taking a few minutes to think about what data we need to store and the relationships between the different objects.
We know that we need to store information about books (title, summary, author, written language, category, ISBN) and that we might have multiple copies available (with globally unique id, availability status, etc.). We might need to store more information about the author than just their name, and there might be multiple authors with the same or similar names. We want to be able to sort information based on book title, author, written language, and category.
When designing your models, it makes sense to have separate models for every "object" (a group of related information). In this case, the obvious objects are books, book instances, and authors.
You might also want to use models to represent selection-list options (e.g. like a drop down list of choices), rather than hard coding the choices into the website itself — this is recommended when all the options aren't known up front or may change. Obvious candidates for models, in this case, include the book genre (e.g. Science Fiction, French Poetry, etc.) and language (English, French, Japanese).
Once we've decided on our models and field, we need to think about the relationships. Django allows you to define relationships that are one to one (
  
 OneToOneField
  
 ), one to many (
  
 ForeignKey
  
 ) and many to many (
  
 ManyToManyField
  
 ).
With that in mind, the UML association diagram below shows the models we'll define in this case (as boxes).
  
  
  
 We've created models for the book (the generic details of the book), book instance (status of specific physical copies of the book available in the system), and author. We have also decided to have a model for the genre so that values can be created/selected through the admin interface. We've decided not to have a model for the
  
 BookInstance:status
  
 — we've hardcoded the values (
  
 LOAN\_STATUS
  
 ) because we don't expect these to change. Within each of the boxes, you can see the model name, the field names, and types, and also the methods and their return types.
The diagram also shows the relationships between the models, including their
  
 multiplicities
  
 . The multiplicities are the numbers on the diagram showing the numbers (maximum and minimum) of each model that may be present in the relationship. For example, the connecting line between the boxes shows that Book and a Genre are related. The numbers close to the Genre model show that a book must have one or more Genres (as many as you like), while the numbers on the other end of the line next to the Book model show that a Genre can have zero or many associated books.
  
  
 Note:
  
 The next section provides a basic primer explaining how models are defined and used. As you read it, consider how we will construct each of the models in the diagram above.
This section provides a brief overview of how a model is defined and some of the more important fields and field arguments.
Models are usually defined in an application's
  
 models.py
  
 file. They are implemented as subclasses of
  
 django.db.models.Model
  
 , and can include fields, methods and metadata. The code fragment below shows a "typical" model, named
  
 MyModelName
  
 :
In the below sections we'll explore each of the features inside the model in detail:
A model can have an arbitrary number of fields, of any type — each one represents a column of data that we want to store in one of our database tables. Each database record (row) will consist of one of each field value. Let's look at the example seen below:
Our above example has a single field called
  
 my\_field\_name
  
 , of type
  
 models.CharField
  
 — which means that this field will contain strings of alphanumeric characters. The field types are assigned using specific classes, which determine the type of record that is used to store the data in the database, along with validation criteria to be used when values are received from an HTML form (i.e. what constitutes a valid value). The field types can also take arguments that further specify how the field is stored or can be used. In this case we are giving our field two arguments:
The field name is used to refer to it in queries and templates.
Fields also have a label, which is specified using the
  
 verbose\_name
  
 argument (with a default value of
  
 None
  
 ).
If
  
 verbose\_name
  
 is not set, the label is created from the field name by replacing any underscores with a space, and capitalizing the first letter (for example, the field
  
 my\_field\_name
  
 would have a default label of
  
 My field name
  
 when used in forms).
The order that fields are declared will affect their default order if a model is rendered in a form (e.g. in the Admin site), though this may be overridden.
The following common arguments can be used when declaring many/most of the different field types:
  
  
 Note:
  
 Apps created using
  
 manage.py
  
 set the type of the primary key to a
  
 BigAutoField
  
 .
You can see this in the local library
  
 catalog/apps.py
  
 file:
There are many other options — you can view the
  
 full list of field options here
  
 .
The following list describes some of the more commonly used types of fields.
There are many other types of fields, including fields for different types of numbers (big integers, small integers, floats), booleans, URLs, slugs, unique ids, and other "time-related" information (duration, time, etc.). You can view the
  
 full list here
  
 .
You can declare model-level metadata for your Model by declaring
  
 class Meta
  
 , as shown.
One of the most useful features of this metadata is to control the
  
 default ordering
  
 of records returned when you query the model type. You do this by specifying the match order in a list of field names to the
  
 ordering
  
 attribute, as shown above. The ordering will depend on the type of field (character fields are sorted alphabetically, while date fields are sorted in chronological order). As shown above, you can prefix the field name with a minus symbol (-) to reverse the sorting order.
So as an example, if we chose to sort books like this by default:
the books would be sorted alphabetically by title, from A-Z, and then by publication date inside each title, from newest to oldest.
Another common attribute is
  
 verbose\_name
  
 , a verbose name for the class in singular and plural form:
Other useful attributes allow you to create and apply new "access permissions" for the model (default permissions are applied automatically), allow ordering based on another field, or to declare that the class is "abstract" (a base class that you cannot create records for, and will instead be derived from to create other models).
Many of the other metadata options control what database must be used for the model and how the data is stored (these are really only useful if you need to map a model to an existing database).
The full list of metadata options are available here:
  
 Model metadata options
  
 (Django docs).
A model can also have methods.
  
  
 Minimally, in every model you should define the standard Python class method
  
 \_\_str\_\_()
  
 to return a human-readable string for each object.
  
 This string is used to represent individual records in the administration site (and anywhere else you need to refer to a model instance). Often this will return a title or name field from the model.
Another common method to include in Django models is
  
 get\_absolute\_url()
  
 , which returns a URL for displaying individual model records on the website (if you define this method then Django will automatically add a "View on Site" button to the model's record editing screens in the Admin site). A typical pattern for
  
 get\_absolute\_url()
  
 is shown below.
  
