'''\_

Let's create a simple web application framework. We'll need view

classes for each page on the site:

>>> home\_view = HomePageView() HTML Views ???

>>> home\_view.render()

'<html><body>Welcome!</body></html>'

>>> home\_view.body

'Welcome!'

>>> about\_view = AboutPageView() HTML Views ???

>>> about\_view.render()

'<html><body>This is a simple website about nutrition.</body></html>'

To route URLs to views, we'll create a global webapp object.

>>> app = WebApp()

It will keep track of the mappings with a protected variable:

>>> type(app.\_routes)

<class 'dict'>

Routings - that is, a mapping from a URL to a view object - are

created with add\_route(). Then you can get and render the view with

get().

>>> app.add\_route("/", home\_view)

>>> app.add\_route("/about/", about\_view)

>>> app.get("/")

'<html><body>Welcome!</body></html>'

>>> app.get("/about/")

'<html><body>This is a simple website about nutrition.</body></html>'

Let's create some more views. This site has two kinds of endpoints:

HTML pages, for browser content, and JSON objects, for API access.

>>> class ContactView(0):

... body = 'Get in touch at hello@example.com'

...

>>> app.add\_route("/contact/", ContactView())

>>> app.get("/contact/")

'<html><body>Get in touch at hello@example.com</body></html>'

(Hint for the JSON endpoints: One of the functions in the json module

takes an option to sort object keys.)

>>> class CarrotInfoView(JSONView):

... def data(self):

... return {

... 'serving\_size': 61,

... 'fat': 0.1,

... 'calories': 25,

... 'protein': 0.6,

... }

>>> carrot\_view = CarrotInfoView()

>>> carrot\_view.render()

'{"calories": 25, "fat": 0.1, "protein": 0.6, "serving\_size": 61}'

>>> app.add\_route("/api/carrot/", carrot\_view)

Create a view for baked chicken nutritional info:

>>> app.add\_route("/api/chicken/", ChickenInfoView())

>>> app.get("/api/chicken/")

'{"calories": 231, "fat": 5, "protein": 43, "serving\_size": 140}'

And another for tomato:

>>> app.add\_route("/api/tomato/", TomatoInfoView())

>>> app.get("/api/tomato/")

'{"calories": 22, "fat": 0.2, "protein": 1.1, "serving\_size": 123}'

And let's add a helper to give us a sorted list of all URLs defined so far.

>>> app.urls()

['/', '/about/', '/api/carrot/', '/api/chicken/', '/api/tomato/', '/contact/']

It's important to keep your class hierarchies well-organized. Since

every view has a render() method, it makes sense to put that in a

common base class.

>>> type(View.render)

<class 'function'>

>>> isinstance(home\_view, View)

True

>>> issubclass(ChickenInfoView, View)

True

And we can have more specialized views, FOR EXAMPLE WHEN WE WANT

OUR WRITING TO MAKE AN IMPACT:

>>> class LegalView(ShoutingHTMLView):

... body = 'you agree to our terms of service!'

>>> legal\_view = LegalView()

>>> app.add\_route("/legal/", legal\_view)

>>> app.get("/legal/")

'<HTML><BODY>YOU AGREE TO OUR TERMS OF SERVICE!</BODY></HTML>'

>>> isinstance(legal\_view, HTMLView)

True

Python lets you do something called "monkey patching". It can lead to

hard-to-understand code, so don't overuse it. But it can be useful

when working with certain libraries whose source you cannot/do not

want to modify, for example.

The idea is you modify a method of an already-created object, or a

superclass deep in an inheritance hierarchy, by assigning to it

directly. It works because in Python, (a) everything is an object, and

(b) methods are just attributes that can be assigned to.

>>> original\_htmlview\_render = HTMLView.render

>>> def new\_htmlview\_render(self):

... # Add <p> tag around content

... return '<html><body><p>' + self.body + '<p></body></html>'

>>> HTMLView.render = new\_htmlview\_render

Now that HTMLView is modified, instances of its subclasses are

modified too - provided that subclass reuses that method defined in

the superclass:

>>> legal\_view.render()

'<HTML><BODY><P>YOU AGREE TO OUR TERMS OF SERVICE!<P></BODY></HTML>'

'''

# Write your code here: