CS50’s Web Programming with Python and JavaScript

OpenCourseWare

Brian Yu
brian@cs.harvard.edu

David J. Malan
malan@harvard.edu

CS50 Certificate

Ed Discussion for Q&A
Quick Start Guide

edX
iTunes U
YouTube

Discord
Facebook Group
Facebook Page

- Instagram
- LinkedIn Group
- Quora
- Snapchat
- Twitter
- YouTube

CC BY-NC-SA 4.0

Python and Flask

Python

- In this class, Python 3.6 will be used. For those unfamiliar, Python is an interpreted language that will be used in the context of this class to generate dynamic websites and web applications.
- Some basic Python syntax:
  - Print a string to the screen:

    ```
    print("Hello, world!")
    ```

  - Print a format string (variable names enclosed with `{}` will be replaced by variable values)

    ```
    print(f"Hello, {name}!")
    ```

- Set variable `name` to the user input returned by `input()`

  ```
  name = input()
  ```

- Conditional statement:

  ```
  if x > 0:
      print("x is positive")
  elif x < 0:
      print("x is negative")
  else:
      print("x is zero")
  ```

  - `elif` and `else` blocks are optional.
  - Note that indentation in Python is not stylistic, but rather is used to demarcate blocks of code. In this example, the Python interpreter knows where the conditional `if` block ends and the `elif` block begins because of the changes in indentation.

Data Types

- `int` : integer value
- `float` : floating point value
- `str` : text string
- `bool` : boolean value (`True` or `False`)
- `None` : empty value
- Note that Python is a weakly typed language.

Sequences

- Strings:

  ```
  name = "Alice"
  print(name[0])
  ```

  - Strings are justs sequence of characters, and can be indexed as such.
- Tuples:

  ```
  coordinates = (10.0, 20.0)
  print(coordinates[1])
  ```

  - Tuples are immutable collections of values under a single name, which can be indexed positionally.
- Lists:

```
names = ["Alice", "Bob", "Charlie"]
print(names[2])
```

- Lists are mutable collections of values under a single name, which can be indexed positionally.
- Indexing out of range raises a Python 'exception'. In this case, an `IndexError`, because there is no fourth value in `names` for Python to return.

- Note that any sequence in Python can contain any number of data types.
- Sets:

```
s = set()
s.add(1)
s.add(3)
s.add(5)
```

- Sets are unordered collection of unique items. Because they are unordered, they cannot be indexed.
- `s` is a `set`, an unordered collection of unique items
- Dictionaries:

```
ages = {"Alice": 22, "Bob": 27}
print(ages["Alice"])
ages["Alice"] += 1
```

- Dictionaries (or dicts) are like lists, except that they are unordered and their `value`s are indexed by `keys`.
- The `+=` operator increments the left-hand side by the right-hand side.

Loops

```
for i in range(5):
    print(i)
```

- For-loops iterate over their bodies a limited number of times. In this case, the number of iterations is set by `range(5)`.
- `range(5)` returns the sequence of numbers starting at 0 through 4. Each value is passed to `i` once, resulting in the loop running a total of 5 times. `i` is normally referred to as an iterator variable.

```
for name in names:
    print(name)
```

- This for-loop iterates over `names`, which is a list. Every value in the list is assigned, in order, to the iterator `name` once.

Functions

- Python has built-in functions, such as `print()` and `input()`, but Python also allows for the creation of user-defined functions

```
def square(x):
    return x * x
```

  - This is a function called `square`, which takes a single argument `x`, and returns the value `x * x`.
  - Like loops, the body of a function must be indented.

  ```
  for i in range(10):
  print("{} squared is {}".format(i, square(i)))
  ```

    - This loop, which prints out the results of `square` with a range of arguments, using an older method for format strings.
- Trying to call a function that hasn't been defined will raise a `NameError` exception.

Modules

- Modules are separate `.py` files of code, often written by others, used in a new file without rewriting all the old code again. Using modules allows, for example, the use of functions across a program larger than a single file.
- Assuming the `square` function in the earlier example was saved in `functions.py`, adding this line atop a new module will allow for the use of `square` there as well.

```
from functions import square
```

- If, for example, `functions.py` also included the example loop demonstration of the `square` function, that loop would be executed every time `square` was imported from `functions`, because the Python interpreter reads through the entire `functions.py` file. To remedy this, code that should only run when their containing file is run directly should be encapsulated in a function, called, for example, `main`. After, the following should be appended:

```
if __name__ == "__main__":
    main()
```

This should be interpreted as saying 'if this file is currently being run', execute `main`.

Classes

- A Python `class` can be thought of as a way to define a new, custom Python data type, somewhat analagous to defining a custom function.
- This creates a new `class` called `Point`:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

  - The `__init__` function is a special function that defines the information needed when a new `Point` is created. `self` is always required, which refers to the `Point` being created, while `x` and `y` are its coordinates.
  - `self.x` and `self.y` actually do the work of creating `x` and `y` attributes for the `Point` and assigning the m the values passed to `__init__`.
  - By convention, `class` names tend to start with a capital letter.
- This instantiates a new `Point` with `x = 3` and `y = 5`:

```
p = Point(3, 5)
```

  - When this line is run, the `__init__` function of the `Point` class is automatically run.
- To access the `x` attribute of `p`, use dot notation:

```
print(p.x)
```

Flask

- HTTP (Hypertext Transfer Protocol) is the system the internet uses to interact and communicate between computers and servers. When a URL is entered into a browser, an HTTP request is sent to a server, which interprets the request and sends appropriate HTTP response, which, if all goes as expected, contains the requested information to be displayed by the web browser.
- Having already begun to design websites, the next step is to write the code that takes care of the server-side processing: receiving and interpreting requests, and generating a response for the user.
- Flask a microframework written in Python that makes it easy to get a simple web application up and running with some features that can be useful in the development process.