  
 Note:
  
 Assuming you will use URLs like
  
 /myapplication/mymodelname/2
  
 to display individual records for your model (where "2" is the
  
 id
  
 for a particular record), you will need to create a URL mapper to pass the response and id to a "model detail view" (which will do the work required to display the record). The
  
 reverse()
  
 function above is able to "reverse" your URL mapper (in the above case named
  
 'model-detail-view'
  
 ) in order to create a URL of the right format.
Of course to make this work you still have to write the URL mapping, view, and template!
You can also define any other methods you like, and call them from your code or templates (provided that they don't take any parameters).
Once you've defined your model classes you can use them to create, update, or delete records, and to run queries to get all records or particular subsets of records. We'll show you how to do that in the tutorial when we define our views, but here is a brief summary.
To create a record you can define an instance of the model and then call
  
 save()
  
 .
  
  
 Note:
  
 If you haven't declared any field as a
  
 primary\_key
  
 , the new record will be given one automatically, with the field name
  
 id
  
 . You could query this field after saving the above record, and it would have a value of 1.
You can access the fields in this new record using the dot syntax, and change the values. You have to call
  
 save()
  
 to store modified values to the database.
You can search for records that match certain criteria using the model's
  
 objects
  
 attribute (provided by the base class).
  
  
 Note:
  
 Explaining how to search for records using "abstract" model and field names can be a little confusing. In the discussion below, we'll refer to a
  
 Book
  
 model with
  
 title
  
 and
  
 genre
  
 fields, where genre is also a model with a single field
  
 name
  
 .
We can get all records for a model as a
  
 QuerySet
  
 , using
  
 objects.all()
  
 . The
  
 QuerySet
  
 is an iterable object, meaning that it contains a number of objects that we can iterate/loop through.
Django's
  
 filter()
  
 method allows us to filter the returned
  
 QuerySet
  
 to match a specified
  
 text
  
 or
  
 numeric
  
 field against particular criteria. For example, to filter for books that contain "wild" in the title and then count them, we could do the following.
The fields to match and the type of match are defined in the filter parameter name, using the format:
  
 field\_name\_\_match\_type
  
 (note the
  
 double underscore
  
 between
  
 title
  
 and
  
 contains
  
 above). Above we're filtering
  
 title
  
 with a case-sensitive match. There are many other types of matches you can do:
  
 icontains
  
 (case insensitive),
  
 iexact
  
 (case-insensitive exact match),
  
 exact
  
 (case-sensitive exact match) and
  
 in
  
 ,
  
 gt
  
 (greater than),
  
 startswith
  
 , etc. The
  
 full list is here
  
 .
In some cases, you'll need to filter on a field that defines a one-to-many relationship to another model (e.g. a
  
 ForeignKey
  
 ). In this case, you can "index" to fields within the related model with additional double underscores.
So for example to filter for books with a specific genre pattern, you will have to index to the
  
 name
  
 through the
  
 genre
  
 field, as shown below:
  
  
 Note:
  
 You can use underscores (
  
 \_\_
  
 ) to navigate as many levels of relationships (
  
 ForeignKey
  
 /
  
 ManyToManyField
  
 ) as you like.
For example, a
  
 Book
  
 that had different types, defined using a further "cover" relationship might have a parameter name:
  
 type\_\_cover\_\_name\_\_exact='hard'.
  
  
 There is a lot more you can do with queries, including backwards searches from related models, chaining filters, returning a smaller set of values, etc. For more information, see
  
 Making queries
  
 (Django Docs).
In this section we will start defining the models for the library. Open
  
 models.py
  
 (in /locallibrary/catalog/). The boilerplate at the top of the page imports the
  
 models
  
 module, which contains the model base class
  
 models.Model
  
 that our models will inherit from.
Copy the
  
 Genre
  
 model code shown below and paste it into the bottom of your
  
 models.py
  
 file. This model is used to store information about the book category — for example whether it is fiction or non-fiction, romance or military history, etc.
As mentioned above, we've created the genre as a model rather than as free text or a selection list so that the possible values can be managed through the database rather than being hard coded.
The model has a single
  
 CharField
  
 field (
  
 name
  
 ), which is used to describe the genre (this is limited to 200 characters and has some
  
 help\_text
  
 ). At the end of the model, we declare a
  
 \_\_str\_\_()
  
 method, which returns the name of the genre defined by a particular record. No verbose name has been defined, so the field will be called
  
 Name
  
 in forms.
Copy the
  
 Book
  
 model below and again paste it into the bottom of your file. The
  
 Book
  
 model represents all information about an available book in a general sense, but not a particular physical "instance" or "copy" available for loan. The model uses a
  
 CharField
  
 to represent the book's
  
 title
  
 and
  
 isbn
  
 . For
  
 isbn
  
 , note how the first unnamed parameter explicitly sets the label as "ISBN" (otherwise, it would default to "Isbn"). We also set the parameter
  
 unique
  
 as
  
 true
  
 to ensure all books have a unique ISBN (the unique parameter makes the field value globally unique in a table). The model uses
  
 TextField
  
 for the
  
 summary
  
 , because this text may need to be quite long.
The genre is a
  
 ManyToManyField
  
 , so that a book can have multiple genres and a genre can have many books. The author is declared as
  
 ForeignKey
  
 , so each book will only have one author, but an author may have many books (in practice a book might have multiple authors, but not in this implementation!)
In both field types the related model class is declared as the first unnamed parameter using either the model class or a string containing the name of the related model. You must use the name of the model as a string if the associated class has not yet been defined in this file before it is referenced! The other parameters of interest in the
  
 author
  
 field are
  
 null=True
  
 , which allows the database to store a
  
 Null
  
 value if no author is selected, and
  
 on\_delete=models.SET\_NULL
  
 , which will set the value of the book's author field to
  
 Null
  
 if the associated author record is deleted.
  
