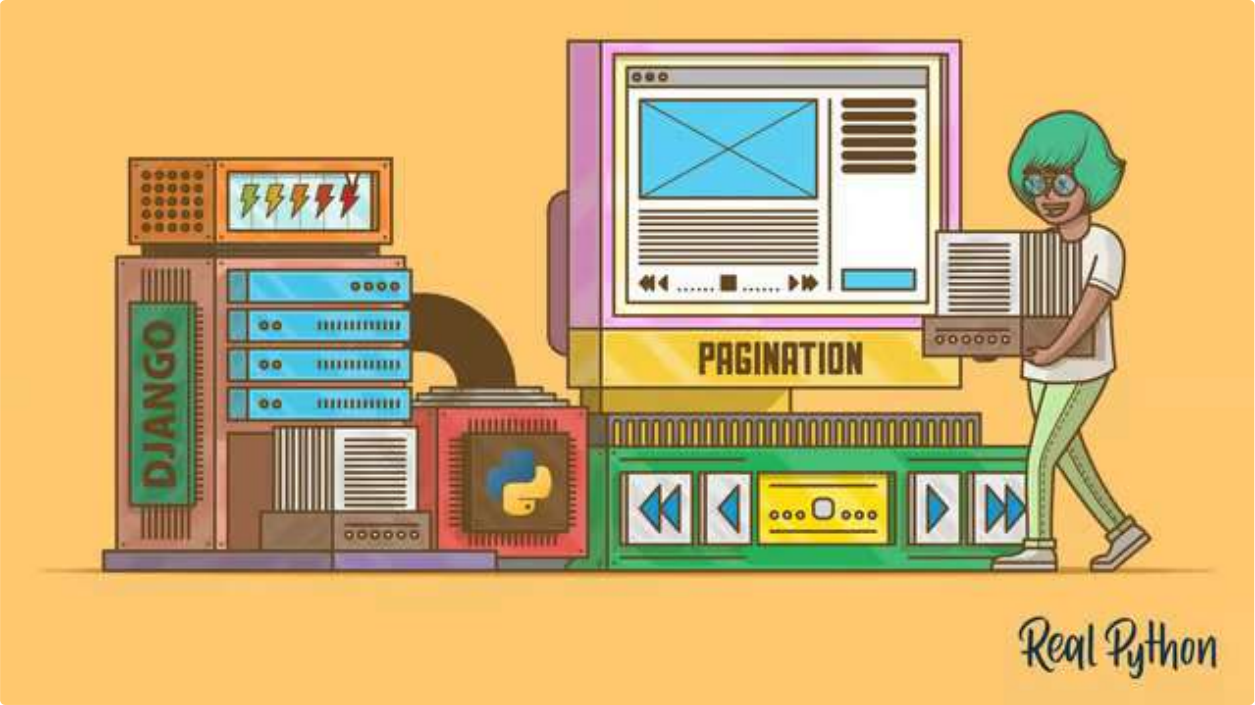


Pagination for a User-Friendly Django App

by Philipp Acsany · Mar 28, 2022 · 2 Comments · django · intermediate · web-dev

Mark as Completed

Share Share Email

Table of Contents

- Pagination in the Wild
  - What Pagination Is
  - When to Use Pagination
  - When Not to Use Pagination
- Django's Built-in Paginator
  - Preparing Your Sample Django Project
  - Exploring the Django Paginator in the Django Shell
  - Using the Django Paginator in Views
  - Responding With Paginated Data
  - Implementing Pagination URL Parameters
- Pagination in Django Templates
  - Current Page
  - All Pages
  - Elided Pages
  - Previous and Next
  - First and Last
  - Combined Example
- Dynamic JavaScript Pagination
  - Faux Pagination
  - Load More
  - Infinite Scrolling
  - Search
- Conclusion

Help

You can improve the user experience of your Django web app significantly by spreading your content over multiple pages instead of serving all of it at once. This practice is called **pagination**. To implement pagination, you have to consider page count, items per page, and the order of pages.

But if you're using Django for your web projects, you're in luck! Django has the pagination functionality built in. With only a few configuration steps, you can provide paginated content to your users.

In this tutorial, you'll learn how to:

- Decide when to use **Django's paginator**
- Implement pagination in **class-based views**
- Implement pagination in **function-based views**
- Add **pagination elements** to templates
- Browse pages directly with **paginated URLs**
- Combine Django pagination with **JavaScript calls**

This tutorial is for intermediate Python programmers with basic Django experience. Ideally, you've completed some introductory tutorials or created your own smaller Django projects. To have the best experience with this tutorial, you should know what models, views, and templates are and how to create them. If you need a refresher, then check out the tutorial on how to build a portfolio app with Django.

> **Get Source Code: Click here to get the source code** that you'll use to implement Django pagination.

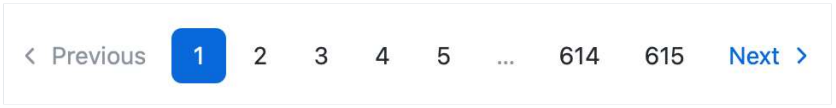
Pagination in the Wild

Before you try your hand at building your own pagination flows with Django, it's worth looking around to spot pagination in action. Pagination is so common on bigger websites that you've most likely experienced it in one form or another when browsing the Internet.

What Pagination Is

Pagination describes the practice of distributing your website's content across multiple consecutive pages instead of serving it on a single page. If you visit shopping sites, blogs, or archives, you're likely to encounter paginated content.

On GitHub, you'll find paginated content on Django's pull requests page. When you reach the bottom of the page, you can navigate to other pages:



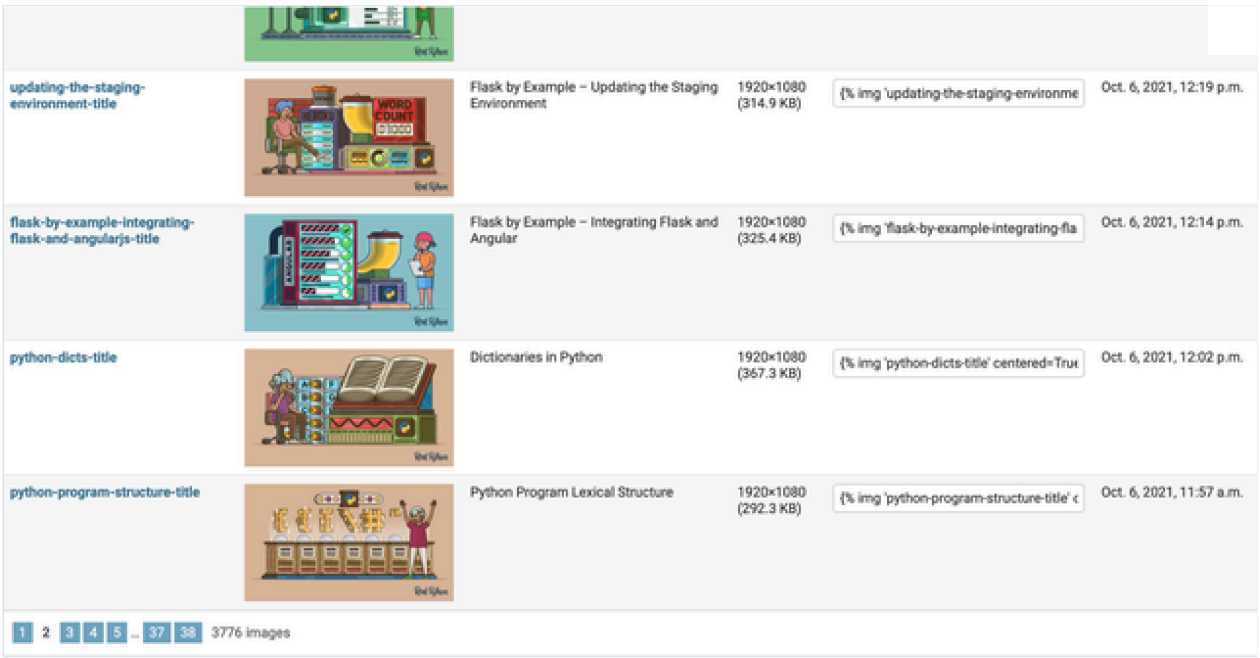
Imagine how crowded the bottom of the page would be if all the page numbers were displayed. What's more, consider how long the page would take to load if all the issues were displayed at once instead of being spread over 615 pages.

You could even argue that having page numbers is unnecessary. How could anybody know which issue is on which page? For that reason, some sites ditch page numbers entirely and give you a condensed form of pagination.

The PyCoder's Weekly Newsletter paginates its archive with *Previous* and *Next* buttons. This type of pagination lets you conveniently browse through all newsletter issues:

Underneath the *Subscribe* button, you see the pagination controls for navigating to the previous and next issues. Thanks to this pagination technique, you're able hop from one newsletter issue to another instead of selecting issues from the archive one by one.

You can also see pagination in action when you've got more than one hundred objects in your Django admin interface. To access more content, you have to click another page number:



Instead of showing a list of all 3,776 items, the Django admin divides the content into 38 pages. Again, imagine how overwhelming the Django admin interface would be if all the content were presented in one giant table!

But pagination is not only used in the front-end design of websites. It's also very common to paginate the content of **API responses**. The Random User API is one of many REST APIs that give you the option of paginating the response:

By adding a `results=2` parameter, you tell the Random User API that you only want two results per response. With the `page` parameter, you can navigate to a specific page of these paginated responses.

> **Note:** Do you have any interesting examples of websites or APIs that use pagination? Share them with the community in the comments below!

Once you know what pagination is, you'll probably notice it often while surfing the Web. In thinking about implementing pagination in your projects, it's worth taking a closer look at when to use pagination and when not to use it.

When to Use Pagination

Pagination is a great way to divide content into smaller chunks. The examples above highlight that it's a common practice on the Internet. This is rightfully so, as paginating your content has plenty of advantages:

- Sending less content at once speeds up your page load.
- Subsetting the content cleans up your website's user interface.
- Your content is easier to grasp.
- Your visitors don't have to scroll endlessly to reach the footer of your website.
- When you're not sending all the data at once, you reduce your server's payload of a request.
- You reduce the volume of items retrieved from your database.

Pagination can be helpful in structuring the content of your website, and it can improve your website's performance. Still, paginating your content isn't always the best choice. There are situations where it can be better not to paginate your content. Read on to learn more about when you'd be better off not using pagination.

When Not to Use Pagination

There are many advantages to using pagination. But it's not always the best choice. Depending on your use case, you might want to decide against using pagination for your user interface. You should consider the potential drawbacks to pagination:

- You interrupt continuous reading for your users.

- Your users have to navigate through results.
- Viewing more content requires new page loads.
- If there are too many pages, then it can become unwieldy to navigate.
- It slows your users down, so reading takes more time.
- Finding something specific within paginated data can be more difficult.
- Going back and forth between pages to compare information is cumbersome.