A Simple App

- Flask code is generally stored inside `application.py`, and might look like so:

```
  from flask import Flask # Import the class `Flask` from the `flask` module, written
by someone else.

  app = Flask(__name__) # Instantiate a new web application called `app`, with
`__name__` representing the current file

  @app.route("/") # A decorator; when the user goes to the route `/`, exceute the
function immediately below
  def index():
      return "Hello, world!"
```

  - Flask is designed in terms of routes. A route is the part of the URL that determines which page is being requested. The route for the default page is simply `/`.
- To start up a flask application, run `flask run` in the directory where `application.py` is located, with `flask` being the web server. Flask will print out the URL the server is running on and where the website can be accessed at.
  - `flask run` produces an error, try running `export FLASK_APP=application.py` to make sure it knows to look for `application.py` as the web server.

Fancier Flask and Jinja2

```
  @app.route("/<string:name>")
  def hello(name):
      return f"Hello, {name}!"
```

- When any string is entered as a route, that will be stored as `name`, which is can then be used inside the decorated function.
- Since Python code is rendering the website, anything Python is capable of can be used. For example, `name` can be capitalized before it’s displayed:

```
name = name.capitalize()
```

- HTML can also be used inside the return value:

```
return f"<h1>Hello, {name}!</h1>".
```

- Inline HTML isn’t that useful, though. Separate HTML files can be used like so:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def index():
    return render_template("index.html")
```

- `index.html` and any other template files should be stored in a directory named `templates`.
- Variables can be defined as Python variables in `application.py` and used in HTML templates by passing them in as arguments to `render_template`. These templates are rendered using a separate templating language called Jinja2:
- In `application.py`:

```
headline = "Hello, world!"
return render_template("index.html", headline=headline)
```

- In `index.html`:

```
<h1>{{ headline }}</h1>
```

- Jinja2 also allows for conditional statements:

```
{% if new_year %}
    <h1>Yes! Happy New Year!</h1>
{% else %}
    <h1>No.</h1>
{% endif %}
```

- Loops:

```
{% for name in names %}
    <li>{{ name }}</li>
```

```
{% endfor %}
```

- `names` should be something that can be looped over, like a Python list, for example.
- If there are multiple routes on the Flask server, then one route can link to another as so:

```
<a href="{{ url_for('more') }}">See more...</a>
```

- `more` is the name of a function associated with a route.

Template Inheritance

- In order to cut down on repetitive HTML amongst many different pages, Jinja2 has a feature called 'template inheritance' that uses the idea of `block`s to organize content. For examples, have a look at `layout.html` and `index.html` in the `inheritance/` directory of the same source code.
- Everything in the `heading` block is placed where indicated in `layout.html`, and same for `body`.

Forms

- With Flask and Jinja2, the results from HTML forms can now be actually stored and used.
- An HTML form might look like this:

  ```
  <form action="" method="post">
      <input type="text" name="name" placeholder="Enter Your Name">
      <button>Submit</button>
  <form>
  ```

  - The `action` attribute lists the route that should be 'notified' when the form is submitted. In this case, it's the URL for a function called `hello`.
  - The `method` attribute is how the HTTP request to submit the form should be made. The default method is `get`, which is what browsers make when a URL is entered. When data is being submitted, however, `post` should be used.
  - The `name` attribute of the input, while not new, is now relevant because it can be referenced when the form is submitted.
- The Python code to process this form might look like this:

```
from flask import Flask, render_template, request
```

```
# some lines omitted here

@app.route("/hello", methods=["POST"])
def hello():
    name = request.form.get("name") # take the request the user made, access the form,
                                    # and store the field called `name` in a Python
variable also called `name`
    return render_template("hello.html", name=name)
```

- The route `/hello` is the same `hello` listed in the Jinja2 code. This route can also accept the `POST` method, which is how the form's data is being submitted. If any other method is used to access this route, a `Method Not Allowed` error will be raised.
    - If there are multiple request methods that should be allowed, which method is being used can be checked with `request.method`, which will be equal to, for example, `"GET"` or `"POST"`.

Sessions

- Sessions are how Flask can keep track of data that pertains to a particular user. Let's take a note-taking app, for example. Users should only be able to see their own notes.
- To use sessions, they must be imported and set up:

```
from flask import Flask, render_template, request, session # gives access to a variable
called `session`
                                                           # which can be used to keep
vaules that are specific to a particular user
from flask_session import Session # an additional extension to sessions which allows them
                                  # to be stored server-side

app.config["SESSION_PERMANENT"] = False
app.config["SESSION_TYPE"] = "filesystem"
Session(app)
```

- Then, assuming there is some HTML form that can submit a note, the note can be stored in a place specific to the user using their session:

```
@app.route("/", methods=["GET", "POST"])
def index():
    if session.get("notes") is None:
        session["notes"] = []
    if request.method == "POST":
        note = request.form.get("note")
        session["notes"].append(note)

    return render_template("index.html", notes=session["notes"])
```

- `notes` is the list where the notes will be stored. If the user doesn't have a notes list already (checked with `if session.get("notes") is None`), then they are given an empty one.
- If a request is submitted via `"POST"` (that is, through the form), then the note is processed from the form in the same way as before.
- The processed note, now in a Python variable called `note`, is appended to the `notes` list. This list is itself inside a `dict` called `session`. Every user has a unique `session` `dict`, and therefore a unique `notes` list.
- Finally, the notelist is rendered by passing `session["notes"]` to `render_template`.