  
 Warning:
  
 By default
  
 on\_delete=models.CASCADE
  
 , which means that if the author was deleted, this book would be deleted too! We use
  
 SET\_NULL
  
 here, but we could also use
  
 PROTECT
  
 or
  
 RESTRICT
  
 to prevent the author being deleted while any book uses it.
The model also defines
  
 \_\_str\_\_()
  
 , using the book's
  
 title
  
 field to represent a
  
 Book
  
 record. The final method,
  
 get\_absolute\_url()
  
 returns a URL that can be used to access a detail record for this model (for this to work, we will have to define a URL mapping that has the name
  
 book-detail
  
 , and define an associated view and template).
Next, copy the
  
 BookInstance
  
 model (shown below) under the other models. The
  
 BookInstance
  
 represents a specific copy of a book that someone might borrow, and includes information about whether the copy is available or on what date it is expected back, "imprint" or version details, and a unique id for the book in the library.
Some of the fields and methods will now be familiar. The model uses:
We additionally declare a few new types of field:
The method
  
 \_\_str\_\_()
  
 represents the
  
 BookInstance
  
 object using a combination of its unique id and the associated
  
 Book
  
 's title.
  
  
 Note:
  
 A little Python:
Copy the
  
 Author
  
 model (shown below) underneath the existing code in
  
 models.py
  
 .
All of the fields/methods should now be familiar. The model defines an author as having a first name, last name, and dates of birth and death (both optional). It specifies that by default the
  
 \_\_str\_\_()
  
 returns the name in
  
 last name
  
 ,
  
 firstname
  
 order. The
  
 get\_absolute\_url()
  
 method reverses the
  
 author-detail
  
 URL mapping to get the URL for displaying an individual author.
All your models have now been created. Now re-run your database migrations to add them to your database.
Imagine a local benefactor donates a number of new books written in another language (say, Farsi). The challenge is to work out how these would be best represented in our library website, and then to add them to the models.
Some things to consider:
After you've decided, add the field. You can see what we decided on GitHub
  
 here
  
 .
Don't forget that after a change to your model, you should again re-run your database migrations to add the changes.
In this article we've learned how models are defined, and then used this information to design and implement appropriate models for the
  
 LocalLibrary
  
 website.
At this point we'll divert briefly from creating the site, and check out the
  
 Django Administration site
  
 . This site will allow us to add some data to the library, which we can then display using our (yet to be created) views and templates.
  
  
 Last modified:
  
  
  
 Aug 17, 2022
  
 ,
  
  
  
 by MDN contributors
  
  
 Web technology reference for developers
Structure of content on the web
Code used to describe document style
General-purpose scripting language
Protocol for transmitting web resources
Interfaces for building web applications
Developing extensions for web browsers
Web technology reference for developers
Learn web development
Learn web development
Learn to structure web content with HTML
Learn to style content using CSS
Learn to run scripts in the browser
Learn to make the web accessible to all
A customized MDN experience
Learn how to use MDN Plus
Frequently asked questions about MDN Plus
Web technology reference for developers
Structure of content on the web
Code used to describe document style
General-purpose scripting language
Protocol for transmitting web resources
Interfaces for building web applications
Developing extensions for web browsers
Web technology reference for developers
Learn web development
Learn web development
Learn to structure web content with HTML
Learn to style content using CSS
Learn to run scripts in the browser
Learn to make the web accessible to all
A customized MDN experience
Learn how to use MDN Plus
Frequently asked questions about MDN Plus
Web technology reference for developers
Structure of content on the web
Code used to describe document style
General-purpose scripting language
Protocol for transmitting web resources
Interfaces for building web applications
Developing extensions for web browsers
Web technology reference for developers
Learn web development
Learn web development
Learn to structure web content with HTML
Learn to style content using CSS
Learn to run scripts in the browser
Learn to make the web accessible to all
A customized MDN experience
Learn how to use MDN Plus
Frequently asked questions about MDN Plus
Web technology reference for developers
Structure of content on the web
Code used to describe document style
General-purpose scripting language
Protocol for transmitting web resources
Interfaces for building web applications
Developing extensions for web browsers
Web technology reference for developers
Learn web development
Learn web development
Learn to structure web content with HTML
Learn to style content using CSS
Learn to run scripts in the browser
Learn to make the web accessible to all
A customized MDN experience
Learn how to use MDN Plus
Frequently asked questions about MDN Plus
This article shows how to define models for the LocalLibrary website. It explains what a model is, how it is declared, and some of the main field types. It also briefly shows a few of the main ways you can access model data.
To be able to design and create your own models, choosing fields appropriately.
Django web applications access and manage data through Python objects referred to as models. Models define the
  
 structure
  
 of stored data, including the field
  
 types
  
 and possibly also their maximum size, default values, selection list options, help text for documentation, label text for forms, etc. The definition of the model is independent of the underlying database — you can choose one of several as part of your project settings. Once you've chosen what database you want to use, you don't need to talk to it directly at all — you just write your model structure and other code, and Django handles all the dirty work of communicating with the database for you.
This tutorial shows how to define and access the models for the
  
 LocalLibrary website
  
 example.
Before you jump in and start coding the models, it's worth taking a few minutes to think about what data we need to store and the relationships between the different objects.
We know that we need to store information about books (title, summary, author, written language, category, ISBN) and that we might have multiple copies available (with globally unique id, availability status, etc.). We might need to store more information about the author than just their name, and there might be multiple authors with the same or similar names. We want to be able to sort information based on book title, author, written language, and category.
When designing your models, it makes sense to have separate models for every "object" (a group of related information). In this case, the obvious objects are books, book instances, and authors.
You might also want to use models to represent selection-list options (e.g. like a drop down list of choices), rather than hard coding the choices into the website itself — this is recommended when all the options aren't known up front or may change. Obvious candidates for models, in this case, include the book genre (e.g. Science Fiction, French Poetry, etc.) and language (English, French, Japanese).
Once we've decided on our models and field, we need to think about the relationships. Django allows you to define relationships that are one to one (
  
 OneToOneField
  
 ), one to many (
  
 ForeignKey
  
 ) and many to many (
  
 ManyToManyField
  
 ).
With that in mind, the UML association diagram below shows the models we'll define in this case (as boxes).
  
