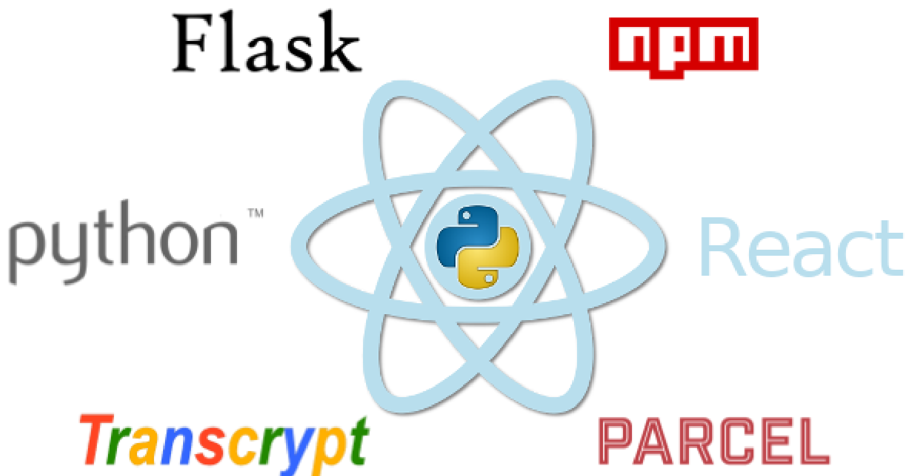
Creating React Applications with Python

A novel approach for achieving Full-Stack Python



John Sheehan

JennaSys™

Creating React Applications with Python
by John Sheehan

Copyright © 2021 by John Sheehan. All rights reserved.

Published May 2021

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor publisher shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

To report errors, please e-mail: rtp@jennasys.com

Source code used in this book is available at:
https://github.com/JennaSys/rtp_demo

For update notifications, subscribe at:
https://pyreact.com

[1346353]

About the Author

John Sheehan graduated with a degree in Computer Science & Engineering from the University of Illinois at Chicago, and has been programming primarily with Python for over a decade. As a freelance software developer, they have developed custom business software for scores of companies throughout the United States. In their free time, John is an avid DIYer, electronics enthusiast, and musician. They are currently based in sunny Southern California, and are the meetup organizer for the local Python and Raspberry Pi user groups.

Contents

About the Author . . . ii

1 Introduction **1**
- 1.1 Full-Stack Python . . . 1
- 1.2 Features of Transcrypt . . . 1
- 1.3 npm instead of pip . . . 2

2 Tutorial **3**
- 2.1 Installation . . . 3
- 2.2 Hello World . . . 3
- 2.3 Sourcemaps . . . 5
- 2.4 React . . . 6
- 2.5 Building a React Application . . . 8

3 For more... **15**
- 3.1 React to Python . . . 15
- 3.2 Resources . . . 16

Discount Offers! **17**

Introduction

1.1 Full-Stack Python

Let me start by getting this out of the way: *I really like programming in Python, and I'm not a big fan of JavaScript*. But let's face it, JavaScript is the way of the web, and Python doesn't run in a web browser. So end of story, right? Well not so fast, because just like the popular TypeScript language gets transpiled into JavaScript to run in a web browser, Transcrypt does the same thing for Python.

Because of the way Transcrypt maps Python data types and language constructs to JavaScript, your Python code is able to utilize the full ecosystem of JavaScript libraries that exist. Transcrypt acts as a bridge that enables you to take advantage of existing JavaScript web application technologies rather than trying to reinvent them. And, it does it in a way that doesn't significantly affect application performance over using plain JavaScript, or that requires a large runtime module to be downloaded to the client. And though we *use* JavaScript libraries, we don't have to code in JavaScript to use their APIs.

1.2 Features of Transcrypt

- It's PIP installable
- Python code is transpiled to JavaScript before being deployed
- It uses a very small JavaScript runtime (~40K)
- It can generate sourcemaps for troubleshooting Python in the browser
- The generated JavaScript is human-readable
- The generated JavaScript can be minified
- Performance is comparable to native JavaScript
- It maps Python data types and language constructs to JavaScript
- It acts as a bridge between the Python and JavaScript worlds
- It supports almost all Python built-ins and language constructs
- It only has limited support for the Python standard library
- Your Python code can "directly" call JavaScript functions
- Native JavaScript can call your Python functions
- It only supports 3rd party Python libraries that are pure Python

1.3 npm instead of pip

Most Python language constructs and built-ins have been implemented in Transcrypt, so working with standard Python objects like lists, dictionaries, strings, and more will feel just like Python should. Generally speaking however, third-party Python libraries are not supported unless the library (and its dependencies) are pure Python.