With a long list of benefits as well as an equally long list of drawbacks, you may be wondering when you should use pagination. Often it comes down to the amount of content and the user experience that you want to provide.

Here are two questions that you can ask yourself to help decide whether or not you should use pagination:

1. Is there enough content on your website to justify pagination?
2. Does pagination improve your website's user experience?

If you're still on the fence, then the convenience of implementing pagination with Django may be a convincing argument for you. In the next section, you'll learn how Django's built-in `Paginator` class works.

Django's Built-in Paginator

Django has a `Paginator` class that comes built in and ready to use. Perhaps you have a project on the go, and you'd like to try out the pagination implementations in the following sections with your app as your base. No problem! However, if you want to follow the step-by-step code examples in this tutorial, then you can download the source code for the Django Python wiki project from the Real Python materials repository:

> **Get Source Code: Click here to get the source code** that you'll use to implement Django pagination.

This wiki project contains an app called `terms`. For now, the app's only purpose is to show all the Python keywords. In the next section, you'll get a short overview of this sample project, which you'll use as the basis for the pagination in this tutorial. If you want to learn about the concept of Django pagination in general without using the provided sample project, then you can skip ahead to Exploring the Django Paginator in the Django Shell.

Preparing Your Sample Django Project

The pagination examples in this tutorial will work with any Django project. But for the purposes of this tutorial, you'll be working in a Python wiki. So that you can follow along closely, it's worth downloading the Python wiki Django sample project from the link above. To set up the Python wiki project, first follow the instructions in the accompanying `README.md` file.

The Python wiki sample project contains an app named `terms`, which includes a `Keyword` model:

Python

```python
# terms/models.py

from django.db import models

class Keyword(models.Model):
    name = models.CharField(max_length=30)

    def __str__(self):
        return self.name
```

The `Keyword` model consists of the `name` character field only. The string representation of a `Keyword` instance with the primary key `1` would be `Keyword object (1)` by default. When you add the `.__str__()` method, the name of `Keyword` is shown instead.

The Python wiki project already contains **migration** files. To work with your database, you must run the project's migrations. Select your operating system below and use your platform-specific command accordingly:

Windows | Linux + macOS

Windows Command Prompt

```
(venv) C:\> python manage.py migrate
```

After you've applied all migrations, your database contains the tables that your Django project requires. With the database tables in place, you can start adding content. To populate your project's database with the list of all the Python keywords, move into the folder of your Django project and start the **Django shell**:

Windows | Linux + macOS

Windows Command Prompt

```
(venv) C:\> python manage.py shell
```

Using the Django shell is a great way to interact with your Django project. You can conveniently try out code snippets and connect to the back-end without a front-end. Here, you programmatically add items to your database:

Python

```
1 >>> import keyword
2 >>> from terms.models import Keyword
3 >>> for kw in keyword.kwlist:
4 ...     k = Keyword(name=kw)
5 ...     k.save()
6 ...
```

First, you import Python's built-in `keyword` module in line 1. Afterward, you import the `Keyword` model from the `terms` app. In line 3, you loop through Python's keyword list. Finally, you create a `Keyword` class instance with the keyword string and save it to the database.

> **Note:** The **variable** names in the code block above are similar and hard to read. This is acceptable for smaller tasks in an interactive interpreter. When you want to write Python code that lasts, it's a good idea to think of better variable names.

To verify that your database contains the Python keywords, list them in the Django shell:

Python

```
>>> from terms.models import Keyword
>>> Keyword.objects.all()
<QuerySet [<Keyword: False>, <Keyword: None>, <Keyword: True>, ... ]>
```

When you import the `Keyword` model from your `terms` app, you can list all the items in your **database**. The database entries are all thirty-five Python keywords, arranged in the order that they were added to the database.

Your Python wiki project also contains a **class-based view** to show you all the keywords on one page:

Python

```
# terms/views.py

from django.views.generic import ListView
from terms.models import Keyword

class AllKeywordsView(ListView):
    model = Keyword
    template_name = "terms/base.html"
```

This view returns all database entries of the `Keyword` model. As a subclass of Django's generic `ListView`, it expects a template named `keyword_list.html`. However, by setting the `.template_name` attribute to `"terms/base.html"`, you tell Django to look for the base template instead. The other Django templates that you'll discover in this tutorial will extend the `base.html` template shown above.

> **Note:** If you want to follow along closely, make sure that you didn't skip the above step of populating the database.

Once you have the sample project in place, then you can run Django's built-in web server:

Windows | Linux + macOS

Windows Command Prompt

```
(venv) C:\> python manage.py runserver
```

When your development web server is running, visit `http://localhost:8000/all`. This page displays all the Python keywords in one continuous list. Later you'll create views to paginate this list with the help of Django's `Paginator` class. Read on to learn how the Django paginator works.

Exploring the Django Paginator in the Django Shell

Before you take a look at the Django paginator in detail, make sure that you've populated your database and entered the Django shell, as shown in the previous section.

The Django shell is perfect for trying out commands without adding code to your codebase. If you haven't already, start by importing the `Keyword` model:

Python

```
>>> from terms.models import Keyword
>>> from django.core.paginator import Paginator
>>> keywords = Keyword.objects.all().order_by("name")
>>> paginator = Paginator(keywords, per_page=2)
```

First, you import the `Paginator` class. Then you create a variable for your Django QuerySet named `keywords`. Because you don't filter the query set, `keywords` will contain all Python keywords that you listed in the previous section. Remember that Django's `QuerySets` are lazy:

> The act of creating a `QuerySet` doesn't involve any database activity. You can stack filters together all day long, and Django won't actually run the query until the `QuerySet` is evaluated. (Source)

In the example above, you created a query set for all the items in a database table. So when you hit the database with Django's paginator at some point, you're requesting a subset of your database's content. That way, pagination can speed up your Django app tremendously when you need to serve huge amounts of data from a database.

> **Note:** Pagination only makes sense when the database actively handles it. If you only perform pagination in memory on the back-end, then it won't solve the bottleneck of querying the database.
>
> The built-in Django `Paginator` class communicates which page to request and how many rows to get from the database. Depending on which database management system you're using, it'll translate to a slightly different SQL dialect.
>
> Strictly speaking, the database does the paging, and Django only requests a page number and offset. The Python and MySQL Database tutorial gives you a practical introduction to best practices when building database-heavy applications.

It's also important that you add some ordering to the **query set**. When you order your Python keywords by `name`, then you'll receive them in alphabetical order. Otherwise, you may get inconsistent results in your `keywords` list.

When initializing your `Paginator` class, you pass in `keywords` as the first **argument**. As the second argument, you must add an integer that defines how many items you want to show on a page. In this case, it's two. That means you want to display two items per page.

The Django `Paginator` class accepts four arguments. Two of them are required:

Argument	Required	Explanation
`object_list`	✅	Usually a Django QuerySet, but it can be any sliceable object with a `.count()` or `.__len__()` method, like a list or a tuple.
`per_page`	✅	Defines the number of items that you want to display on each page.
`orphans`		Declares the minimum number of items that you allow on the last page. If the last page has equal or fewer items, then they'll be added to the previous page. The default value is `0`, which means you can have a last page with any item count between one and the value you set for `per_page`.
`allow_empty_first_page`		Has a default value of `True`. If `object_list` is empty, then you'll get one empty page. Set `allow_empty_first_page` to `False` to raise an `EmptyPage` error instead.

Once you've created your `Paginator`, then you can access its **attributes**. Head back to the Django shell to see the paginator in action:

Python

```
>>> paginator.count
35
>>> paginator.num_pages
18
>>> paginator.page_range
range(1, 19)
>>> paginator.ELLIPSIS
"…"
```

The `.count` attribute of your `Paginator` class is the length of the `object_list` that you passed in. Remember that you wanted the paginator to show two items per page. The first seventeen pages will contain two items each. The last page will contain one only item. This makes eighteen pages total, as displayed by `paginator.num_pages`.

Since looping through your pages is a common task, the Django `Paginator` class provides you with the `.page_range` iterator directly as an attribute.

In the last line, you use the `.ELLIPSIS` attribute. This attribute comes in handy when you're not showing the whole page range to the user in the front-end. You'll see it in action in one of the examples later in this tutorial.

The Django `Paginator` class has four attributes:

Attribute	Explanation
`.ELLIPSIS`	The string displayed when you don't show the whole page range. The default value is an ellipsis (…).
`.count`	The total count of items across all pages. This is the length of your `object_list`.
`.num_pages`	The total number of pages.
`.page_range`	A range iterator of page numbers. Note that this iterator is 1-based and therefore starts with page number one.

Besides the attributes, the `Paginator` class contains three **methods**. Two of them look pretty similar at first glance. Start with the `.get_page()` method:

Python

```
>>> paginator.get_page(4)
<Page 4 of 18>
>>> paginator.get_page(19)
<Page 18 of 18>
>>> paginator.get_page(0)
<Page 18 of 18>
```

With `.get_page()`, you can access pages of `Paginator` directly. Note that pages in a Django paginator are indexed starting at one rather than zero. When you pass in a number outside of the page range, `.get_page()` returns the final page.

Now try the same with the `.page()` method:

Python

```
>>> paginator.page(4)
<Page 4 of 18>
>>> paginator.page(19)
Traceback (most recent call last):
  ...
django.core.paginator.EmptyPage: That page contains no results
>>> paginator.page(0)
Traceback (most recent call last):
  ...
django.core.paginator.EmptyPage: That page number is less than 1
```

Just like with `.get_page()`, you can access pages directly with the `.page()` method. Remember that pages are indexed starting at one rather than zero. The key difference from `.get_page()` is that if you pass in a number outside of the page range, then `.page()` raises an `EmptyPage` error. So using `.page()` allows you to be strict when a user requests a page that doesn't exist. You can catch the exception in the back-end and return a message to the user.

> **Note:** When you set `allow_empty_first_page` to `False` and your `object_list` is empty, then `.get_page()` will raise an `EmptyPage` error as well.

With `.get_page()` and `.page()`, you can acess a page directly. Besides these two methods, Django's `Paginator` contains a third method, called `.get_elided_page_range()`:

Python

```
>>> paginator.get_elided_page_range()
<generator object Paginator.get_elided_page_range at 0x1046c3e60>
>>> list(paginator.get_elided_page_range())
[1, 2, 3, 4, "…", 17, 18]
```

The `.get_elided_page_range()` method returns a **generator object**. When you pass that generator object into a `list()` function, you display the values that `.get_elided_page_range()` yields.

First, you pass no arguments in. By default, `.get_elided_page_range()` uses `number=1`, `on_each_side=3`, and `on_ends=2` as arguments. The yielded list shows you page `1` with its following three neighbors: `2`, `3`, and `4`. After that, the `.ELLIPSIS` string is shown to suppress all pages until the two last pages.