  
  
 We've created models for the book (the generic details of the book), book instance (status of specific physical copies of the book available in the system), and author. We have also decided to have a model for the genre so that values can be created/selected through the admin interface. We've decided not to have a model for the
  
 BookInstance:status
  
 — we've hardcoded the values (
  
 LOAN\_STATUS
  
 ) because we don't expect these to change. Within each of the boxes, you can see the model name, the field names, and types, and also the methods and their return types.
The diagram also shows the relationships between the models, including their
  
 multiplicities
  
 . The multiplicities are the numbers on the diagram showing the numbers (maximum and minimum) of each model that may be present in the relationship. For example, the connecting line between the boxes shows that Book and a Genre are related. The numbers close to the Genre model show that a book must have one or more Genres (as many as you like), while the numbers on the other end of the line next to the Book model show that a Genre can have zero or many associated books.
  
  
 Note:
  
 The next section provides a basic primer explaining how models are defined and used. As you read it, consider how we will construct each of the models in the diagram above.
This section provides a brief overview of how a model is defined and some of the more important fields and field arguments.
Models are usually defined in an application's
  
 models.py
  
 file. They are implemented as subclasses of
  
 django.db.models.Model
  
 , and can include fields, methods and metadata. The code fragment below shows a "typical" model, named
  
 MyModelName
  
 :
In the below sections we'll explore each of the features inside the model in detail:
A model can have an arbitrary number of fields, of any type — each one represents a column of data that we want to store in one of our database tables. Each database record (row) will consist of one of each field value. Let's look at the example seen below:
Our above example has a single field called
  
 my\_field\_name
  
 , of type
  
 models.CharField
  
 — which means that this field will contain strings of alphanumeric characters. The field types are assigned using specific classes, which determine the type of record that is used to store the data in the database, along with validation criteria to be used when values are received from an HTML form (i.e. what constitutes a valid value). The field types can also take arguments that further specify how the field is stored or can be used. In this case we are giving our field two arguments:
The field name is used to refer to it in queries and templates.
Fields also have a label, which is specified using the
  
 verbose\_name
  
 argument (with a default value of
  
 None
  
 ).
If
  
 verbose\_name
  
 is not set, the label is created from the field name by replacing any underscores with a space, and capitalizing the first letter (for example, the field
  
 my\_field\_name
  
 would have a default label of
  
 My field name
  
 when used in forms).
The order that fields are declared will affect their default order if a model is rendered in a form (e.g. in the Admin site), though this may be overridden.
The following common arguments can be used when declaring many/most of the different field types:
  
  
 Note:
  
 Apps created using
  
 manage.py
  
 set the type of the primary key to a
  
 BigAutoField
  
 .
You can see this in the local library
  
 catalog/apps.py
  
 file:
There are many other options — you can view the
  
 full list of field options here
  
 .
The following list describes some of the more commonly used types of fields.
There are many other types of fields, including fields for different types of numbers (big integers, small integers, floats), booleans, URLs, slugs, unique ids, and other "time-related" information (duration, time, etc.). You can view the
  
 full list here
  
 .
You can declare model-level metadata for your Model by declaring
  
 class Meta
  
 , as shown.
One of the most useful features of this metadata is to control the
  
 default ordering
  
 of records returned when you query the model type. You do this by specifying the match order in a list of field names to the
  
 ordering
  
 attribute, as shown above. The ordering will depend on the type of field (character fields are sorted alphabetically, while date fields are sorted in chronological order). As shown above, you can prefix the field name with a minus symbol (-) to reverse the sorting order.
So as an example, if we chose to sort books like this by default:
the books would be sorted alphabetically by title, from A-Z, and then by publication date inside each title, from newest to oldest.
Another common attribute is
  
 verbose\_name
  
 , a verbose name for the class in singular and plural form:
Other useful attributes allow you to create and apply new "access permissions" for the model (default permissions are applied automatically), allow ordering based on another field, or to declare that the class is "abstract" (a base class that you cannot create records for, and will instead be derived from to create other models).
Many of the other metadata options control what database must be used for the model and how the data is stored (these are really only useful if you need to map a model to an existing database).
The full list of metadata options are available here:
  
 Model metadata options
  
 (Django docs).
A model can also have methods.
  
  
 Minimally, in every model you should define the standard Python class method
  
 \_\_str\_\_()
  
 to return a human-readable string for each object.
  
 This string is used to represent individual records in the administration site (and anywhere else you need to refer to a model instance). Often this will return a title or name field from the model.
Another common method to include in Django models is
  
 get\_absolute\_url()
  
 , which returns a URL for displaying individual model records on the website (if you define this method then Django will automatically add a "View on Site" button to the model's record editing screens in the Admin site). A typical pattern for
  
 get\_absolute\_url()
  
 is shown below.
  
  
 Note:
  
 Assuming you will use URLs like
  
 /myapplication/mymodelname/2
  
 to display individual records for your model (where "2" is the
  
 id
  
 for a particular record), you will need to create a URL mapper to pass the response and id to a "model detail view" (which will do the work required to display the record). The
  
 reverse()
  
 function above is able to "reverse" your URL mapper (in the above case named
  
 'model-detail-view'
  
 ) in order to create a URL of the right format.
Of course to make this work you still have to write the URL mapping, view, and template!
You can also define any other methods you like, and call them from your code or templates (provided that they don't take any parameters).
Once you've defined your model classes you can use them to create, update, or delete records, and to run queries to get all records or particular subsets of records. We'll show you how to do that in the tutorial when we define our views, but here is a brief summary.
To create a record you can define an instance of the model and then call
  
 save()
  
 .
  
  
 Note:
  
 If you haven't declared any field as a
  
 primary\_key
  
 , the new record will be given one automatically, with the field name
  
 id
  
 . You could query this field after saving the above record, and it would have a value of 1.
You can access the fields in this new record using the dot syntax, and change the values. You have to call
  
 save()
  
 to store modified values to the database.
You can search for records that match certain criteria using the model's
  
 objects
  
 attribute (provided by the base class).
  