What this means is that instead of turning to `urllib` or the `requests` library when you need to make an HTTP request from your web browser application, you would utilize `window.fetch()` or the JavaScript `axios` library instead. But you would still code to those JavaScript libraries using Python.

Tutorial

2.1 Installation

Getting started with Transcrypt is pretty easy. Ideally, you would want to create a Python virtual environment for your project, activate it, and then use PIP to install Transcrypt:

$ `python3.7 -m venv venv`

$ `source venv/bin/activate`
(for Windows use `venv\Scripts\activate`)

(venv) $ `pip install transcrypt`

For the time being, Transcrypt only supports Python 3.7 so you will need to create your virtual environment with that version.

2.2 Hello World

With Transcrypt installed, we can try a simple *Hello World* web application to see how it works. We'll create two files: a Python file with a few functions, and an HTML file that we will open up in a web browser:

Listing 1: *hello.py*

```python
def say_hello():
    document.getElementById('destination').innerHTML = "Hello World!"

def clear_it():
    document.getElementById('destination').innerHTML = ""
```

Listing 2: *hello.html*

```html
<!DOCTYPE html>
<html lang="en">
    <body>
        <script type="module">
            import {say_hello, clear_it} from "./__target__/hello.js";
            document.getElementById("sayBtn").onclick = say_hello;
            document.getElementById("clearBtn").onclick = clear_it;
        </script>
        <button type="button" id="sayBtn">Click Me!</button>
        <button type="button" id="clearBtn">Clear</button>
```

```
        <div id="destination"></div>
    </body>
</html>
```

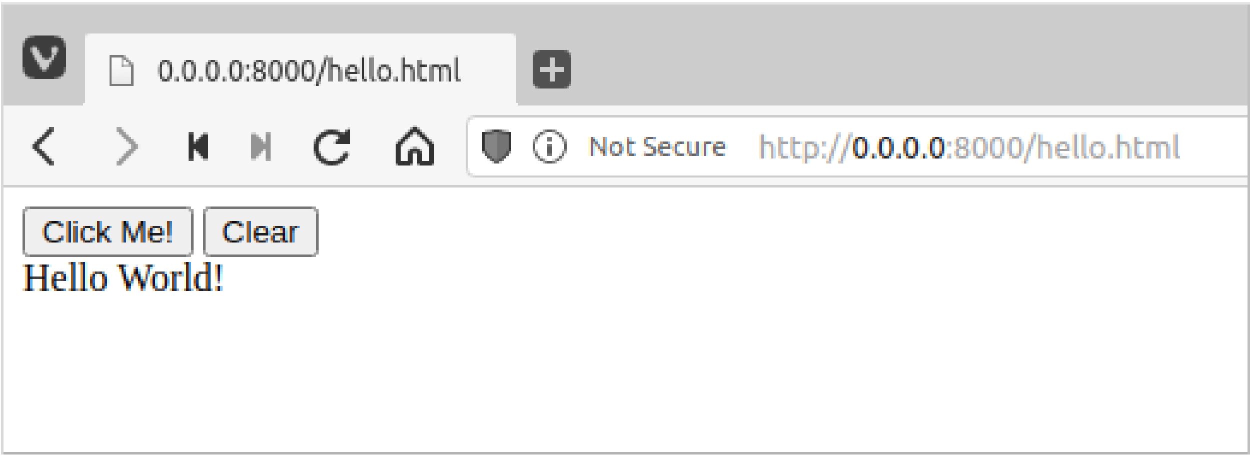
We then transpile the Python file with the Transcrypt CLI:
(venv) $ `transcrypt --nomin --map hello`

Here, we passed the `transcrypt` command three arguments:

- `--nomin` turns off minification to leave the generated code in a human-readable format
- `--map` generates sourcemaps for debugging Python code in the web browser
- `hello` is the name of python module to transpile

We can serve up the *Hello World* application using the built-in Python HTTP server:
(venv) $ `python -m http.server`

This starts up a webserver that serves up files in the current directory, from which we can open our HTML file at:
http://localhost:8000/hello.html



As you can see with this simple demonstration, we have Python calling methods of JavaScript objects using Python syntax, and JavaScript calling "Python" functions that have been transpiled. And as mentioned earlier, the generated JavaScript code is quite readable:

Listing 3 (Generated code): *__target__/hello.js*

```
// Transcrypt'ed from Python
import {AssertionError, ... , zip} from './org.transcrypt.__runtime__.js';
var __name__ = '__main__';
export var say_hello = function () {
   document.getElementById ('destination').innerHTML = 'Hello World!';
};
export var clear_it = function () {
   document.getElementById ('destination').innerHTML = '';
};

//# sourceMappingURL=hello.map
```