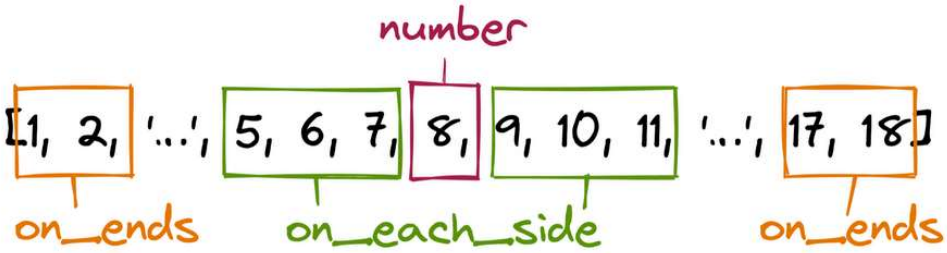
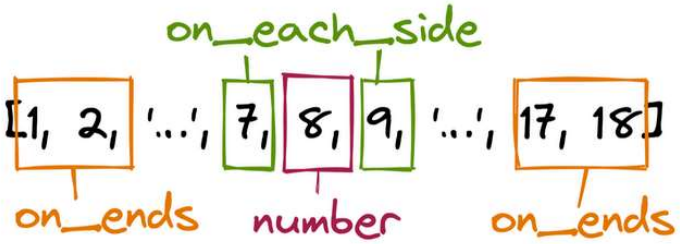
There are no pages before page `1`, so only pages after it are elided. That's why a number that's toward the middle of the page range showcases the capabilities of `.get_elided_page_range()` better:

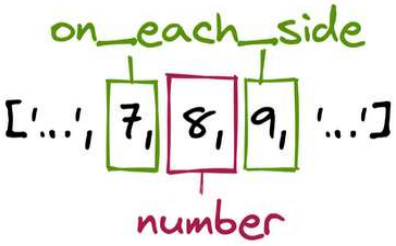
Python

```
>>> list(paginator.get_elided_page_range(8))
[1, 2, "…", 5, 6, 7, 8, 9, 10, 11, '…', 17, 18]
>>> list(paginator.get_elided_page_range(8, on_each_side=1))
[1, 2, "…", 7, 8, 9, "…", 17, 18]
>>> list(paginator.get_elided_page_range(8, on_each_side=1, on_ends=0))
["…", 7, 8, 9, "…"]
```

Notice how on each side of `8`, there are now three neighbors plus an ellipsis (…) and the first or last two pages. When you set `on_each_side` to `1`, then `7` and `9` are the only neighbors displayed. These are the pages immediately before and after page `8`. Finally, you set `on_ends` to `0`, and the first and last pages get elided, too.

To better understand how `.get_elided_page_range()` works, revisit the output from above with some annotation:

Arguments	Annotated Output
`number=8`	
`number=8` `on_each_side=1`	

`number=8`
`on_each_side=1`
`on_ends=0`
	

Trying out Django's `Paginator` class in the Django shell gave you a first impression of how pagination works. You got your feet wet by learning about the attributes and methods of Django's `Paginator`. Now it's time to dive in and implement pagination workflows in your Django views.

Using the Django Paginator in Views

Investigating the Django paginator in the Django shell is an excellent way to understand how the Django paginator behaves. However, using pagination in your **Django views** will reveal how powerful the Django paginator can be in structuring your content.

Django has two kinds of views: **class-based views** and **function-based views**. Both take a web request and return a web response. Class-based views are a good choice for generic views, like showing a list of database items.

While preparing your sample Django project, you already learned about `AllKeywordsView`. This class-based view returned all the keywords on one page, without paginating them. But you can paginate a class-based view in Django by adding the `.paginate_by` attribute to your view class:

Python

```
1  # terms/views.py
2
3  from django.views.generic import ListView
4  from terms.models import Keyword
5
6  class KeywordListView(ListView):
7      paginate_by = 5
8      model = Keyword
```

When you add the `.paginate_by` attribute to your view class in line 7, you limit the number of objects that each page shows. In this case, you'll show five objects per page.

Django also adds `.paginator` and `.page_obj` attributes to `.context` of the view's response. Also, the `ListView` expects a template whose name consists of the model's name in lowercase, followed by a `_list` suffix. Otherwise you'd have to define a `.template_name` attribute in the class. In the next section, you'll work with a template named `keyword_list.html`, so there's no need to add the `.template_name` attribute to `KeywordListView`.

In class-based views, adding the `.paginator`, `.page_obj`, and `.context` attributes happens under the hood. When you write a function-based view, you have to add them yourself. Update your `views.py` file to see both views side by side:

Python

```
1  # terms/views.py
2
3  from django.core.paginator import Paginator
4  from django.shortcuts import render
5
6  from terms.models import Keyword
7
8  # ...
9
10 def listing(request, page):
11     keywords = Keyword.objects.all().order_by("name")
12     paginator = Paginator(keywords, per_page=2)
13     page_object = paginator.get_page(page)
14     context = {"page_obj": page_object}
15     return render(request, "terms/keyword_list.html", context)
```

This function-based view does almost exactly what the class-based view above does. But you have to define the variables explicitly.

In line 3, you import `Paginator` for your use. You instantiate the `Paginator` class in line 12 with your `keywords` list and the `per_page` argument set to `2`.

So far, `listing()` contains the same functionality as `KeywordListView`. In line 13, you enhance the `listing()` view. You create `page_object` with `paginator.get_page()`. The `page` variable that you're passing in is available as a **URL parameter**. Later in this tutorial, you'll learn how to leverage the `page` parameter by implementing it within your URL definitions.

Finally, in line 15, you call the `render()` function with `request`, the template that you want to render, and a `context` dictionary. The `context` dictionary contains the `page_obj` value with `"page_obj"` as the key. You could name the key differently, but when you call it `page_obj`, you can use the same `keyword_list.html` template that your class-based view expects.

Both `KeywordListView` and `listing()` need templates to render their `context`. You'll create this template and the URLs to access the views later in this tutorial. Before you do, stick around in `views.py` for a bit to investigate how a paginated API endpoint works.

Responding With Paginated Data

Paginating your response is also a common practice when you design an API. When creating an API with Django, you can use frameworks like the Django REST framework. But you don't need external frameworks to build an API. In this section, you'll create a Django API endpoint *without* the Django REST framework.

The function body of the API view is similar to the `listing()` view that you created in the previous section. To spice things up, you'll implement more functionality so that your users can customize the API response with their GET request:

Python

```
1  # terms/views.py
2
3  from django.core.paginator import Paginator
4  from django.http import JsonResponse
5
6  from terms.models import Keyword
7
8  # ...
9
10 def listing_api(request):
11     page_number = request.GET.get("page", 1)
12     per_page = request.GET.get("per_page", 2)
13     startswith = request.GET.get("startswith", "")
14     keywords = Keyword.objects.filter(
15         name__startswith=startswith
16     )
17     paginator = Paginator(keywords, per_page)
18     page_obj = paginator.get_page(page_number)
19     data = [{"name": kw.name} for kw in page_obj.object_list]
20
21     payload = {
22         "page": {
23             "current": page_obj.number,
24             "has_next": page_obj.has_next(),
25             "has_previous": page_obj.has_previous(),
26         },
27         "data": data
28     }
29     return JsonResponse(payload)
```

There are a few things going on, so study the most significant lines in detail:

- **Line 4** imports `JsonResponse`, which will be the return type of `listing_api`.
- **Line 10** defines the view function named `listing_api`, which receives `request`.
- **Line 11** sets `page_number` to the value of the `page` GET parameter or defaults to `1`.
- **Line 12** sets `per_page` to the value of the `per_page` GET parameter or defaults to `2`.
- **Line 13** sets `startswith` to the value of the `startswith` GET parameter or defaults to an empty string.
- **Line 14** creates `keywords`, which is a `QuerySet` that contains either all the keywords or the ones that start with the letters that `startswith` contains.
- **Lines 17 and 18** create `Paginator` and a `Page` instance.
- **Line 19** creates a list with dictionaries that contain the Python keyword names.
- **Line 121** defines the `payload` dictionary with the data that you want to send to the user.
- **Line 29** returns `payload` as a JSON-encoded response.

With `listing_api()`, you created a function-based view as a flexible API endpoint. When a user sends a request to `listing_api()` without any GET parameters, then `JsonResponse` responds with the first page and your first two keywords. You also provide the flexibility to return fine-grained data to the user when they provide parameters.

The only piece that your Django JSON API endpoint is missing is a URL that it's connected to. Time to fix that!

Implementing Pagination URL Parameters

In the previous sections, you created three views that respond with paginated data: a `KeywordListView` class, a `listing()` function-based view, and an API endpoint named `listing_api()`. To access your views, you must create three **URLs**:

Python

```
1  # terms/urls.py
2
3  from django.urls import path
4  from . import views
5
6  urlpatterns = [
7      # ...
8      path(
9          "terms",
10         views.KeywordListView.as_view(),
11         name="terms"
12     ),
13     path(
14         "terms/<int:page>",
15         views.listing,
16         name="terms-by-page"
17     ),
18     path(
19         "terms.json",
20         views.listing_api,
21         name="terms-api"
22     ),
23 ]
```

You add a path to each corresponding view to the `urlpatterns` list. At first glance, it may seem odd that only `listing` contains a page reference. Don't the other views work with paginated data as well?

Remember that only your `listing()` function accepts a `page` parameter. That's why you refer to a page number with `<int:page>` as a **URL pattern** in line 14 only. Both `KeywordListView` and `listing_api()` will work solely with GET parameters. You'll access your paginated data with your web requests without the need for any special URL patterns.