  
 Note:
  
 Explaining how to search for records using "abstract" model and field names can be a little confusing. In the discussion below, we'll refer to a
  
 Book
  
 model with
  
 title
  
 and
  
 genre
  
 fields, where genre is also a model with a single field
  
 name
  
 .
We can get all records for a model as a
  
 QuerySet
  
 , using
  
 objects.all()
  
 . The
  
 QuerySet
  
 is an iterable object, meaning that it contains a number of objects that we can iterate/loop through.
Django's
  
 filter()
  
 method allows us to filter the returned
  
 QuerySet
  
 to match a specified
  
 text
  
 or
  
 numeric
  
 field against particular criteria. For example, to filter for books that contain "wild" in the title and then count them, we could do the following.
The fields to match and the type of match are defined in the filter parameter name, using the format:
  
 field\_name\_\_match\_type
  
 (note the
  
 double underscore
  
 between
  
 title
  
 and
  
 contains
  
 above). Above we're filtering
  
 title
  
 with a case-sensitive match. There are many other types of matches you can do:
  
 icontains
  
 (case insensitive),
  
 iexact
  
 (case-insensitive exact match),
  
 exact
  
 (case-sensitive exact match) and
  
 in
  
 ,
  
 gt
  
 (greater than),
  
 startswith
  
 , etc. The
  
 full list is here
  
 .
In some cases, you'll need to filter on a field that defines a one-to-many relationship to another model (e.g. a
  
 ForeignKey
  
 ). In this case, you can "index" to fields within the related model with additional double underscores.
So for example to filter for books with a specific genre pattern, you will have to index to the
  
 name
  
 through the
  
 genre
  
 field, as shown below:
  
  
 Note:
  
 You can use underscores (
  
 \_\_
  
 ) to navigate as many levels of relationships (
  
 ForeignKey
  
 /
  
 ManyToManyField
  
 ) as you like.
For example, a
  
 Book
  
 that had different types, defined using a further "cover" relationship might have a parameter name:
  
 type\_\_cover\_\_name\_\_exact='hard'.
  
  
 There is a lot more you can do with queries, including backwards searches from related models, chaining filters, returning a smaller set of values, etc. For more information, see
  
 Making queries
  
 (Django Docs).
In this section we will start defining the models for the library. Open
  
 models.py
  
 (in /locallibrary/catalog/). The boilerplate at the top of the page imports the
  
 models
  
 module, which contains the model base class
  
 models.Model
  
 that our models will inherit from.
Copy the
  
 Genre
  
 model code shown below and paste it into the bottom of your
  
 models.py
  
 file. This model is used to store information about the book category — for example whether it is fiction or non-fiction, romance or military history, etc.
As mentioned above, we've created the genre as a model rather than as free text or a selection list so that the possible values can be managed through the database rather than being hard coded.
The model has a single
  
 CharField
  
 field (
  
 name
  
 ), which is used to describe the genre (this is limited to 200 characters and has some
  
 help\_text
  
 ). At the end of the model, we declare a
  
 \_\_str\_\_()
  
 method, which returns the name of the genre defined by a particular record. No verbose name has been defined, so the field will be called
  
 Name
  
 in forms.
Copy the
  
 Book
  
 model below and again paste it into the bottom of your file. The
  
 Book
  
 model represents all information about an available book in a general sense, but not a particular physical "instance" or "copy" available for loan. The model uses a
  
 CharField
  
 to represent the book's
  
 title
  
 and
  
 isbn
  
 . For
  
 isbn
  
 , note how the first unnamed parameter explicitly sets the label as "ISBN" (otherwise, it would default to "Isbn"). We also set the parameter
  
 unique
  
 as
  
 true
  
 to ensure all books have a unique ISBN (the unique parameter makes the field value globally unique in a table). The model uses
  
 TextField
  
 for the
  
 summary
  
 , because this text may need to be quite long.
The genre is a
  
 ManyToManyField
  
 , so that a book can have multiple genres and a genre can have many books. The author is declared as
  
 ForeignKey
  
 , so each book will only have one author, but an author may have many books (in practice a book might have multiple authors, but not in this implementation!)
In both field types the related model class is declared as the first unnamed parameter using either the model class or a string containing the name of the related model. You must use the name of the model as a string if the associated class has not yet been defined in this file before it is referenced! The other parameters of interest in the
  
 author
  
 field are
  
 null=True
  
 , which allows the database to store a
  
 Null
  
 value if no author is selected, and
  
 on\_delete=models.SET\_NULL
  
 , which will set the value of the book's author field to
  
 Null
  
 if the associated author record is deleted.
  
  
 Warning:
  
 By default
  
 on\_delete=models.CASCADE
  
 , which means that if the author was deleted, this book would be deleted too! We use
  
 SET\_NULL
  
 here, but we could also use
  
 PROTECT
  
 or
  
 RESTRICT
  
 to prevent the author being deleted while any book uses it.
The model also defines
  
 \_\_str\_\_()
  
 , using the book's
  
 title
  
 field to represent a
  
 Book
  
 record. The final method,
  
 get\_absolute\_url()
  
 returns a URL that can be used to access a detail record for this model (for this to work, we will have to define a URL mapping that has the name
  
 book-detail
  
 , and define an associated view and template).
Next, copy the
  
 BookInstance
  
 model (shown below) under the other models. The
  
 BookInstance
  
 represents a specific copy of a book that someone might borrow, and includes information about whether the copy is available or on what date it is expected back, "imprint" or version details, and a unique id for the book in the library.
Some of the fields and methods will now be familiar. The model uses:
We additionally declare a few new types of field:
The method
  
 \_\_str\_\_()
  
 represents the
  
 BookInstance
  
 object using a combination of its unique id and the associated
  
 Book
  
 's title.
  
  
 Note:
  
 A little Python:
Copy the
  
 Author
  
 model (shown below) underneath the existing code in
  
 models.py
  
 .
All of the fields/methods should now be familiar. The model defines an author as having a first name, last name, and dates of birth and death (both optional). It specifies that by default the
  
 \_\_str\_\_()
  
 returns the name in
  
 last name
  
 ,
  
 firstname
  
 order. The
  
 get\_absolute\_url()
  
 method reverses the
  
 author-detail
  
 URL mapping to get the URL for displaying an individual author.
All your models have now been created. Now re-run your database migrations to add them to your database.
Imagine a local benefactor donates a number of new books written in another language (say, Farsi). The challenge is to work out how these would be best represented in our library website, and then to add them to the models.
Some things to consider:
After you've decided, add the field. You can see what we decided on GitHub
  
 here
  
 .
Don't forget that after a change to your model, you should again re-run your database migrations to add the changes.
In this article we've learned how models are defined, and then used this information to design and implement appropriate models for the
  
 LocalLibrary
  
 website.
At this point we'll divert briefly from creating the site, and check out the
  
 Django Administration site
  
 . This site will allow us to add some data to the library, which we can then display using our (yet to be created) views and templates.
  