2.3 Sourcemaps

To demonstrate the sourcemap feature, we can again create two source files: a Python file with a function to be transpiled, and an HTML file that will be the entry point for our application in the web browser. This time, our Python file will have a function that outputs text to the web browser console using both JavaScript and Python methods, along with a JavaScript method call that will generate an error at runtime:

Listing 4: *sourcemap.py*

```python
def print_stuff():
    console.log("Native JS console.log call")
    print("Python print")
    console.invalid_method("This will be an error")
```

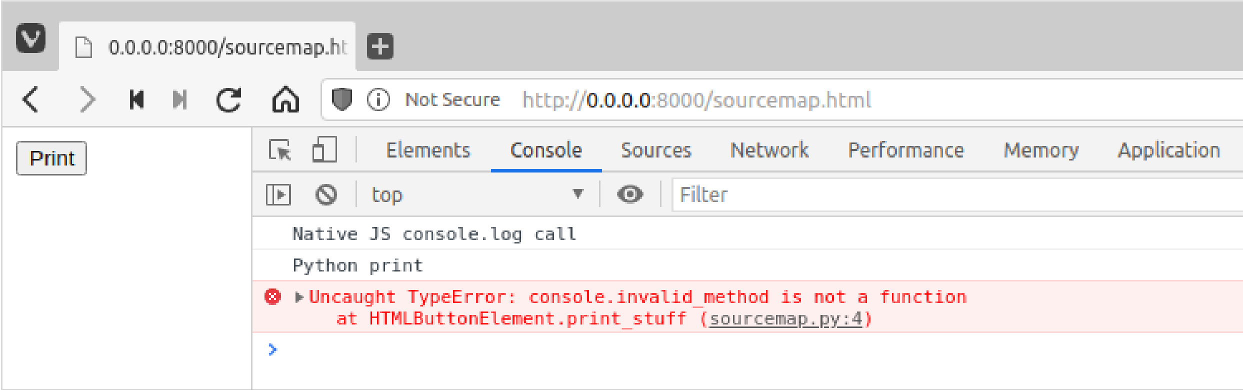
Listing 5: *sourcemap.html*

```html
<!DOCTYPE html>
<html lang="en">
    <body>
        <script type="module">
            import {print_stuff} from "./__target__/sourcemap.js";
            document.getElementById("printBtn").onclick = print_stuff;
        </script>
        <button type="button" id="printBtn">Print</button>
    </body>
</html>
```

(venv) $ `transcrypt --nomin --map sourcemap`

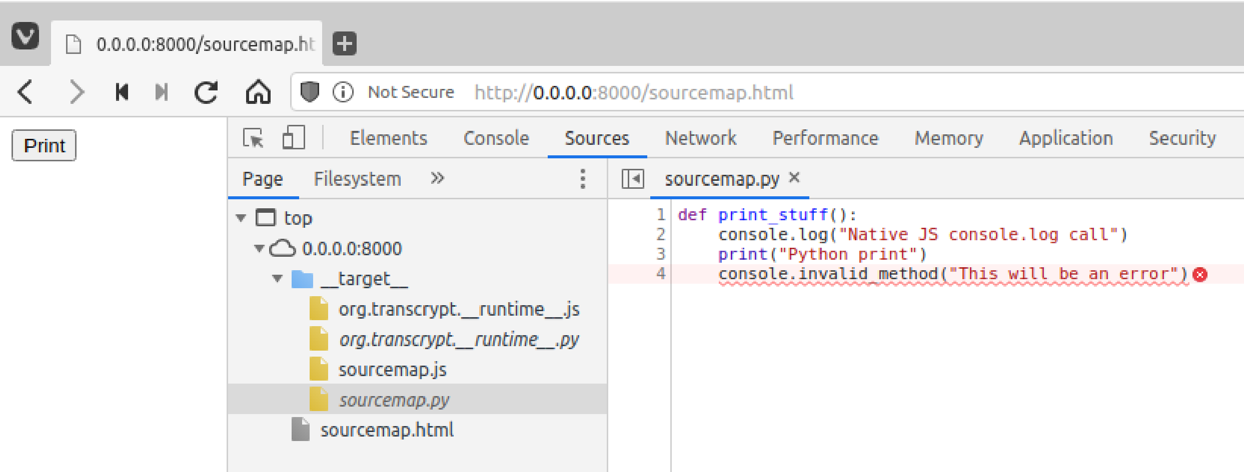
This time, with the built-in Python HTTP server started using:
(venv) $ `python -m http.server`

We can open our test application at:
http://localhost:8000/sourcemap.html



If you open the developer console in the web browser and then click the button, the first two calls will execute, printing the text to the web browser console. The call to the JavaScript `console.log()` method behaves as you would expect. But as you can see here, the Python `print()` function ends up getting transpiled to call the JavaScript `console.log()` method as well.

The third function call generates an error since we are trying to call a nonexistent method of the JavaScript `console` object. However, what's nice in this case is that the sourcemap can direct us to the cause of the problem in our *Python* source file. So, even though it is the generated JavaScript that is actually running in the web browser, using a sourcemap, we can still view our Python code right in the web browser and see where the error occurred in the Python file as well.



2.4 React

Now that we've seen how Transcrypt lets us make calls to JavaScript, let's step it up and use Transcrypt to make calls to the React library. We'll start with another simple *Hello World* application again, but this time doing it the React way. We'll stick with the two source files: a python file to be transpiled and an HTML file that will be opened in a web browser. The HTML file will be doing a little extra work for us in that it will be responsible for loading the React JavaScript libraries.

Listing 6: *hello_react.py*

```
useState = React.useState
el = React.createElement

def App():
    val, setVal = useState("")

    def say_hello():
        setVal("Hello React!")
```

```
    def clear_it():
        setVal("")

    return [
        el('button', {'onClick': say_hello}, "Click Me!"),
        el('button', {'onClick': clear_it}, "Clear"),
        el('div', None, val)
    ]

def render():
    ReactDOM.render(
        el(App, None),
        document.getElementById('root')
    )

document.addEventListener('DOMContentLoaded', render)
```

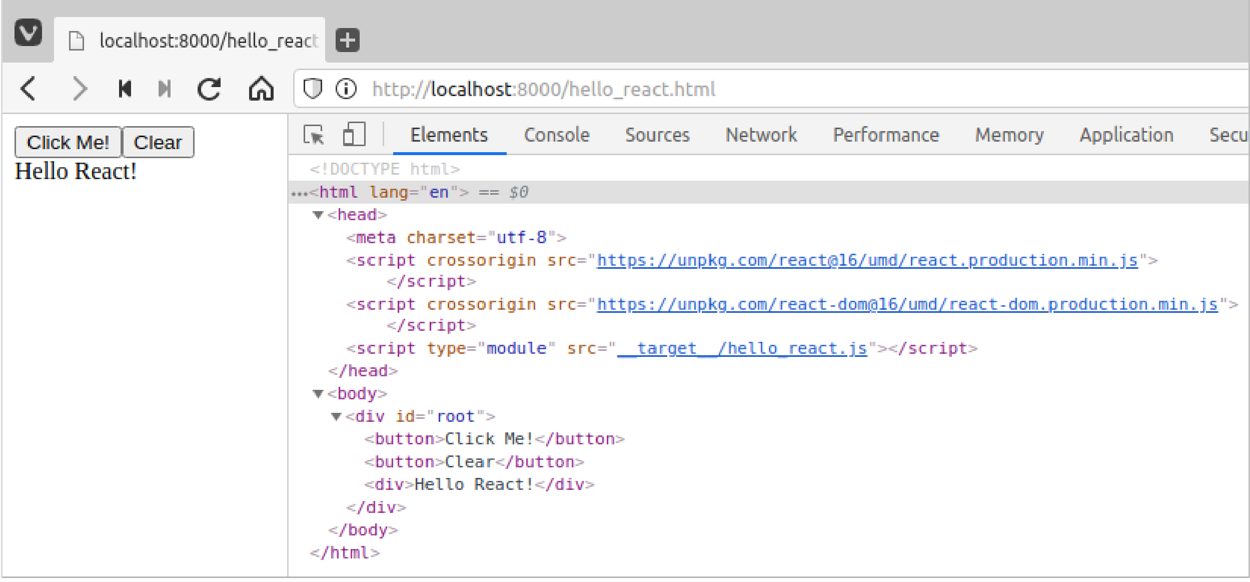
Listing 7: *hello_react.html*

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8"/>
    <script crossorigin
      src="https://unpkg.com/react@16/umd/react.production.min.js">
    </script>
    <script crossorigin
      src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js">
    </script>
    <script type="module" src="__target__/hello_react.js"></script>
  </head>
  <body>
    <div id="root">Loading...</div>
  </body>
</html>
```

Now transpile the Python file with Transcrypt:
(venv) $ `transcrypt --nomin --map hello_react`

Once again, after Transcrypt is done generating the JavaScript files, start up the built-in Python HTTP server using:
(venv) $ `python -m http.server`

Then open the demo React application at:
http://localhost:8000/hello_react.html



While functionally the same as the first demo application we did, this time React adds dynamically generated HTML as a child of a specified element - in this case, the "root" `div`.

Here, we added some convenience variables, `useState` and `el`, to map the global React methods to local Python variables. The React `createElement()` method is the workhorse of the library and is used to generate HTML elements in the browser dynamically.

React is declarative, functional, and is based on state. What this means, is that you define the view, and then React handles when and how it gets updated when there are changes in state. By design, React state variables are immutable and use a setter function to make updates. This helps React to know when changes to state occur, so it can then re-render the view as needed. In this example, we used the React `useState()` method to create the `val` variable and its corresponding `setVal()` setter function.

The return statement of a React functional component generally consists of a number of nested and chained calls to the React `createElement()` function that collectively form a tree structure of HTML elements and/or React components. This is where the view is declaratively defined. It may take some time to get more comfortable with this if you are not used to doing functional programming in Python.

The ReactDOM `render()` function takes the top-level React component and a reference to the HTML element to attach it to in the DOM. This is where it adds the dynamically generated HTML tree that React produces as a child of the specified element.

2.5 Building a React Application

Having done a simple React application, let's now create one that has a few more moving parts. This demo will take a value entered through the UI and add it to a list when submitted.

Most web applications of any utility will get large enough to where it becomes too unwieldy to manage manually. This is where package managers and application bundlers come into play. For this next example, we'll use the Parcel bundler to build and bundle the application so you can see what this developer stack might look like for larger applications.