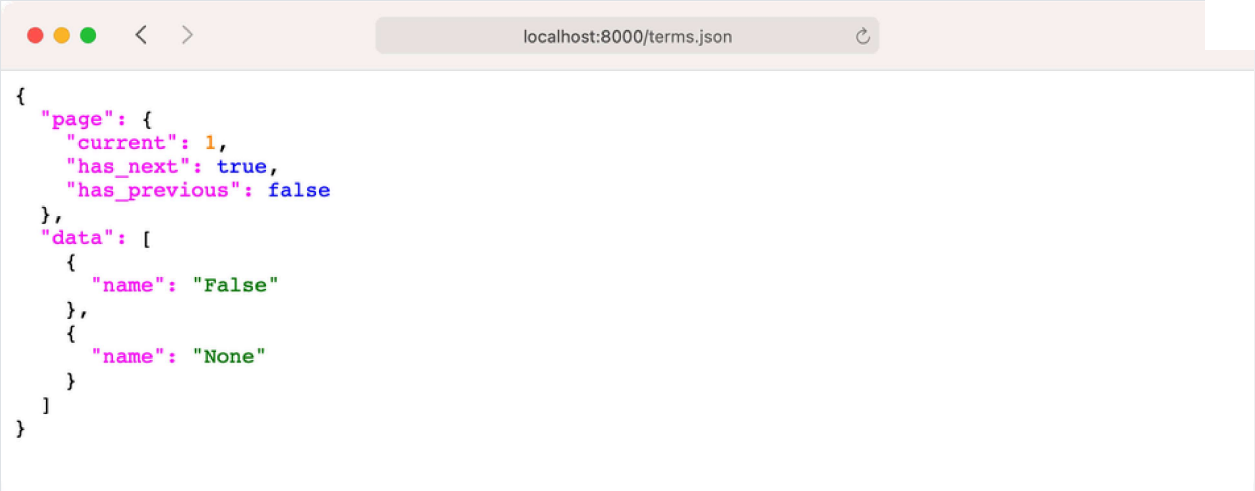
The `terms` URL and the `terms-by-page` URL both rely on templates that you'll explore in the next section. On the other hand, your `terms-api` view responds with a `JSONResponse` and is ready to use. To access your API endpoint, you must first start the Django **development server** if it's not already running:

Windows | Linux + macOS

Windows Command Prompt

```
(venv) C:\> python manage.py runserver
```

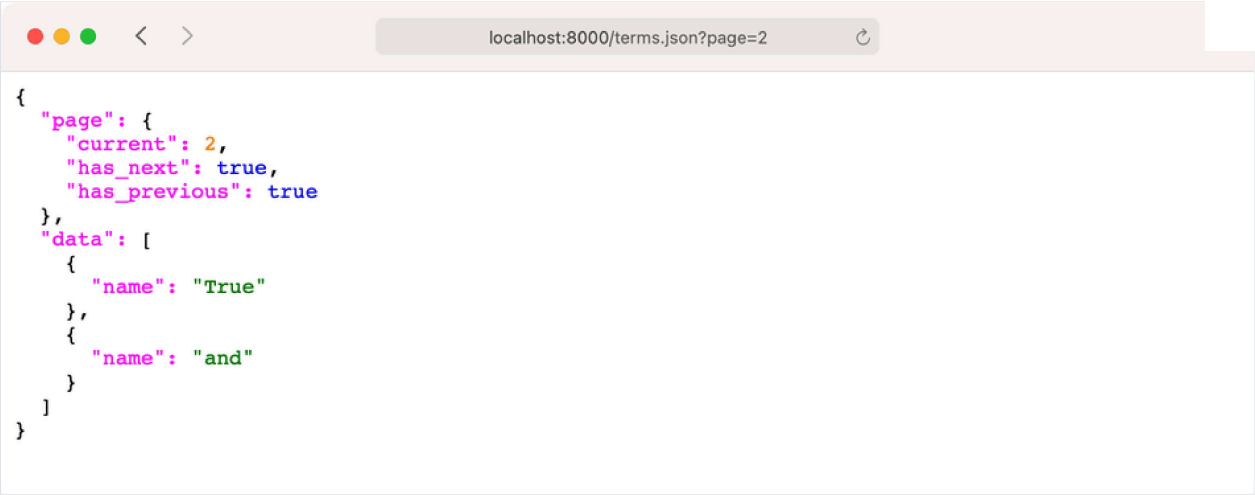
When the Django development server is running, then you can head to your browser and go to `http://localhost:8000/terms.json`:



When you visit `http://localhost:8000/terms.json` without adding any GET parameters, you'll receive the data for the first page. The returned JSON object contains information about the current page that you're on and specifies whether there's a previous or next page. The `data` object contains a list of the two first keywords, `False` and `None`.

> **Note:** Your browser may display the JSON response as unformatted text. You can install a JSON formatter extension for your browser to render JSON responses nicely.

Now that you know there's a page after page one, you can head over to it by visiting `http://localhost:8000/terms.json?page=2`:



When you add `?page=2` to the URL, you're attaching a GET parameter named `page` with a value of `2`. On the server side, your `listing_api` view checks for any parameters and recognizes that you specifically ask for page two. The JSON object that you get in return contains the keywords of the second page and tells you that there's a page before and another after this page.

You can combine GET parameters with an ampersand (`&`). A URL with multiple GET parameters can look like `http://localhost:8000/terms.json?page=4&per_page=5`:

localhost:8000/terms.json?page=4&per_page=5


```
{
  "page": {
    "current": 4,
    "has_next": true,
    "has_previous": true
  },
  "data": [
    {
      "name": "except"
    },
    {
      "name": "finally"
    },
    {
      "name": "for"
    },
    {
      "name": "from"
    },
    {
      "name": "global"
    }
  ]
}
```


You chain the `page` GET parameter and the `per_page` GET parameter this time. In return, you get the five keywords on page four of your dataset.

In your `listing_api()` view, you also added the functionality to look for a keyword based on its first letter. Head to `http://localhost:8000/terms.json?startswith=i` to see this functionality in action:

localhost:8000/terms.json?startswith=i


```
{
  "page": {
    "current": 1,
    "has_next": true,
    "has_previous": false
  },
  "data": [
    {
      "name": "if"
    },
    {
      "name": "import"
    }
  ]
}
```


By sending the `startswith` GET parameter with the value `i`, you're looking for all keywords that start with the letter *i*. Notice that `has_next` is `true`. That means there are more pages that contain keywords starting with the letter *i*. You can make another request and pass along the `page=2` GET parameter to access more keywords.

Another approach would be to add a `per_page` parameter with a high number like `99`. This would ensure that you'll get all matched keywords in one return.

Go on and try out different URL patterns for your API endpoint. Once you've seen enough raw JSON data, head to the next section to create some HTML templates with variations of pagination navigation.

Pagination in Django Templates

So far, you've spent most of your time in the **back-end**. In this section, you'll find your way into the **front-end** to explore various pagination examples. You'll try out a different pagination building block in each section. Finally, you'll combine everything that you've learned into one pagination widget.

To start things off, create a new **template** in the `terms/templates/terms/` directory, with the name `keyword_list.html`:

HTML

```
<!-- terms/templates/terms/keyword_list.html -->

{% extends "terms/base.html" %}

{% block content %}
    {% for kw in page_obj %}<pre>{{kw}}</pre>{% endfor %}
{% endblock %}
```

When you name your Django template `keyword_list.html`, both your `KeywordListView` and your `listing` can find the template. When you enter a URL in your browser, Django will resolve the URL and serve the matched view. When your Django development server is running, you can try out different URLs to see the `keyword_list.html` template in action.

> **Note:** This tutorial focuses on implementing pagination in Django templates. If you want to refresh your knowledge about Django's **templating engine**, then learning about built-in tags and filters is a good starting point. Afterward, you can explore implementing custom tags and filters in Django templates.

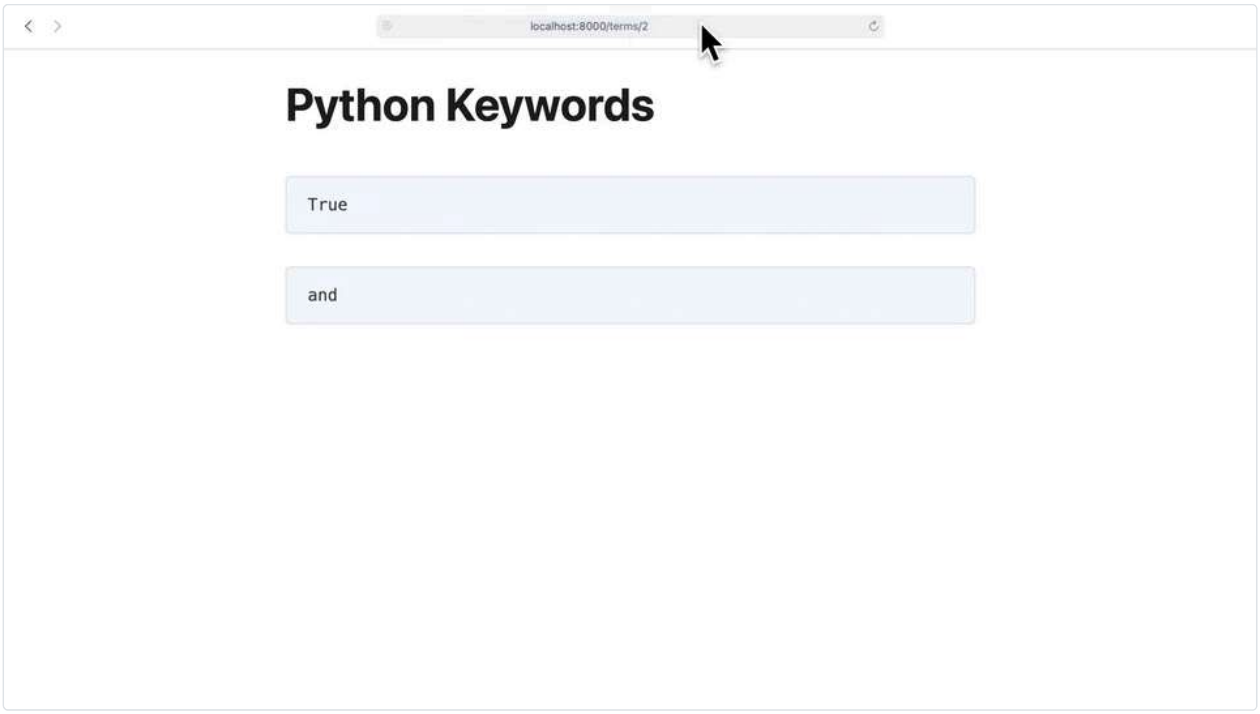
In your `urls.py` file, you gave `KeywordListView` the name `terms`. You can reach `terms` with URL patterns like the following:

- `http://localhost:8000/terms`
- `http://localhost:8000/terms?page=2`
- `http://localhost:8000/terms?page=7`

In your `urls.py` file, you gave your `listing()` view the name `terms-by-page`. You can reach `terms-by-page` with URL patterns like the following:

- `http://localhost:8000/terms/1`
- `http://localhost:8000/terms/2`
- `http://localhost:8000/terms/18`

Both views serve the same template and serve paginated content. However, `terms` shows five keywords while `terms-by-page` shows two. That's expected because you defined the `.paginate_by` attribute in `KeywordListView` differently from the `per_page` variable in the `listing()` view:



So far, you can control your pagination only by manually changing the URL. It's time to enhance your `keyword_list.html` and improve your website's user experience. In the following sections, you'll explore different pagination examples to add to your user interface.