  
 Last modified:
  
  
  
 Aug 17, 2022
  
 ,
  
  
  
 by MDN contributors
  
  
 This article shows how to define models for the LocalLibrary website. It explains what a model is, how it is declared, and some of the main field types. It also briefly shows a few of the main ways you can access model data.
To be able to design and create your own models, choosing fields appropriately.
To be able to design and create your own models, choosing fields appropriately.
Django web applications access and manage data through Python objects referred to as models. Models define the
  
 structure
  
 of stored data, including the field
  
 types
  
 and possibly also their maximum size, default values, selection list options, help text for documentation, label text for forms, etc. The definition of the model is independent of the underlying database — you can choose one of several as part of your project settings. Once you've chosen what database you want to use, you don't need to talk to it directly at all — you just write your model structure and other code, and Django handles all the dirty work of communicating with the database for you.
This tutorial shows how to define and access the models for the
  
 LocalLibrary website
  
 example.
Before you jump in and start coding the models, it's worth taking a few minutes to think about what data we need to store and the relationships between the different objects.
We know that we need to store information about books (title, summary, author, written language, category, ISBN) and that we might have multiple copies available (with globally unique id, availability status, etc.). We might need to store more information about the author than just their name, and there might be multiple authors with the same or similar names. We want to be able to sort information based on book title, author, written language, and category.
When designing your models, it makes sense to have separate models for every "object" (a group of related information). In this case, the obvious objects are books, book instances, and authors.
You might also want to use models to represent selection-list options (e.g. like a drop down list of choices), rather than hard coding the choices into the website itself — this is recommended when all the options aren't known up front or may change. Obvious candidates for models, in this case, include the book genre (e.g. Science Fiction, French Poetry, etc.) and language (English, French, Japanese).
Once we've decided on our models and field, we need to think about the relationships. Django allows you to define relationships that are one to one (
  
 OneToOneField
  
 ), one to many (
  
 ForeignKey
  
 ) and many to many (
  
 ManyToManyField
  
 ).
With that in mind, the UML association diagram below shows the models we'll define in this case (as boxes).
  
  
  
 We've created models for the book (the generic details of the book), book instance (status of specific physical copies of the book available in the system), and author. We have also decided to have a model for the genre so that values can be created/selected through the admin interface. We've decided not to have a model for the
  
 BookInstance:status
  
 — we've hardcoded the values (
  
 LOAN\_STATUS
  
 ) because we don't expect these to change. Within each of the boxes, you can see the model name, the field names, and types, and also the methods and their return types.
The diagram also shows the relationships between the models, including their
  
 multiplicities
  
 . The multiplicities are the numbers on the diagram showing the numbers (maximum and minimum) of each model that may be present in the relationship. For example, the connecting line between the boxes shows that Book and a Genre are related. The numbers close to the Genre model show that a book must have one or more Genres (as many as you like), while the numbers on the other end of the line next to the Book model show that a Genre can have zero or many associated books.
  
  
 Note:
  
 The next section provides a basic primer explaining how models are defined and used. As you read it, consider how we will construct each of the models in the diagram above.
  
  
 Note:
  
 The next section provides a basic primer explaining how models are defined and used. As you read it, consider how we will construct each of the models in the diagram above.
This section provides a brief overview of how a model is defined and some of the more important fields and field arguments.
Models are usually defined in an application's
  
 models.py
  
 file. They are implemented as subclasses of
  
 django.db.models.Model
  
 , and can include fields, methods and metadata. The code fragment below shows a "typical" model, named
  
 MyModelName
  
 :
In the below sections we'll explore each of the features inside the model in detail:
A model can have an arbitrary number of fields, of any type — each one represents a column of data that we want to store in one of our database tables. Each database record (row) will consist of one of each field value. Let's look at the example seen below:
Our above example has a single field called
  
 my\_field\_name
  
 , of type
  
 models.CharField
  
 — which means that this field will contain strings of alphanumeric characters. The field types are assigned using specific classes, which determine the type of record that is used to store the data in the database, along with validation criteria to be used when values are received from an HTML form (i.e. what constitutes a valid value). The field types can also take arguments that further specify how the field is stored or can be used. In this case we are giving our field two arguments:
The field name is used to refer to it in queries and templates.
Fields also have a label, which is specified using the
  
 verbose\_name
  
 argument (with a default value of
  
 None
  
 ).
If
  
 verbose\_name
  
 is not set, the label is created from the field name by replacing any underscores with a space, and capitalizing the first letter (for example, the field
  
 my\_field\_name
  
 would have a default label of
  
 My field name
  
 when used in forms).
The order that fields are declared will affect their default order if a model is rendered in a form (e.g. in the Admin site), though this may be overridden.
The following common arguments can be used when declaring many/most of the different field types:
  
  
 Note:
  
 Apps created using
  
 manage.py
  
 set the type of the primary key to a
  
 BigAutoField
  
 .
You can see this in the local library
  
 catalog/apps.py
  
 file:
There are many other options — you can view the
  
 full list of field options here
  
 .
The following list describes some of the more commonly used types of fields.
There are many other types of fields, including fields for different types of numbers (big integers, small integers, floats), booleans, URLs, slugs, unique ids, and other "time-related" information (duration, time, etc.). You can view the
  
 full list here
  
 .
You can declare model-level metadata for your Model by declaring
  
 class Meta
  
 , as shown.
One of the most useful features of this metadata is to control the
  
 default ordering
  
 of records returned when you query the model type. You do this by specifying the match order in a list of field names to the
  
 ordering
  
 attribute, as shown above. The ordering will depend on the type of field (character fields are sorted alphabetically, while date fields are sorted in chronological order). As shown above, you can prefix the field name with a minus symbol (-) to reverse the sorting order.
So as an example, if we chose to sort books like this by default:
the books would be sorted alphabetically by title, from A-Z, and then by publication date inside each title, from newest to oldest.
Another common attribute is
  
 verbose\_name
  
 , a verbose name for the class in singular and plural form:
Other useful attributes allow you to create and apply new "access permissions" for the model (default permissions are applied automatically), allow ordering based on another field, or to declare that the class is "abstract" (a base class that you cannot create records for, and will instead be derived from to create other models).
Many of the other metadata options control what database must be used for the model and how the data is stored (these are really only useful if you need to map a model to an existing database).
The full list of metadata options are available here:
  
 Model metadata options
  
 (Django docs).
A model can also have methods.
  