First, we need to install the necessary JavaScript libraries to support the development toolchain. This does require Node.js to be installed on your system so that we can use the Node Package Manager. We start by initializing a new project and installing the Parcel bundler library along with the plug-in for Transcrypt:

```
$ npm init -y
$ npm install parcel-bundler --save-dev
$ npm install parcel-plugin-transcrypt --save-dev
```

Then we can install the React libraries:

```
$ npm install react@16 react-dom@16
```

Because of a version incompatibility, there is a file in the current Transcrypt plug-in that requires a patch. The file in question is:

./node_modules/parcel-plugin-transcrypt/asset.js

In that file, change line 2 that loads the Parcel `Logger` module from this:

```
const logger = require('parcel-bundler/src/Logger');
```

to this:

```
const logger = require('@parcel/logger/src/Logger');
```

Once this modification is made to change the location of the Parcel `Logger` module, the Transcrypt plug-in for Parcel should be working.

NOTE FOR WINDOWS USERS:

For those of you using Windows, two more changes need to be made to the **asset.js** file for it to work in Windows environments. The first is to modify the default Transcrypt build configuration to just use the version of Python that you set your virtual environment up with.

To do that, change line 14 that defines the Transcrypt command to simply use `python` instead of `python3`, changing it from this:

```
"command": "python3 -m transcrypt",
```

to this:

```
"command": "python -m transcrypt",
```

The second change has to do with modifying an import file path so that it uses Windows-style back-slashes instead of the Linux/Mac style forward-slashes. For this modification, we can use a string `replace()` method on line 143 to make an inline correction to the file path for Windows environments. So change this line:

```
this.content = `export * from "${this.importPath}";`;
```

to this:

```
this.content = `export * from "${this.importPath.replace(/\\/g, '/')}";`;
```

At some point, I would expect that a modification will be incorporated into the parcel-plugin-transcrypt package so that this hack can be avoided in the future.

Now that we have a bundler in place, we have more options as to how we work with JavaScript libraries. For one, we can now take advantage of the Node `require()` function that allows us to control the namespace that JavaScript libraries get loaded into. We will use this to isolate our Python-to-JavaScript mappings to one module, which keeps the rest of our code modules all pure Python.

Listing 8: *pyreact.py*

```python
# __pragma__ ('skip')
def require(lib):
    return lib

class document:
    getElementById = None
    addEventListener = None
# __pragma__ ('noskip')

# Load React and ReactDOM JavaScript libraries into local namespace
React = require('react')
ReactDOM = require('react-dom')

# Map React javaScript objects to Python identifiers
createElement = React.createElement
useState = React.useState


def render(root_component, props, container):
    """Loads main react component into DOM"""

    def main():
        ReactDOM.render(
            React.createElement(root_component, props),
```

```
        document.getElementById(container)
    )

document.addEventListener('DOMContentLoaded', main)
```

At the top of the file, we used one of Transcrypt's `__pragma__` compiler directives to tell it to ignore the code between the `skip/noskip` block. The code in this block doesn't affect the transpiled JavaScript, but it keeps any Python linter that you may have in your IDE quiet by stubbing out the JavaScript commands that are otherwise unknown to Python.

Next, we use the Node `require()` function to load the React JavaScript libraries into the module namespace. Then, we map the React `createElement()` and `useState()` methods to module-level Python variables as we did before. As we'll see shortly, this will allow us to import those variables into other Python modules. Finally, we moved the `render()` function we created previously into this module as well.

Now that we have the JavaScript interface somewhat self-contained, we can utilize it in our application:

Listing 9: *app.py*