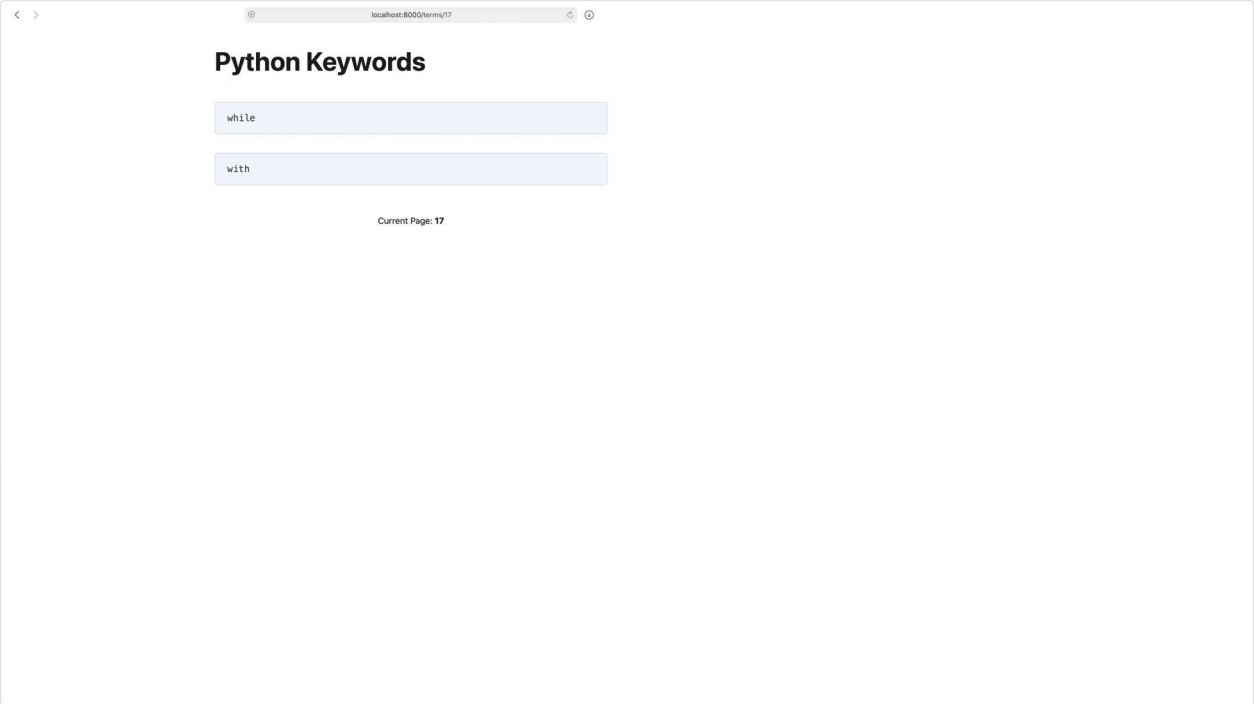
Current Page

This example shows the current page that you're on. Adjust `keyword_list.html` to display the current page number:

HTML

```
1  <!-- terms/templates/terms/keyword_list.html -->
2
3  {% extends "terms/base.html" %}
4
5  {% block content %}
6      {% for kw in page_obj %}<pre>{{kw}}</pre>{% endfor %}
7  {% endblock %}
8
9  {% block pagination %}
10     <p>Current Page: <b>{{page_obj.number}}</b></p>
11 {% endblock %}
```

In line 10, you're accessing the `.number` attribute of your `page_obj`. When you go to `http://localhost:8000/terms/2` the template variable will have the value `2`:



It's good to know which page you're on. But without proper navigation, it's still hard to go to another page. In the next section, you'll add navigation by linking to all available pages.

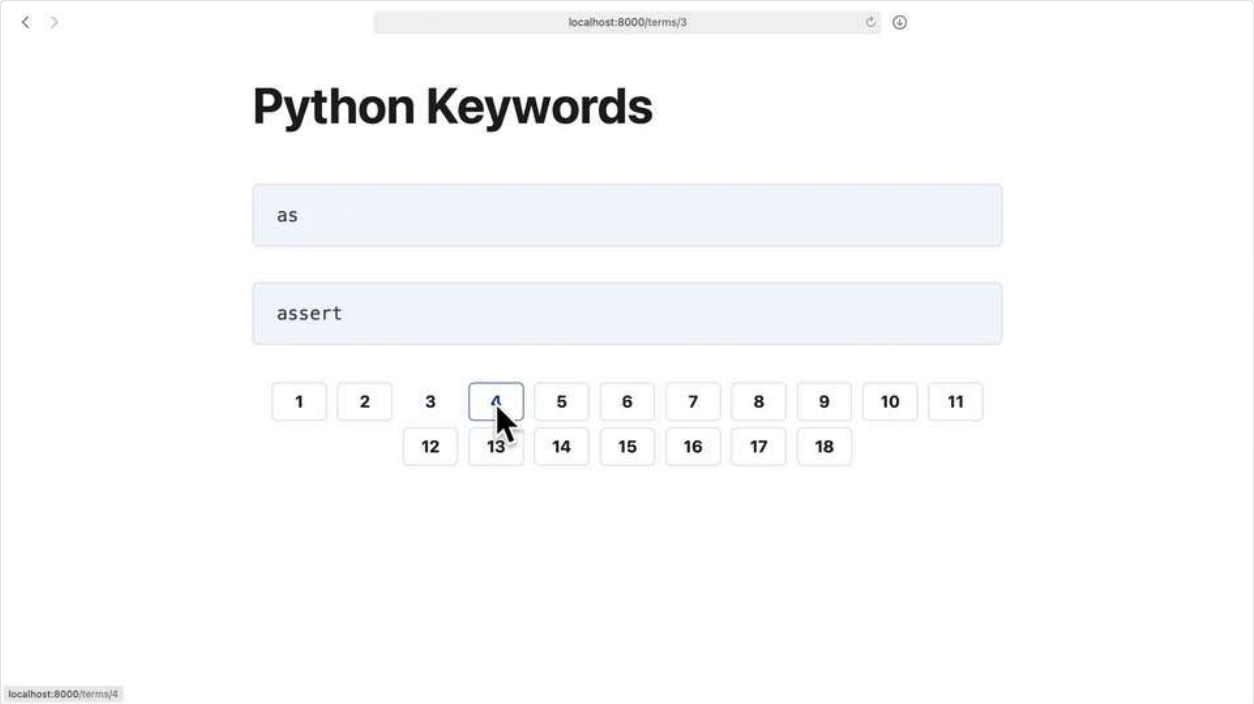
All Pages

In this example, you'll display all pages as clickable **hyperlinks**. You'll jump to another page without entering a URL manually. The Django paginator keeps track of all pages that are available to you. You can access the page numbers by iterating over `page_obj.paginator.page_range`:

HTML

```
1  <!-- terms/templates/terms/keyword_list.html -->
2
3  {% extends "terms/base.html" %}
4
5  {% block content %}
6      {% for kw in page_obj %}<pre>{{kw}}</pre>{% endfor %}
7  {% endblock %}
8
9  {% block pagination %}
10     {% for page_number in page_obj.paginator.page_range %}
11         <a
12             href="{% url 'terms-by-page' page_number %}"
13             class="{% if page_number == page_obj.number %}current{% endif %}"
14         >
15             {{page_number}}
16         </a>
17     {% endfor %}
18 {% endblock %}
```

In line 10, you're looping over all available page numbers. Then you display each `page_number` in line 15. Each page number is displayed as a link to navigate to the clicked page. If `page_number` is the current page, then you add a CSS class to it so that it looks different from the other pages. Go to `http://localhost:8000/terms/1` to see your *All Pages* pagination area in action:



You navigate to the corresponding page when you click a different page number. You can still spot your current page because it's styled differently in the list of page numbers. The current page number isn't surrounded by a square, and it's not clickable.

Elided Pages

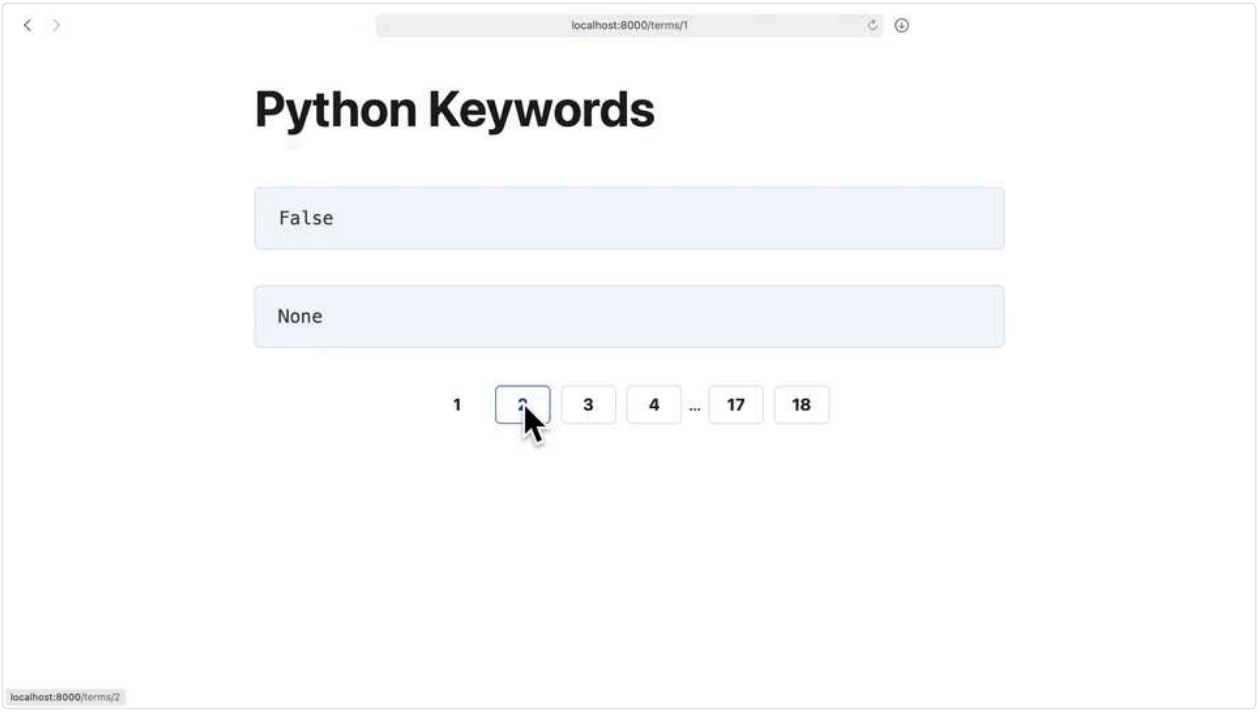
Showing all the pages might make sense if there aren't too many of them. But the more pages there are, the more cluttered your pagination area may get. That's when `.get_elided_page_range()` can help:

HTML

```
<!-- terms/templates/terms/keyword_list.html -->

{% extends "terms/base.html" %}

{% block content %}
    {% for kw in page_obj %}<pre>{{kw}}</pre>{% endfor %}
{% endblock %}

{% block pagination %}
    {% for page_number in page_obj.paginator.get_elided_page_range %}
        {% if page_number == page_obj.paginator.ELLIPSIS %}
            {{page_number}}
        {% else %}
            <a
                href="{% url 'terms-by-page' page_number %}"
                class="{% if page_number == page_obj.number %}current{% endif %}"
            >
                {{page_number}}
            </a>
        {% endif %}
    {% endfor %}
{% endblock %}
```