  
 Minimally, in every model you should define the standard Python class method
  
 \_\_str\_\_()
  
 to return a human-readable string for each object.
  
 This string is used to represent individual records in the administration site (and anywhere else you need to refer to a model instance). Often this will return a title or name field from the model.
Another common method to include in Django models is
  
 get\_absolute\_url()
  
 , which returns a URL for displaying individual model records on the website (if you define this method then Django will automatically add a "View on Site" button to the model's record editing screens in the Admin site). A typical pattern for
  
 get\_absolute\_url()
  
 is shown below.
  
  
 Note:
  
 Assuming you will use URLs like
  
 /myapplication/mymodelname/2
  
 to display individual records for your model (where "2" is the
  
 id
  
 for a particular record), you will need to create a URL mapper to pass the response and id to a "model detail view" (which will do the work required to display the record). The
  
 reverse()
  
 function above is able to "reverse" your URL mapper (in the above case named
  
 'model-detail-view'
  
 ) in order to create a URL of the right format.
Of course to make this work you still have to write the URL mapping, view, and template!
You can also define any other methods you like, and call them from your code or templates (provided that they don't take any parameters).
  
  
 Note:
  
 Apps created using
  
 manage.py
  
 set the type of the primary key to a
  
 BigAutoField
  
 .
You can see this in the local library
  
 catalog/apps.py
  
 file:
  
  
 Note:
  
 Assuming you will use URLs like
  
 /myapplication/mymodelname/2
  
 to display individual records for your model (where "2" is the
  
 id
  
 for a particular record), you will need to create a URL mapper to pass the response and id to a "model detail view" (which will do the work required to display the record). The
  
 reverse()
  
 function above is able to "reverse" your URL mapper (in the above case named
  
 'model-detail-view'
  
 ) in order to create a URL of the right format.
Of course to make this work you still have to write the URL mapping, view, and template!
Once you've defined your model classes you can use them to create, update, or delete records, and to run queries to get all records or particular subsets of records. We'll show you how to do that in the tutorial when we define our views, but here is a brief summary.
To create a record you can define an instance of the model and then call
  
 save()
  
 .
  
  
 Note:
  
 If you haven't declared any field as a
  
 primary\_key
  
 , the new record will be given one automatically, with the field name
  
 id
  
 . You could query this field after saving the above record, and it would have a value of 1.
You can access the fields in this new record using the dot syntax, and change the values. You have to call
  
 save()
  
 to store modified values to the database.
You can search for records that match certain criteria using the model's
  
 objects
  
 attribute (provided by the base class).
  
  
 Note:
  
 Explaining how to search for records using "abstract" model and field names can be a little confusing. In the discussion below, we'll refer to a
  
 Book
  
 model with
  
 title
  
 and
  
 genre
  
 fields, where genre is also a model with a single field
  
 name
  
 .
We can get all records for a model as a
  
 QuerySet
  
 , using
  
 objects.all()
  
 . The
  
 QuerySet
  
 is an iterable object, meaning that it contains a number of objects that we can iterate/loop through.
Django's
  
 filter()
  
 method allows us to filter the returned
  
 QuerySet
  
 to match a specified
  
 text
  
 or
  
 numeric
  
 field against particular criteria. For example, to filter for books that contain "wild" in the title and then count them, we could do the following.
The fields to match and the type of match are defined in the filter parameter name, using the format:
  
 field\_name\_\_match\_type
  
 (note the
  
 double underscore
  
 between
  
 title
  
 and
  
 contains
  
 above). Above we're filtering
  
 title
  
 with a case-sensitive match. There are many other types of matches you can do:
  
 icontains
  
 (case insensitive),
  
 iexact
  
 (case-insensitive exact match),
  
 exact
  
 (case-sensitive exact match) and
  
 in
  
 ,
  
 gt
  
 (greater than),
  
 startswith
  
 , etc. The
  
 full list is here
  
 .
In some cases, you'll need to filter on a field that defines a one-to-many relationship to another model (e.g. a
  
 ForeignKey
  
 ). In this case, you can "index" to fields within the related model with additional double underscores.
So for example to filter for books with a specific genre pattern, you will have to index to the
  
 name
  
 through the
  
 genre
  
 field, as shown below:
  
  
 Note:
  
 You can use underscores (
  
 \_\_
  
 ) to navigate as many levels of relationships (
  
 ForeignKey
  
 /
  
 ManyToManyField
  
 ) as you like.
For example, a
  
 Book
  
 that had different types, defined using a further "cover" relationship might have a parameter name:
  
 type\_\_cover\_\_name\_\_exact='hard'.
  
  
 There is a lot more you can do with queries, including backwards searches from related models, chaining filters, returning a smaller set of values, etc. For more information, see
  
 Making queries
  
 (Django Docs).
  
  
 Note:
  
 If you haven't declared any field as a
  
 primary\_key
  
 , the new record will be given one automatically, with the field name
  
 id
  
 . You could query this field after saving the above record, and it would have a value of 1.
  
  
 Note:
  
 Explaining how to search for records using "abstract" model and field names can be a little confusing. In the discussion below, we'll refer to a
  
 Book
  
 model with
  
 title
  
 and
  
 genre
  
 fields, where genre is also a model with a single field
  
 name
  
 .
  
  
 Note:
  
 You can use underscores (
  
 \_\_
  
 ) to navigate as many levels of relationships (
  
 ForeignKey
  
 /
  
 ManyToManyField
  
 ) as you like.
For example, a
  
 Book
  
 that had different types, defined using a further "cover" relationship might have a parameter name:
  
 type\_\_cover\_\_name\_\_exact='hard'.
  