```
from pyreact import useState, render, createElement as el


def ListItems(props):
    items = props['items']
    return [el('li', {'key': item}, item) for item in items]


def App():
    newItem, setNewItem = useState("")
    items, setItems = useState([])

    def handleSubmit(event):
        event.preventDefault()
        # setItems(items.__add__(newItem))
        setItems(items + [newItem])  # __:opov
        setNewItem("")

    def handleChange(event):
        target = event['target']
        setNewItem(target['value'])

    return el('form', {'onSubmit': handleSubmit},
              el('label', {'htmlFor': 'newItem'}, "New Item: "),
              el('input', {'id': 'newItem',
                           'onChange': handleChange,
                           'value': newItem
                           }
                 ),
              el('input', {'type': 'submit'}),
```

```
            el('ol', None,
               el(ListItems, {'items': items})
               )
            )


render(App, None, 'root')
```

As mentioned before, we import the JavaScript mappings that we need from the **pyreact.py** module, just like we would any other Python import. We aliased the React `createElement()` method to `el` to make it a little easier to work with.

If you're already familiar with React, you're probably wondering at this point why we're calling `createElement()` directly and not hiding those calls behind JSX. The reason has to do with the fact that Transcrypt utilizes the Python AST module to parse the PY files, and since JSX syntax is not valid Python, it would break that. There *are* ways to utilize JSX with Transcrypt if you really wanted to, but in my opinion the way you have to do it kind of defeats the purpose of using JSX in the first place.

In this module, we created two functional React components. The `App` component is the main entry point and serves as the top of the component tree that we are building. Here we have two state variables that we create along with their companion setter functions. The `newItem` state variable will hold an entered value that is to be added to the list. The `items` state variable will then hold all of the values that have been previously entered.

We then have two functions, one to perform an action when the form submits the value that was entered, and another that synchronizes the value that is being entered with the state of our React component.

Then, in the return statement of the `App()` function, we declare the tree of elements that define the UI. The top of the element tree starts with an HTML `form`. This allows us to take advantage of its default submit button, which in this case calls our `handleSubmit()` function that will add new values to the list.

In the `handleSubmit()` function, when adding a new item to our list, we used an in-line compiler directive to let Transcrypt know that this particular line of code is using an operator overload:

```
        setItems(items +  [newItem])  # __:opov`
```

By default, Transcrypt turns off this capability as it would cause the generated JavaScript to take a performance hit if it were enabled globally due to the overhead required to implement that feature. If you'd rather not use the compiler directive to enable operator overloading only where needed, in a case like this you could also call the appropriate Python operator overload dunder method directly as shown in the commented line just above it.

Inside (or below) that, we have an `input` element for entering new values along with a corresponding `label` element that identifies it in the UI. The `input` element has the `handleChange()` function as its `onChange` event handler that keeps the React state synced up with what the UI is showing.

Next in the element tree is the list of values that have already been entered. These will be displayed in the UI using an HTML ordered list element that will number the items that are added to it.

This brings us to this module's second functional component, `ListItems`, that renders the values in our `items` state variable as HTML `li` elements. The `items` are passed into this component as a property that we deconstruct into a local variable. From there, we use a Python list comprehension to build the list of `li` elements by iterating through the items.

The last step is to call the imported `render()` function that will attach our App React component to the DOM hook point identified by `'root'` in the HTML file:

```
render(App, None, 'root')
```

You'll notice that because we put all of the Python-to-JavaScript mappings in the **pyreact.py** module, that this module can be 100% pure pythonic Python. No mixing of languages, no weird contortions of the Python language, and no JavaScript!

To complete this demo, we now just need an HTML entry point that we can load into a web browser:

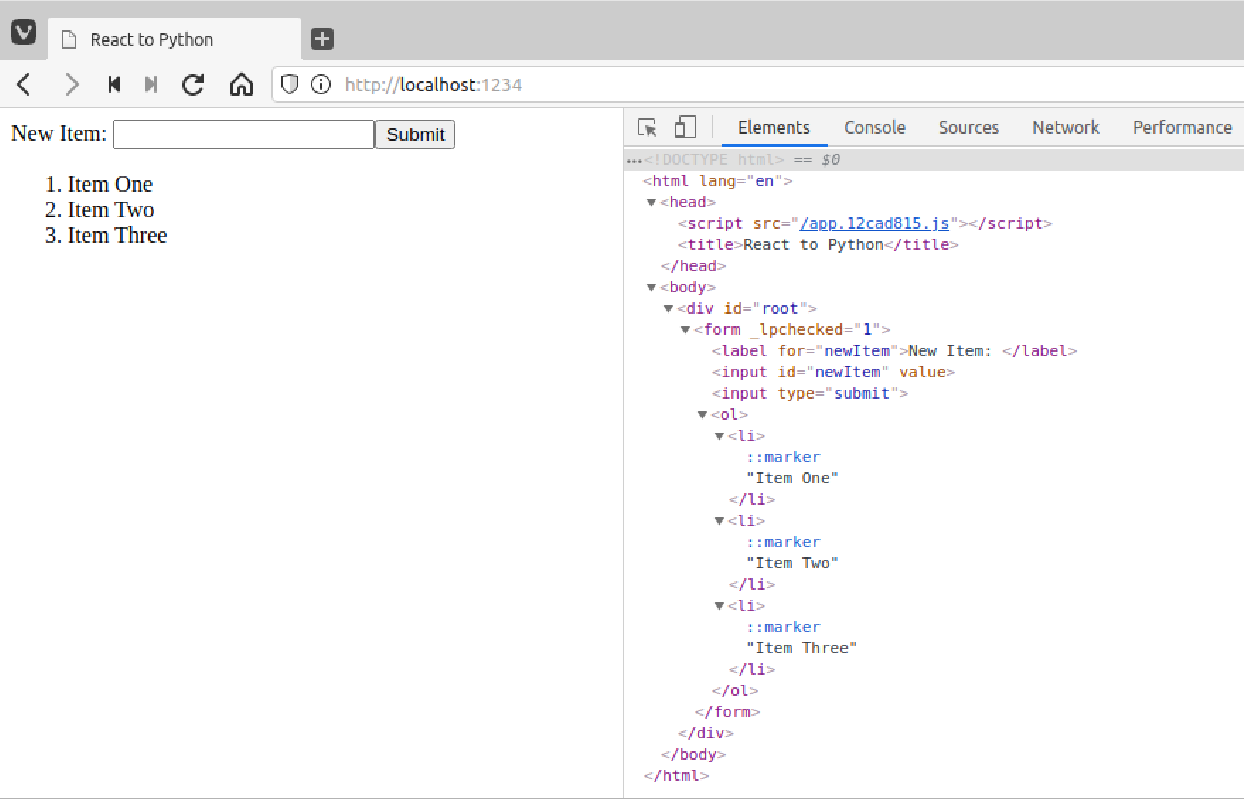
Listing 10: *index.html*

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <script src="app.py"></script>
        <title>React to Python</title>
    </head>
    <body>
        <div id="root"></div>
    </body>
</html>
```

This time, instead of running Transcrypt directly, we can run the `parcel` command using the Node npx package runner. And thanks to the Transcrypt Parcel plugin, it will also run Transcrypt for us and bundle up the generated JavaScript files:

(venv) $ `npx parcel --log-level 4 --no-cache index.html`

This also starts up the Parcel development webserver that will serve up the generated content using a default route at:
http://localhost:1234