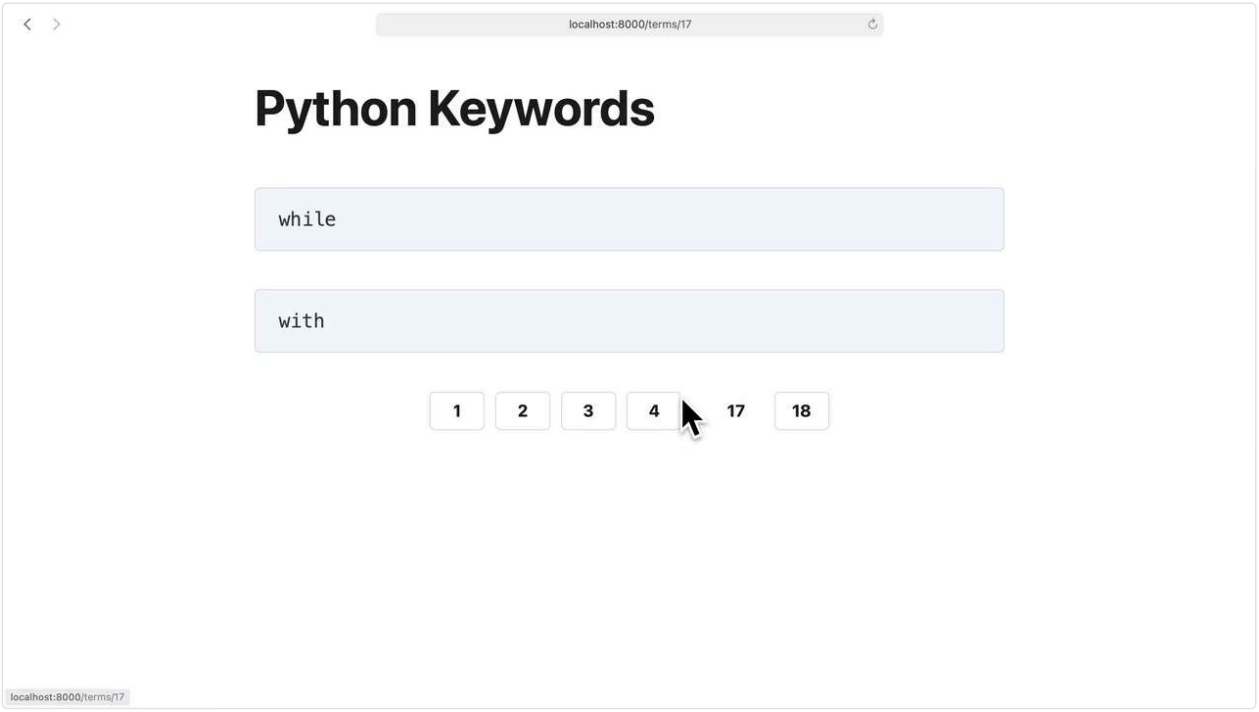
Instead of looping through all the pages, you're now looping through the elided pages list in line 10. In this example, that list includes the numbers *1* to *4*, an ellipsis, *17*, and *18*. When you reach the ellipsis in your list, you don't want to create a hyperlink. That's why you put it into an `if` statement in lines 11 to 13. The numbered pages should be hyperlinks, so you wrap them in an `a` tag in lines 14 to 19.

Visit `http://localhost:8000/terms/1` to see how your elided pages `Paginator` looks:



The elided pages paginator looks less cluttered than a paginator showing all pages. Note that when you visit `http://localhost:8000/terms`, you don't see an ellipsis at all, because you show five keywords per page. On page one, you show the next three pages and the final two pages. The only page that would be elided is page five. Instead of showing an ellipsis for only one page, Django displays the page number.

There's a caveat to using `.get_elided_page_range()` in a template, though. When you visit `http://localhost:8000/terms/7`, you end up on a page that's elided:



No matter which page you're on, the elided page range stays the same. You can find the reason for ending up on an elided page in the `page_obj.paginator.get_elided_page_range` loop. Instead of returning the elided pages list for the current page, `.get_elided_page_range()` always returns the elided pages list for the default value, `1`. To solve this, you need to adjust your elided pages configuration in the back-end:

Python

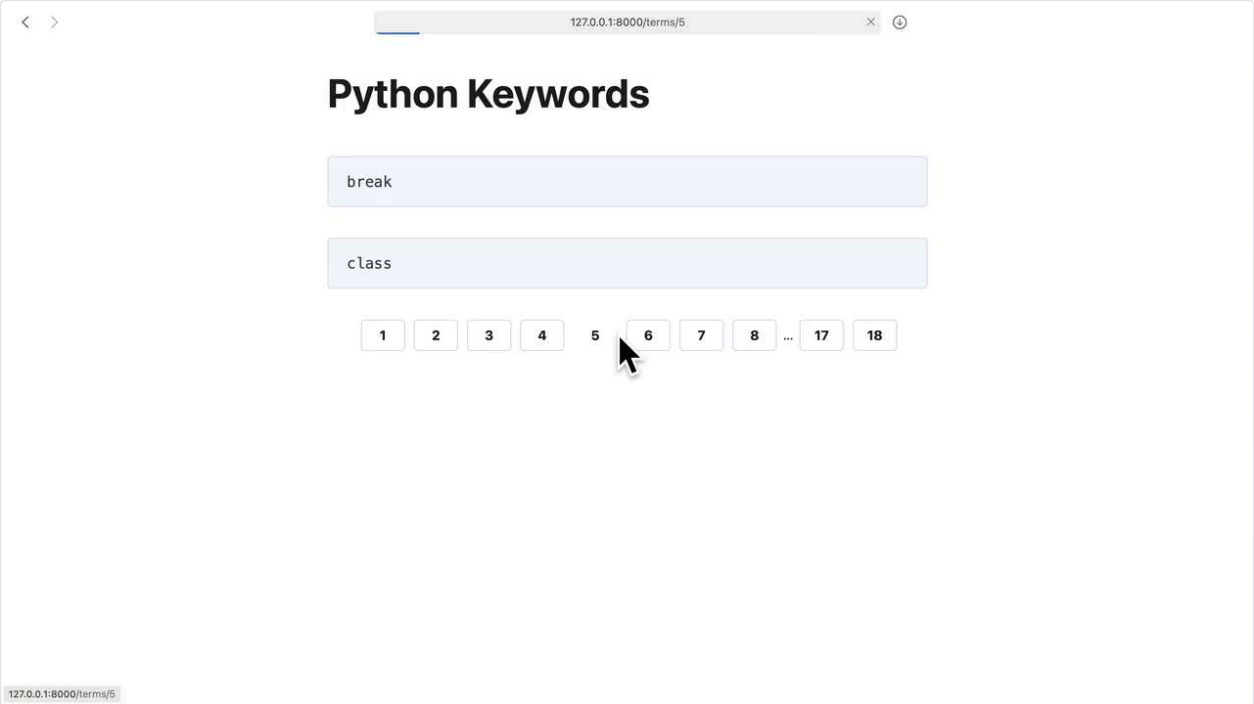
```python
# terms/views.py

# ...

def listing(request, page):
    keywords = Keyword.objects.all().order_by("name")
    paginator = Paginator(keywords, per_page=2)
    page_object = paginator.get_page(page)
    page_object.adjusted_elided_pages = paginator.get_elided_page_range(page)
    context = {"page_obj": page_object}
    return render(request, "terms/keyword_list.html", context)
```

In line 9, you're adding `adjusted_elided_pages` to `page_object`. Every time you call the `listing` view, you create an adjusted elided pages generator based on the current page. To reflect the changes in the front-end, you need to adjust the elided pages loop in `keyword_list.html`:

HTML

```html
<!-- terms/templates/terms/keyword_list.html -->

{% extends "terms/base.html" %}

{% block content %}
    {% for kw in page_obj %}<pre>{{kw}}</pre>{% endfor %}
{% endblock %}

{% block pagination %}
    {% for page_number in page_obj.adjusted_elided_pages %}
        {% if page_number == page_obj.paginator.ELLIPSIS %}
            {{page_number}}
        {% else %}
            <a
                href="{% url 'terms-by-page' page_number %}"
                class="{% if page_number == page_obj.number %}current{% endif %}"
            >
                {{page_number}}
            </a>
        {% endif %}
    {% endfor %}
{% endblock %}
```

With the changes in line 10, you're accessing the custom `page_obj.adjusted_elided_pages` generator, which considers the current page that you're on. Visit `http://localhost:8000/terms/1` and test your adjusted elided pages paginator:



Now you can click through all your pages, and the elided pages adjust accordingly. With the adjustments shown above in your `views.py` file, you can serve a powerful pagination widget to your users. A common approach is to combine an elided page widget with links to the previous and the next page.

Previous and Next

A pagination widget that elides pages is often combined with links to the previous and next page. But you can even get away with pagination that doesn't display page numbers at all:

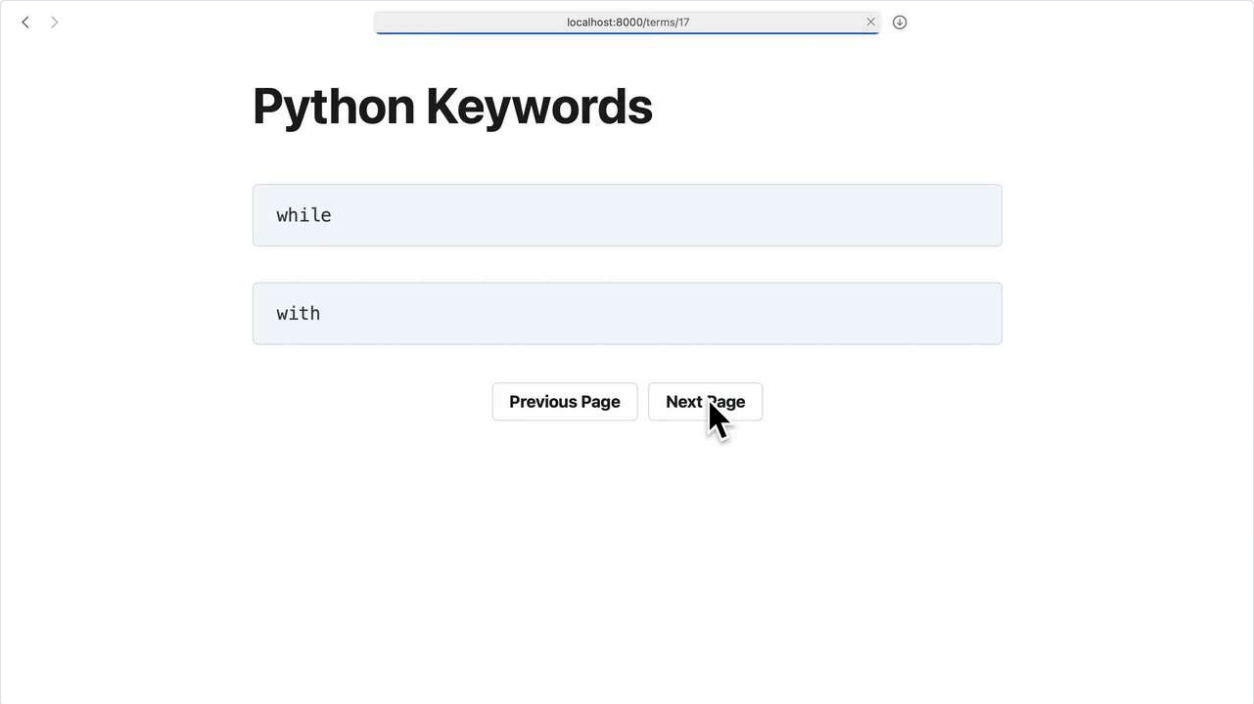
HTML

```
1  <!-- terms/templates/terms/keyword_list.html -->
2
3  {% extends "terms/base.html" %}
4
5  {% block content %}
6      {% for kw in page_obj %}<pre>{{kw}}</pre>{% endfor %}
7  {% endblock %}
8
9  {% block pagination %}
10     {% if page_obj.has_previous %}
11         <a href="{% url 'terms-by-page' page_obj.previous_page_number %}">
12             Previous Page
13         </a>
14     {% endif%}
15     {% if page_obj.has_next %}
16         <a href="{% url 'terms-by-page' page_obj.next_page_number %}">
17             Next Page
18         </a>
19     {% endif%}
20 {% endblock %}
```