  
 In this section we will start defining the models for the library. Open
  
 models.py
  
 (in /locallibrary/catalog/). The boilerplate at the top of the page imports the
  
 models
  
 module, which contains the model base class
  
 models.Model
  
 that our models will inherit from.
Copy the
  
 Genre
  
 model code shown below and paste it into the bottom of your
  
 models.py
  
 file. This model is used to store information about the book category — for example whether it is fiction or non-fiction, romance or military history, etc.
As mentioned above, we've created the genre as a model rather than as free text or a selection list so that the possible values can be managed through the database rather than being hard coded.
The model has a single
  
 CharField
  
 field (
  
 name
  
 ), which is used to describe the genre (this is limited to 200 characters and has some
  
 help\_text
  
 ). At the end of the model, we declare a
  
 \_\_str\_\_()
  
 method, which returns the name of the genre defined by a particular record. No verbose name has been defined, so the field will be called
  
 Name
  
 in forms.
Copy the
  
 Book
  
 model below and again paste it into the bottom of your file. The
  
 Book
  
 model represents all information about an available book in a general sense, but not a particular physical "instance" or "copy" available for loan. The model uses a
  
 CharField
  
 to represent the book's
  
 title
  
 and
  
 isbn
  
 . For
  
 isbn
  
 , note how the first unnamed parameter explicitly sets the label as "ISBN" (otherwise, it would default to "Isbn"). We also set the parameter
  
 unique
  
 as
  
 true
  
 to ensure all books have a unique ISBN (the unique parameter makes the field value globally unique in a table). The model uses
  
 TextField
  
 for the
  
 summary
  
 , because this text may need to be quite long.
The genre is a
  
 ManyToManyField
  
 , so that a book can have multiple genres and a genre can have many books. The author is declared as
  
 ForeignKey
  
 , so each book will only have one author, but an author may have many books (in practice a book might have multiple authors, but not in this implementation!)
In both field types the related model class is declared as the first unnamed parameter using either the model class or a string containing the name of the related model. You must use the name of the model as a string if the associated class has not yet been defined in this file before it is referenced! The other parameters of interest in the
  
 author
  
 field are
  
 null=True
  
 , which allows the database to store a
  
 Null
  
 value if no author is selected, and
  
 on\_delete=models.SET\_NULL
  
 , which will set the value of the book's author field to
  
 Null
  
 if the associated author record is deleted.
  
  
 Warning:
  
 By default
  
 on\_delete=models.CASCADE
  
 , which means that if the author was deleted, this book would be deleted too! We use
  
 SET\_NULL
  
 here, but we could also use
  
 PROTECT
  
 or
  
 RESTRICT
  
 to prevent the author being deleted while any book uses it.
The model also defines
  
 \_\_str\_\_()
  
 , using the book's
  
 title
  
 field to represent a
  
 Book
  
 record. The final method,
  
 get\_absolute\_url()
  
 returns a URL that can be used to access a detail record for this model (for this to work, we will have to define a URL mapping that has the name
  
 book-detail
  
 , and define an associated view and template).
  
  
 Warning:
  
 By default
  
 on\_delete=models.CASCADE
  
 , which means that if the author was deleted, this book would be deleted too! We use
  
 SET\_NULL
  
 here, but we could also use
  
 PROTECT
  
 or
  
 RESTRICT
  
 to prevent the author being deleted while any book uses it.
Next, copy the
  
 BookInstance
  
 model (shown below) under the other models. The
  
 BookInstance
  
 represents a specific copy of a book that someone might borrow, and includes information about whether the copy is available or on what date it is expected back, "imprint" or version details, and a unique id for the book in the library.
Some of the fields and methods will now be familiar. The model uses:
We additionally declare a few new types of field:
The method
  
 \_\_str\_\_()
  
 represents the
  
 BookInstance
  
 object using a combination of its unique id and the associated
  
 Book
  
 's title.
  
  
 Note:
  
 A little Python:
  
  
 Note:
  
 A little Python:
Copy the
  
 Author
  
 model (shown below) underneath the existing code in
  
 models.py
  
 .
All of the fields/methods should now be familiar. The model defines an author as having a first name, last name, and dates of birth and death (both optional). It specifies that by default the
  
 \_\_str\_\_()
  
 returns the name in
  
 last name
  
 ,
  
 firstname
  
 order. The
  
 get\_absolute\_url()
  
 method reverses the
  
 author-detail
  
 URL mapping to get the URL for displaying an individual author.
All your models have now been created. Now re-run your database migrations to add them to your database.
Imagine a local benefactor donates a number of new books written in another language (say, Farsi). The challenge is to work out how these would be best represented in our library website, and then to add them to the models.
Some things to consider:
After you've decided, add the field. You can see what we decided on GitHub
  
 here
  
 .
Don't forget that after a change to your model, you should again re-run your database migrations to add the changes.
In this article we've learned how models are defined, and then used this information to design and implement appropriate models for the
  
 LocalLibrary
  
 website.
At this point we'll divert briefly from creating the site, and check out the
  
 Django Administration site
  
 . This site will allow us to add some data to the library, which we can then display using our (yet to be created) views and templates.
  
  
 Last modified:
  
  
  
 Aug 17, 2022
  
 ,
  
  
  
 by MDN contributors
  
  
 Your blueprint for a better internet.
Visit
  
  
  
 Mozilla Corporation’s
  
  
  
 not-for-profit parent, the
  
  
  
 Mozilla Foundation
  
 .
  
 Portions of this content are ©1998–
  
 2022
  
 by individual mozilla.org contributors. Content available under
  
  
  
 a Creative Commons license
  
 .
Your blueprint for a better internet.
Visit
  
  
  
 Mozilla Corporation’s
  
  
  
 not-for-profit parent, the
  
  
  
 Mozilla Foundation
  
 .
  
 Portions of this content are ©1998–
  
 2022
  
 by individual mozilla.org contributors. Content available under
  
  
  
 a Creative Commons license
  
 .
**bold** and some *italic.*