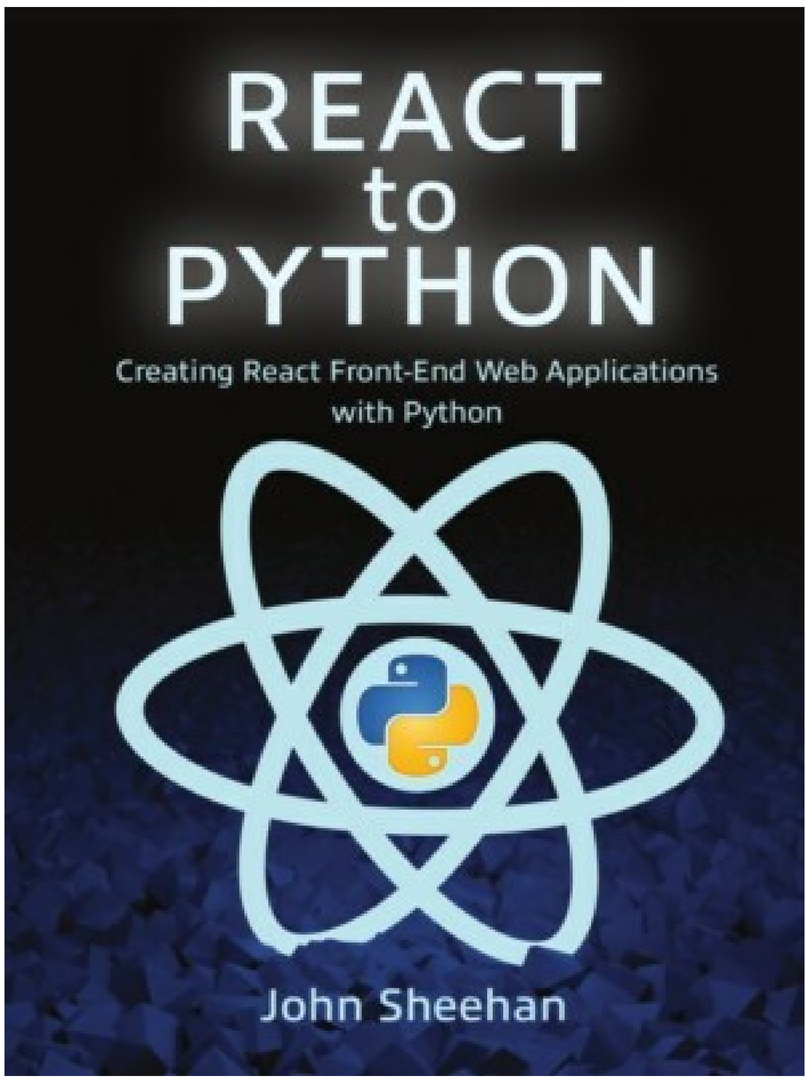


And with this, we have the foundational basis for building React applications using Python!

For more...

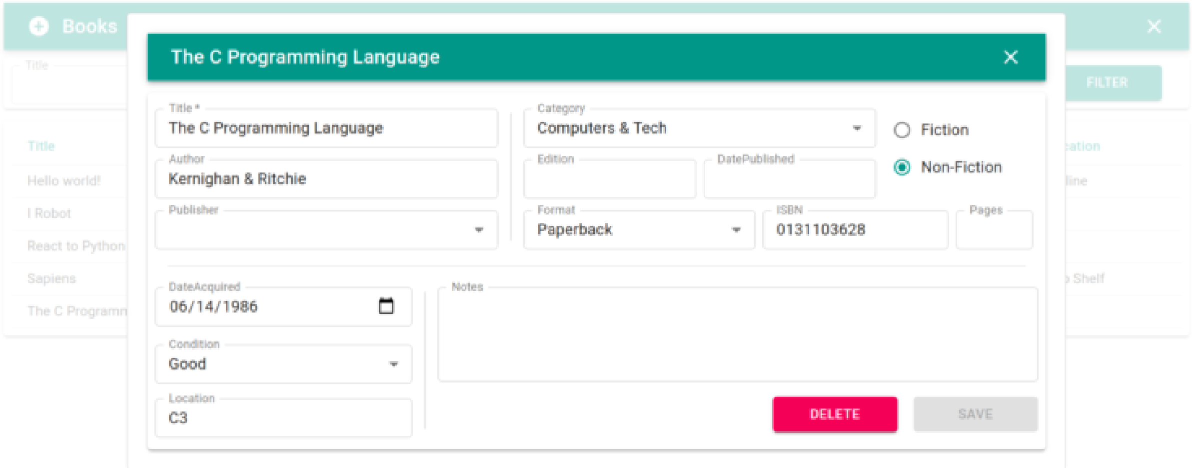
3.1 React to Python

If you are interested in learning more details about what is presented here, the React to Python book dives a lot deeper into what is needed to develop complete web applications using this approach.



The book Includes:

- Setting up the required developer environment tools
- Creating CRUD Forms
- Asynchronous requests with a Flask REST service
- Basics of using the Material-UI component library
- Single Page Applications
- Basic user session management
- SPA view Routing
- Incorporating Google Analytics into your application
- Walks you through building a complete demo project



3.2 Resources

- Source Code:
  https://github.com/JennaSys/rtp_demo
- Transcrypt Site:
  https://www.transcrypt.org
- Transcrypt GitHub:
  https://github.com/qquick/Transcrypt
- React to Python Book:
  https://pyreact.com

Discount Offers!

If you are looking to further explore using Python to create React applications and are interested in purchasing the *React to Python* book, the following discounts are available for a limited time:

- Use the link below to ***get 30% off*** the list price of the E-book on *Leanpub* (you must use this link!):

  https://leanpub.com/rtp/c/rtp21tutorial30

- Use the coupon code **`rtp21tutorial20`** at checkout on *Aerio/Ingram* to ***get 20% off*** the list price of a print copy of *React to Python*:

  https://shop.aer.io/JennaSys

These offers expire December 31, 2021 and are only valid at the points of purchase indicated.