In this example, you're not looping through any pages. You use `page_obj.has_previous` in lines 10 to 14 to check if the current page has a previous page. If there's a previous page, then you show it as a link in line 11. Notice how you don't even provide an actual page number but instead use the `.previous_page_number` attribute of `page_obj`.

In lines 15 to 19, you take the same approach in reverse. You check for the next page with `page_obj.has_next` in line 15. If there's a next page, then you show the link in line 16.

Go to `http://localhost:8000/terms/1` and navigate to some previous and next pages:



Notice how the *Previous Page* link disappears when you reach the first page. Once you're on the last page, there's no *Next Page* link.

First and Last

With the Django paginator, you can also provide your visitors a quick trip to the first or last page. The code from the previous example is only slightly adjusted in the highlighted lines:

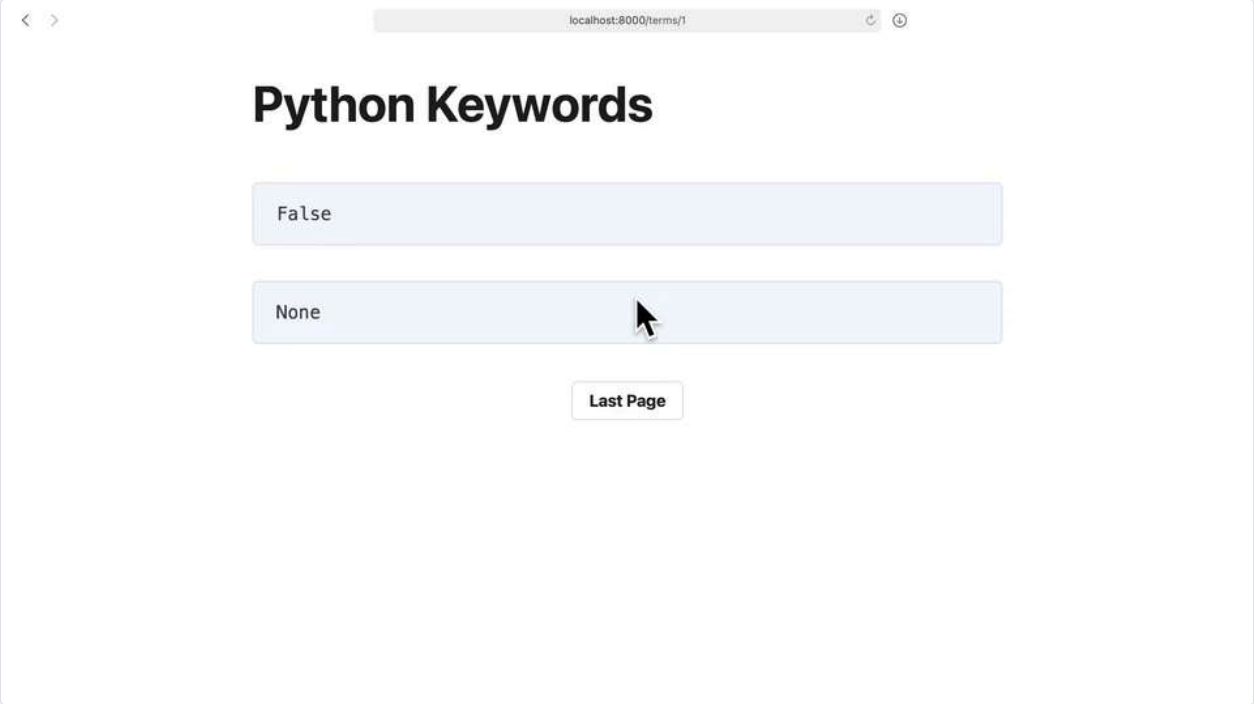
HTML

```
1  <!-- terms/templates/terms/keyword_list.html -->
2
3  {% extends "terms/base.html" %}
4
5  {% block content %}
6      {% for kw in page_obj %}<pre>{{kw}}</pre>{% endfor %}
7  {% endblock %}
8
9  {% block pagination %}
10     {% if page_obj.has_previous %}
11         <a href="{% url 'terms-by-page' 1 %}">
12             First Page
13         </a>
14     {% endif%}
15     {% if page_obj.has_next %}
16         <a href="{% url 'terms-by-page' page_obj.paginator.num_pages %}">
17             Last Page
18         </a>
19     {% endif%}
20 {% endblock %}
```

The logic that you implement in the `if` statements of lines 10 and 15 is the same as in the previous example. When you're on the first page, then there's no link to a previous page. And when you're on the last page, then there's no option to navigate further.

You can spot the difference in the highlighted lines. In line 11, you link directly to page `1`. That's the first page. In line 16, you use `page_obj.paginator.num_pages` to get the length of your paginator. The value of `.num_pages` is `18`, your last page.

Go to `http://localhost:8000/terms/1` and use your pagination links to jump from the first page to the last page:



Jumping from first to last and the reverse might be handy in some cases. But you usually want to give your users the chance to visit the pages in between.

Combined Example

You explored different approaches to implementing pagination in your front-end in the examples above. They all provide some navigation. But they still lack some features to provide a fulfilling user experience.

If you combine the pagination widgets together, then you can create a navigation element that your users will love:

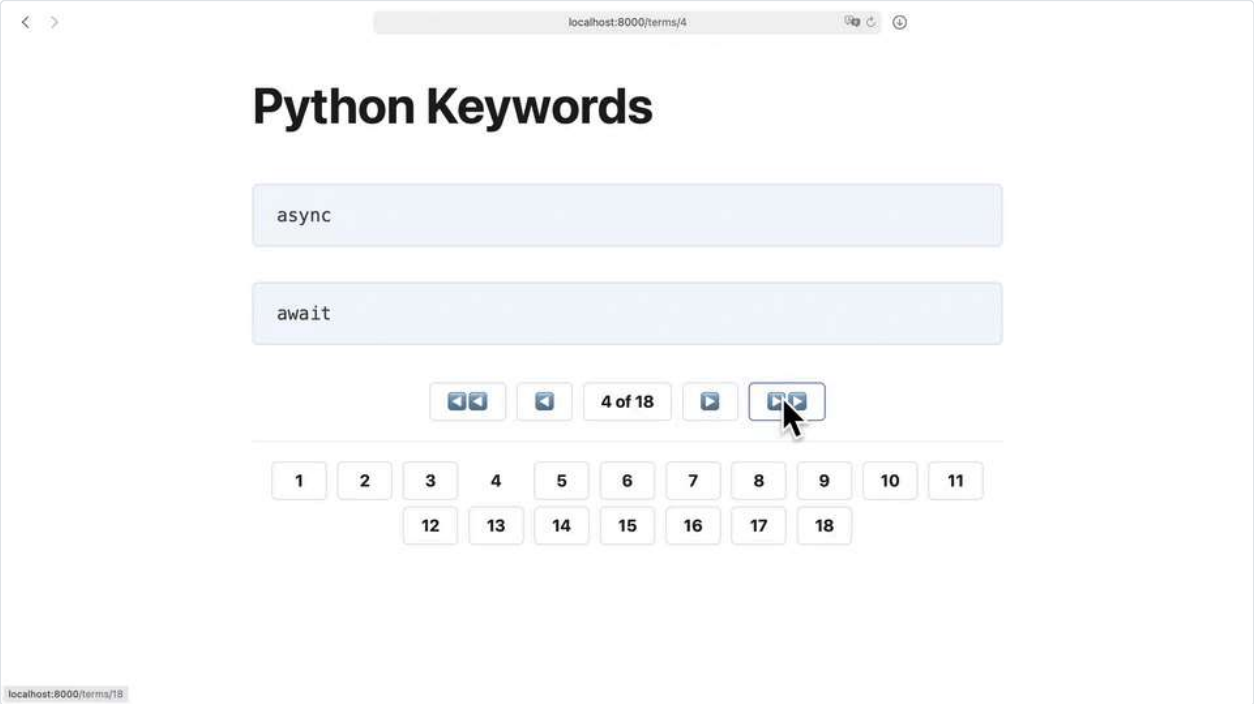
HTML

```
1  <!-- terms/templates/terms/keyword_list.html -->
2
3  {% extends "terms/base.html" %}
4
5  {% block content %}
6      {% for kw in page_obj %}<pre>{{kw}}</pre>{% endfor %}
7  {% endblock %}
8
9  {% block pagination %}
10     {% if page_obj.has_previous %}
11         <a href="{% url 'terms-by-page' 1 %}">
12             ◀◀
13         </a>
14         <a href="{% url 'terms-by-page' page_obj.previous_page_number %}">
15             ◀
16         </a>
17     {% endif%}
18
19     <a>{{page_obj.number}} of {{page_obj.paginator.num_pages}}</a>
20
21     {% if page_obj.has_next %}
22         <a href="{% url 'terms-by-page' page_obj.next_page_number %}">
23             ▶
24         </a>
25         <a href="{% url 'terms-by-page' page_obj.paginator.num_pages %}">
26             ▶▶
27         </a>
28     {% endif%}
29
30     <hr>
31
32     {% for page_number in page_obj.paginator.page_range %}
33         <a
34             href="{% url 'terms-by-page' page_number %}"
35             class="{% if page_number == page_obj.number %}current{% endif %}"
36         >
37             {{page_number}}
38         </a>
39     {% endfor %}
40 {% endblock %}
```

Here you use left-pointing arrows (◀) for the previous and the first page, and right-pointing arrows (▶) for the next and the last page. Apart from this, there are no changes to the examples that you already explored:

- **Line 10** checks if there's a previous page.
- **Lines 11 to 13** provide a link to the first page.
- **Lines 14 to 16** provide a link to the previous page.
- **Line 19** shows the current page and how many pages there are in total.
- **Line 21** checks if there's a next page.
- **Lines 22 to 24** provide a link to the next page.
- **Lines 25 to 27** provide a link to the last page.
- **Lines 32 to 39** create a list of all available pages.

Head to `http://localhost:8000/terms/1` to see this enriched pagination widget in action:



All these examples are merely building blocks for your own user interface. Maybe you come up with different solutions to create slick pagination for your web project. If you do, don't hesitate to share your code with the Real Python community in the comments below.

Dynamic JavaScript Pagination

Pagination helps you structure your content and serve data in chunks for a better user experience. But there are situations where pagination solutions like the ones above may not suit your needs. For example, maybe you want to serve content dynamically without loading a new page.

In this section, you'll learn about alternatives for paginating your content. You'll use JavaScript to control your pagination and leverage the API endpoint that you created earlier to serve the data accordingly.

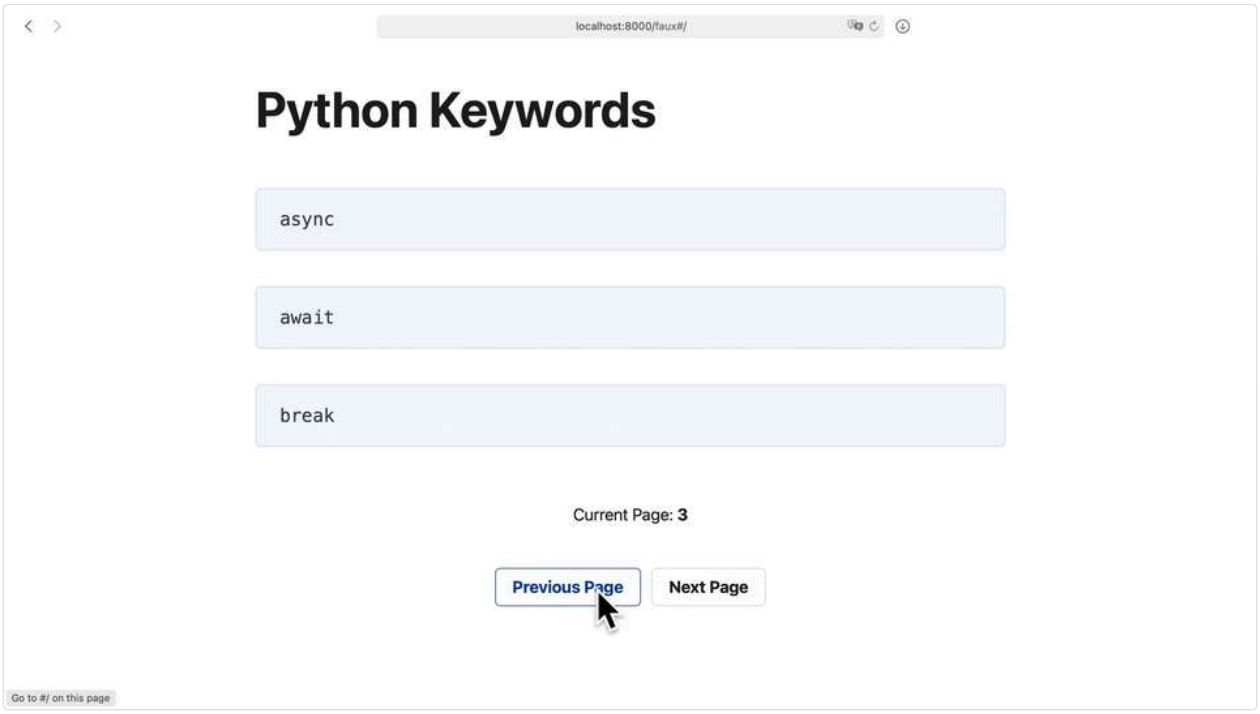
Faux Pagination

You can implement an alternative pagination functionality by loading the content dynamically when you press the *Previous* or *Next* button. The result looks like standard pagination, but you don't load a new page when you click on a pagination link. Instead you perform an **AJAX** call.

AJAX stands for Asynchronous JavaScript and XML. While the X stands for XML, it's more common nowadays to work with JSON responses instead.

> **Note:** You'll use JavaScript's Fetch API to load content asynchronously. This doesn't load a new page. That means you can't use your browser's *Back* button to go to previously loaded pages, and reloading a page will reset your scroll position.

Although you load the data differently, the result looks almost the same:



To create a dynamic front-end, you need to use **JavaScript**. If you're curious and want to give JavaScript a try, then you can follow the steps in the collapsible section below:

Faux Pagination Source Code Show/Hide

Implementing faux pagination functionality on your website can make sense when you don't control the data sent to the front-end. This is usually the case when you work with external APIs.

Load More

Sometimes you don't want to give the user the control to go back and forth between pages. Then you can give them the option to load more content when they want to see more content:

With the *Load more* functionality, you can lead your users deeper into your data without showing all the content at once. A notable example of this is loading hidden comments on GitHub's pull requests. When you want to implement this functionality in your project, then you can copy the source code below:

Load More Source Code Show/Hide

The *Load more* approach can be a smart solution when it makes sense to have all the content on one page, but you don't want to serve it all at once. For example, maybe you're presenting your data in a growing table.

With *Load more*, your users have to click actively to load more content. If you want to create a more seamless experience, then you can load more content automatically once your user reaches the bottom of your page.

Infinite Scrolling

Some web designers believe that clicking causes friction for their users. Instead of clicking *Load more*, the user should see more content once they reach the bottom of the page:

This concept is often referred to as **Infinite Scrolling** because it can seem like the content never reaches an end. *Infinite Scrolling* is similar to the *Load more* implementation. But instead of adding a link, you wait for a JavaScript event to trigger the functionality.

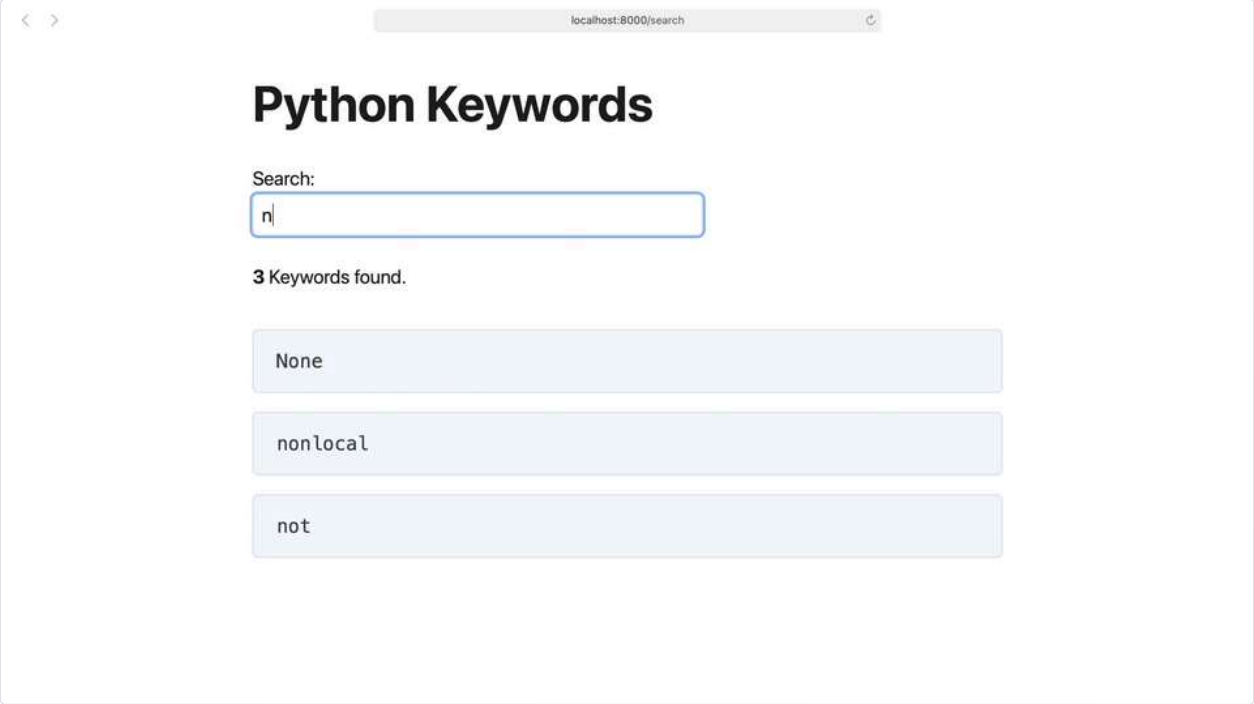
When you want to implement this functionality in your project, you can copy the source code below:

Infinite Scrolling Source Code Show/Hide

Adding *Infinite Scrolling* to your website creates a seamless experience. This approach is convenient when you have an ongoing image feed, like on Instagram.

Search

To stretch the pagination topic further, you can even think of performing a **search** as a way to paginate your data. Instead of showing all the content, you let the user decide what they want to see and then search for it:



When you want to implement this functionality in your project, you can copy the source code below:

Search Source Code Show/Hide

The examples above are a starting point to investigate dynamic pagination further. Because you're not loading new pages, your browser's *Back* button may not work as expected. Also, reloading the page resets the page or scroll position to the beginning unless you add other settings to your JavaScript code.

Still, combining a dynamic, JavaScript-flavored front-end with a reliable Django back-end creates a powerful basis for a modern web application. Building a blog using Django and Vue.js is a great way to explore the interaction of front-end and back-end further.

Conclusion

You can improve your Django web app significantly by paginating your content with the Django paginator. Subsetting the data that you show cleans up the user interface. Your content is easier to grasp, and the user doesn't have to scroll endlessly to reach the footer of your website. When you're not sending all the data to the user at once, you reduce the payload of a request, and your page responds more quickly.

In this tutorial, you learned how to:

- Implement pagination in **class-based views**
- Implement pagination in **function-based views**
- Add **pagination elements** to templates
- Browse pages directly with **paginated URLs**
- Create a dynamic pagination experience with **JavaScript**

You're now equipped with deep knowledge of when and how to use Django's paginator. By exploring multiple pagination examples, you learned what a pagination widget can include. If you want to put pagination into use, then building a portfolio, a diary, or even a social network provides a perfect pagination playground project.